Dezvoltarea Aplicațiilor Web utilizând ASP.NET Core MVC
Curs 6

Cuprins

Baze de Date – Noțiuni generale

Ce este o bază de date

O **bază de date** este un ansamblu structurat de date coerente, fără redundanță **inutilă**, astfel încât datele pot fi prelucrate eficient de mai mulți utilizatori, într-un mod concurent.

Ce este un SGBD/DBMS

Un SGBD este un sistem de gestiune a bazelor de date. Denumirea de SGBD este echivalenta cu cea de DBMS (**Database Management System**), reprezentând un produs software care asigură interacțiunea cu o bază de date. Un **DBMS** stochează date în tabele, pe care ulterior le accesează și prelucrează, cu ajutorul unui limbaj **SQL (Structured Query Language)**. Un DBMS asigură securitatea, integritatea și consistența datelor.

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

Cerințe minimale ale bazelor de date

O bază de date trebuie să îndeplinească următoarele cerințe minimale:

- Redundanță minimă;
- Sincronizarea datelor – utilizarea simultană a datelor de către mai mulți utilizatori;
- Securitatea datelor;
- Integritatea datelor – date corecte, posibilitatea recuperării lor;
- Furnizare rapidă a datelor – cereri eficiente;
- Flexibilitatea datelor – adaptarea rapidă la cerințe noi;

Ce este cheia primară

Cheia primară este un identificator **UNIC** în cadrul entităților. Ea trebuie să fie cunoscută în orice moment – ceea ce înseamnă că **nu poate fi null**

Cu ajutorul cheii primare se identifică unic intrările dintr-un tabel al bazei de date.

Ce este cheia externă

Cheia externă poate fi ori null în întregime, ori trebuie să refere cheia primară din tabelul de legătură (adică să corespundă unei valori a cheii primare asociate).

Ce este o cheie primară compusă

Cheia primară compusă se creează în momentul în care se combină două sau mai multe coloane în cadrul unui tabel, formând un tuplu, care mai apoi este utilizat ca identificator unic în cadrul tabelului respectiv. Tuplul o să identifice unic fiecare intrare din tabelul în care se află.

Ce este o entitate

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

Exemplu de entități dintr-o bază de date care gestionează o Universitate → Studenți, Cursuri, Note, etc

În modelul Entitate/Relație, entitățile sunt substantive. În modelele relaționale entitățile devin tabele.

Nu pot exista în aceeași diagramă două entități cu același nume sau aceeași entitate având un nume diferit.

Ce este o relație

O **relație** este o asociere dintre două sau mai multe entități. În modelul Entitate/Relație acestea sunt verbe. În modelul relațional, relațiile devin fie tabele speciale, fie coloane care referă chei primare.

Relațiile au asociată și o cardinalitate, existând trei tipuri de cardinalități:

- **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 să aibă un tip de date, un nume și constrângeri.

Diagrama Entitate/Relație

Diagrama Entitate/Relație 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 implementam;
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 (atributele 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 știri**, având următoarele reguli de proiectare:

- Să existe cel putin 4 tipuri de utilizatori: vizitator neînregistrat, utilizator înregistrat, editor și administrator;
- Orice utilizator poate vizualiza știrile apărute pe site. Pe pagina principală vor aparea știrile cele mai recente, în funcție de data la care au fost postate. În acest caz, se alege un număr de x știri pentru afișare;
- Știrile vor fi împărțite pe categorii (create dinamic de către administrator): știință, tehnologie, sport, etc, existând posibilitatea de adăugare a noi categorii (administratorul poate face CRUD pe categorii);
- Știrile o să se afișeze paginat, alegându-se un număr x de știri pe o pagină;
- O să se includă și un editor de text, astfel încât să se editeze textul în momentul în care un editor publică o nouă știre. Știrea se poate scrie folosind și elemente de markup;
- Editorii se ocupă de publicarea știrilor noi și pot vizualiza, edita, șterge propriile știri;
- Utilizatorii care au cont pot adăuga comentarii la știrile apărute, își pot șterge și edita propriile comentarii;
- Utilizatorii care nu au cont pot să vadă doar pagina principală a aplicației. Dacă doresc să citească știri și să interacționeze cu alți utilizatori, aceștia vor fi redirecționați către pagina de înregistrare/autentificare;
- Știrile pot fi căutate prin intermediul unui motor de căutare propriu, în funcție de titlu, conținut, sau chiar conținutul comentariilor;
- Administratorii se ocupă de buna funcționare a întregii aplicații (ex: pot face CRUD pe știri, pe categorii, pe utilizatori etc.) și pot activa sau revoca drepturile utilizatorilor și editorilor;

PASUL 1 – Identificarea entităților

- să existe cel putin 4 tipuri de utilizatori → **USER**;
- vizitator neînregistrat, utilizator înregistrat, editor și administrator → **ROLE**;
- orice utilizator poate vizualiza știrile → **ARTICLE**;
- știrile vor fi împărțite pe categorii → **CATEGORY**;
- utilizatorii pot adăuga comentarii la știrile apărute → **COMMENT**;
- restul funcționalităților se implementează folosind logica de backend (cod), nefiind nevoie de alte entități/tabele;

/!\ OBSERVAȚIE

În **Entity Framework**, clasele se denumesc la singular, deoarece sistemul convertește automat fiecare clasă într-un tabel, pluralizând numele. Se definește clasa **Article.cs** în Models, clasă care o să devină automat tabelul **Articles** (prin intermediul proprietății **DbSet** din clasa care definește contextul bazei de date – această clasă este cea derivată din clasa de bază **DbContext**).

PASUL 2 – Identificarea relațiilor și a cardinalităților

Avem următoarele entități identificate: **USER, ROLE, ARTICLE, COMMENT, CATEGORY**

- Să existe cel puțin 4 tipuri de utilizatori: vizitator neînregistrat, utilizator înregistrat, editor și administrator → **un utilizator poate avea un singur rol**;
  /!\ OBS → având în vedere că Entity Framework utilizează în acest caz o relație de tip **many-to-many**, vom implementa și noi o astfel de relație, dar o vom utiliza ca fiind one-to-many, făcând asocierea UNUI utilizator cu UN singur rol – ca logică în aplicație;

- Editorii se ocupă de publicarea știrilor noi și pot vizualiza, edita, șterge propriile știri. Un user de tip **EDITOR** poate să facă CRUD, iar un user înregistrat poate doar să vizualizeze articole.

  Un editor vizualizează/redactează/editează/șterge/ mai multe articole, iar un articol este vizualizat/creat/editat/șters/ de un singur editor → relația (cardinalitatea maximă) este one-to-many;

  Pentru vizualizare se folosește logica scrisă în cod;

- Știrile vor fi împărțite pe categorii. **Un articol face parte dintr-o singură categorie, iar o categorie conține mai multe articole → cardinalitatea maximă este one-to-many**;

- Utilizatorii pot adăuga comentarii la știrile apărute, își pot șterge și edita propriile comentarii → **un articol conține mai multe comentarii, iar un comentariu aparține unui singur articol → cardinalitate one-to-many**;
  → un user postează mai multe comentarii, iar un comentariu este postat de un singur user → one-to-many;

PASUL 3- Proiectarea diagramei E/R

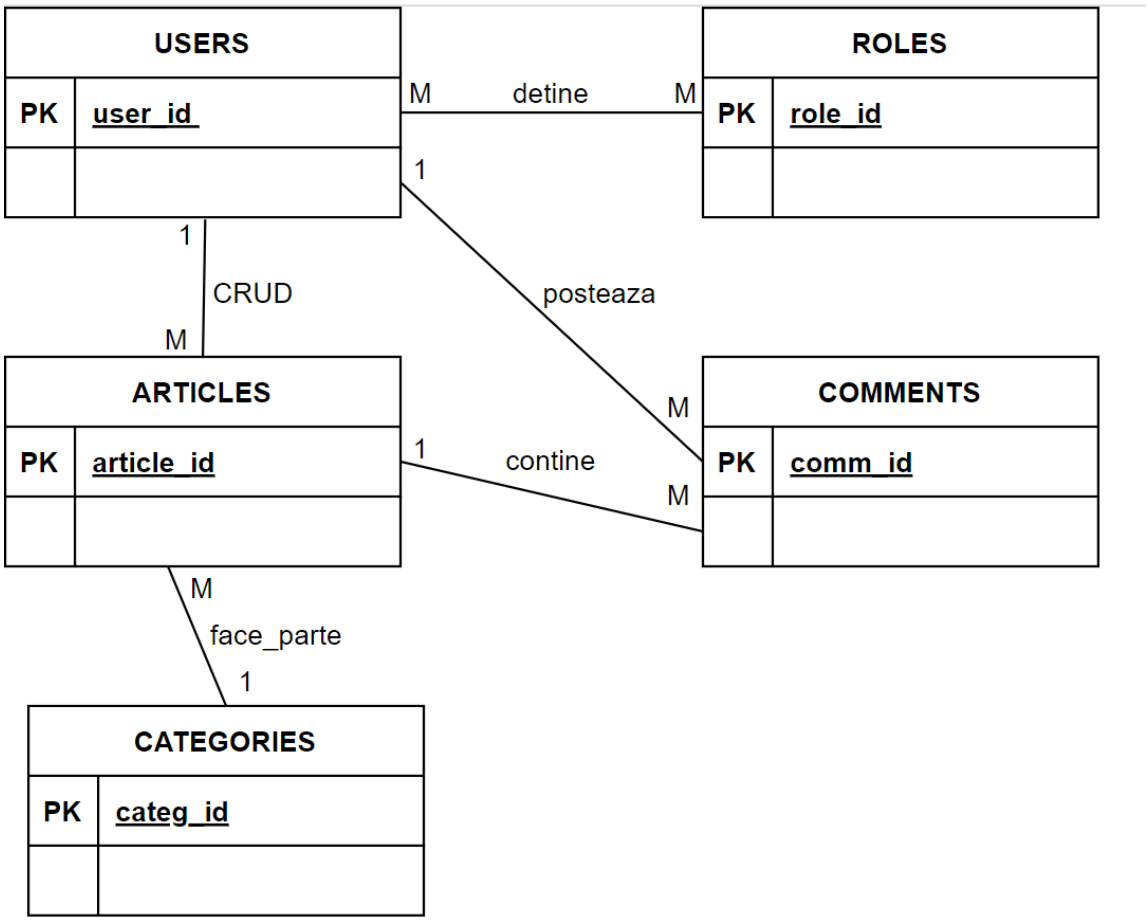


Diagrama Conceptuală

Diagrama Conceptuală este un model formal de organizare a datelor, conținând legătura dintre acestea sub formă de tabele.

Reguli de transformare a diagramei E/R în Diagramă Conceptuală

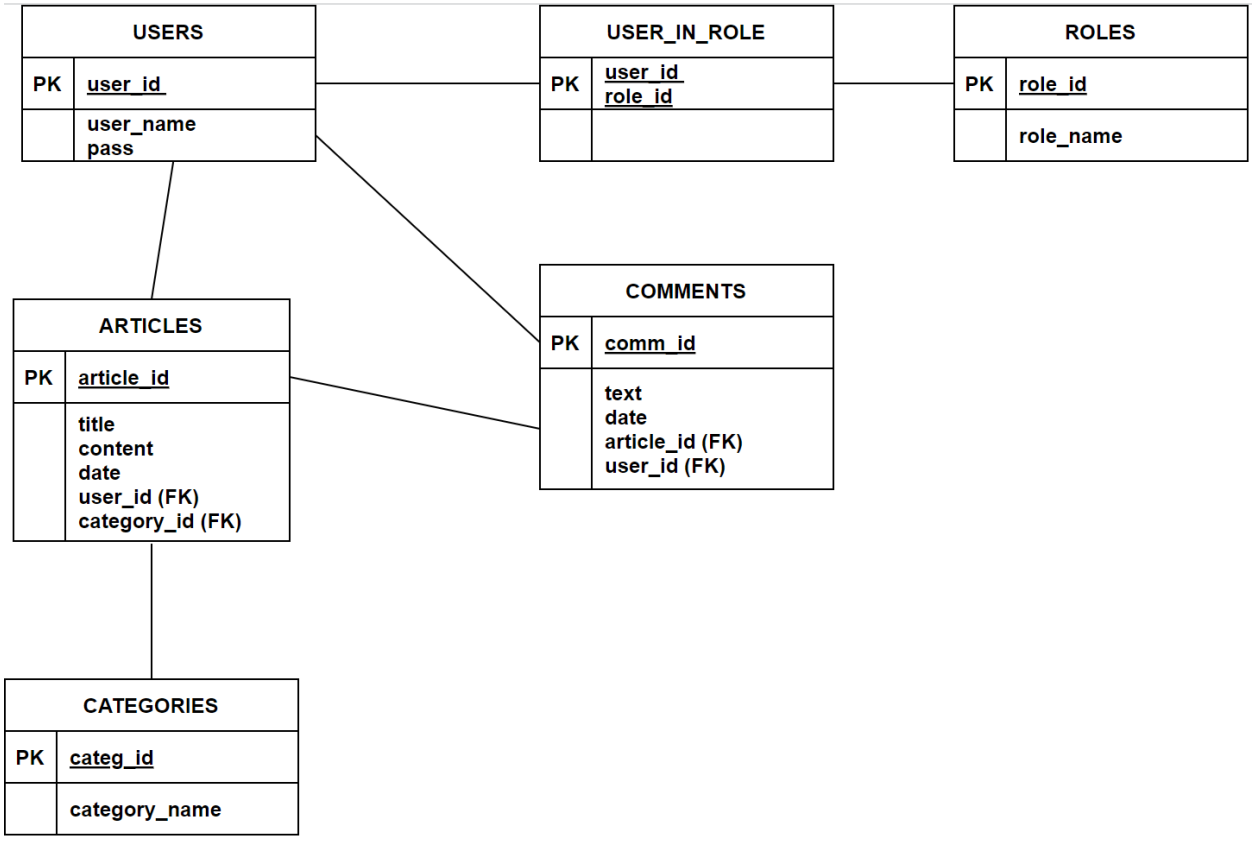
Pentru transformarea Diagramei Entitate/Relație în Diagramă Conceptuală se parcurg următoarele reguli:

- Entitățile devin tabele;
- Relațiile one-to-one și one-to-many devin chei externe
  - În cazul relațiilor **one-to-many**, cheia externă se plasează în tabelul în dreptul căruia se află cardinalitatea **many**;
  - În cazul relațiilor **one-to-one**, cheia externă se plasează în tabelul care conține mai puține intrări în baza de date (din motive de performanță/eficiență);
- Relațiile many-to-many devin tabele asociative;

Exemplu pentru proiectarea Diagramei Conceptuale

Urmând regulile anterioare, transformăm Diagrama Entitate/Relație din secțiunea **Exemplu practic pentru proiectarea Diagramei E/R – PASUL 3**, în Diagramă Conceptuală.

Diagrama Conceptuală:



Adăugarea sistemului de autentificare

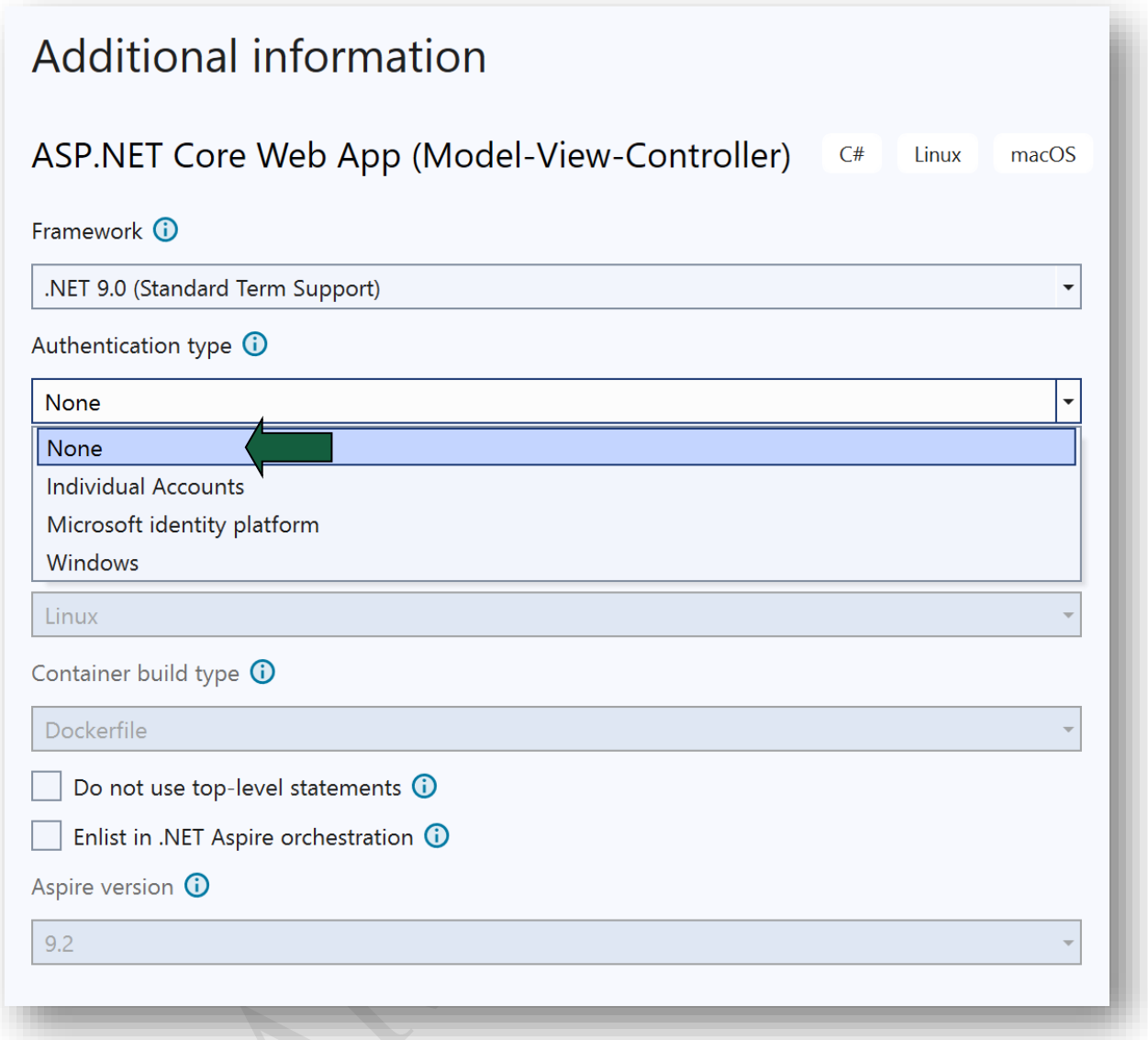
Framework-ul ASP.NET Core oferă posibilitatea integrării unui sistem de autentificare, folosind **Identity Framework**.

Identity Framework este considerat un framework în cadrul ecosistemului .NET. Acesta oferă servicii preconfigurate pentru gestionarea utilizatorilor și a datelor lor de autentificare. Utilizează, de obicei, **Entity Framework Core** pentru a stoca datele într-o bază de date, dar poate fi configurat și pentru alte sisteme de stocare. Este compus dintr-o suită de clase și secvențe de cod, care facilitează implementarea rapidă a unui sistem de autentificare complex.

ASP.NET Core Identity:

- oferă posibilitatea autentificării folosind user și parolă;
- asigură gestionarea utilizatorilor (crearea, modificarea, ștergerea);
- asigură gestionarea rolurilor utilizatorilor și a drepturilor acestora;
- permite resetarea parolelor și validarea prin email;
- permite autentificarea externă, folosind conturi 3rd party (autentificare prin rețele de socializare – Google, Facebook, Twitter, Microsoft, etc).

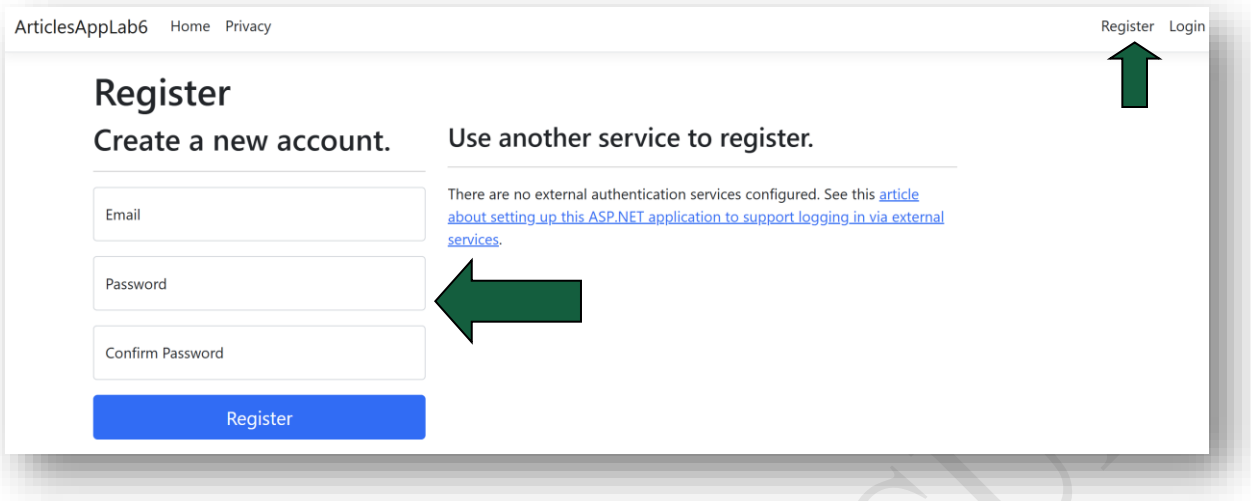
Pentru a genera un proiect care include componenta **Identity** pentru autentificare, trebuie să alegem la crearea proiectului forma de autentificare: **Individual Accounts**.



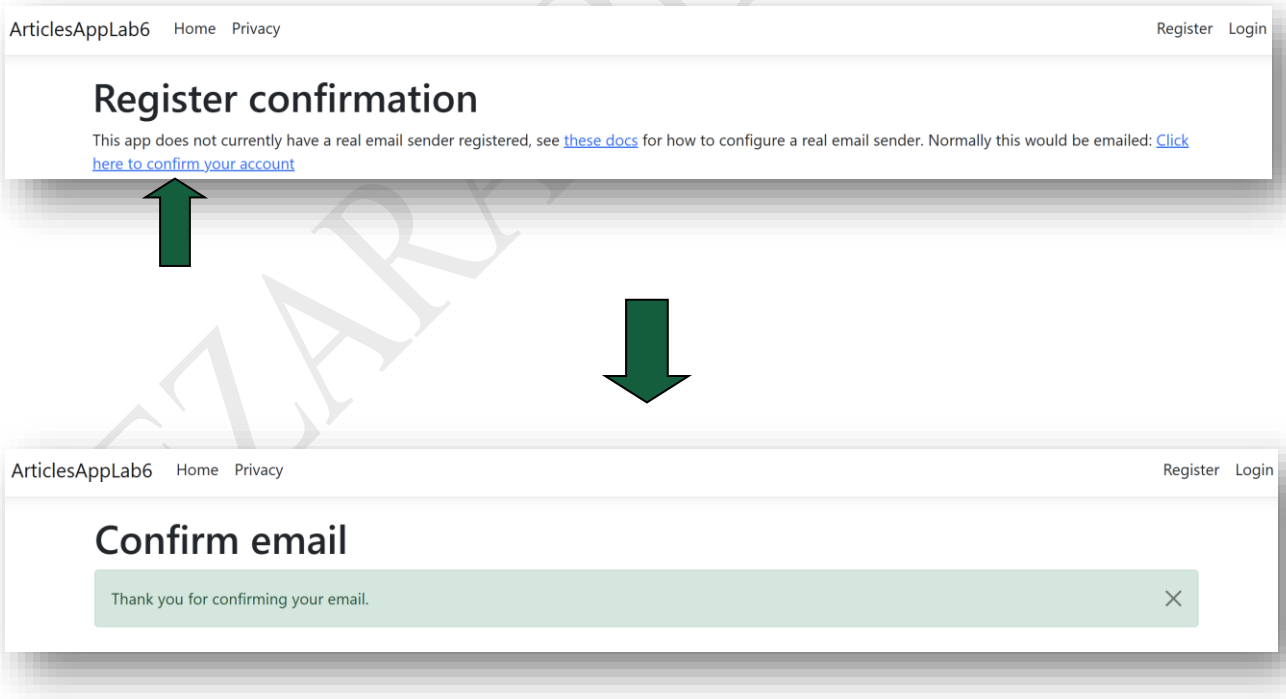
Proiectul nou creat conține **Identity Framework** și toate mecanismele aferente autentificării.

Înainte de rularea proiectului, trebuie executată în consolă (Tools → NuGet Package Manager → Package Manager Console) comanda **Update-Database**.

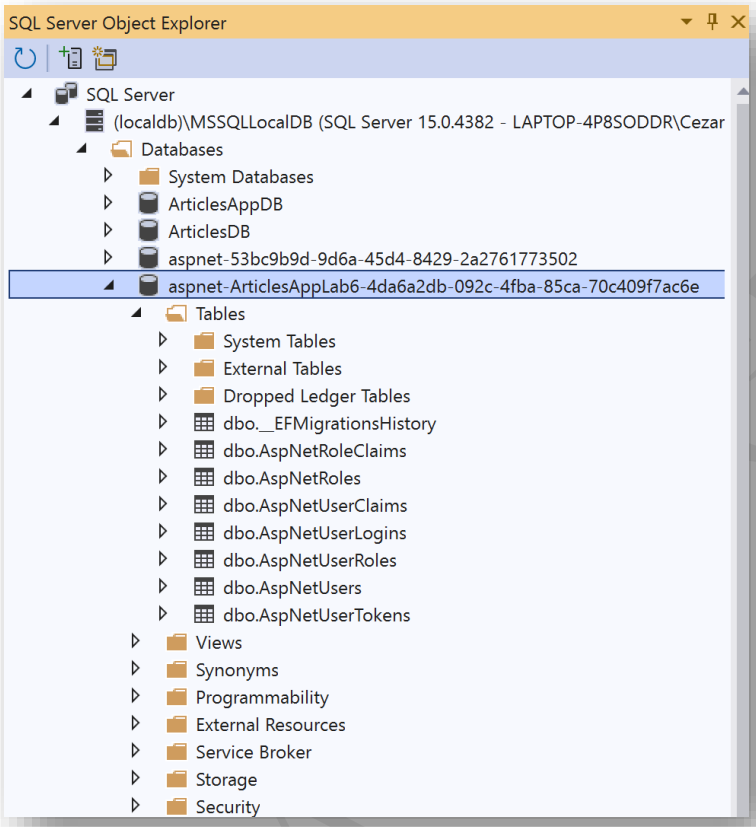
După rularea comenzii, ne putem înregistra cu un cont.



După înregistrare se apasă **Click here to confirm your account** deoarece nu există inclus serviciul de trimitere de e-mailuri.

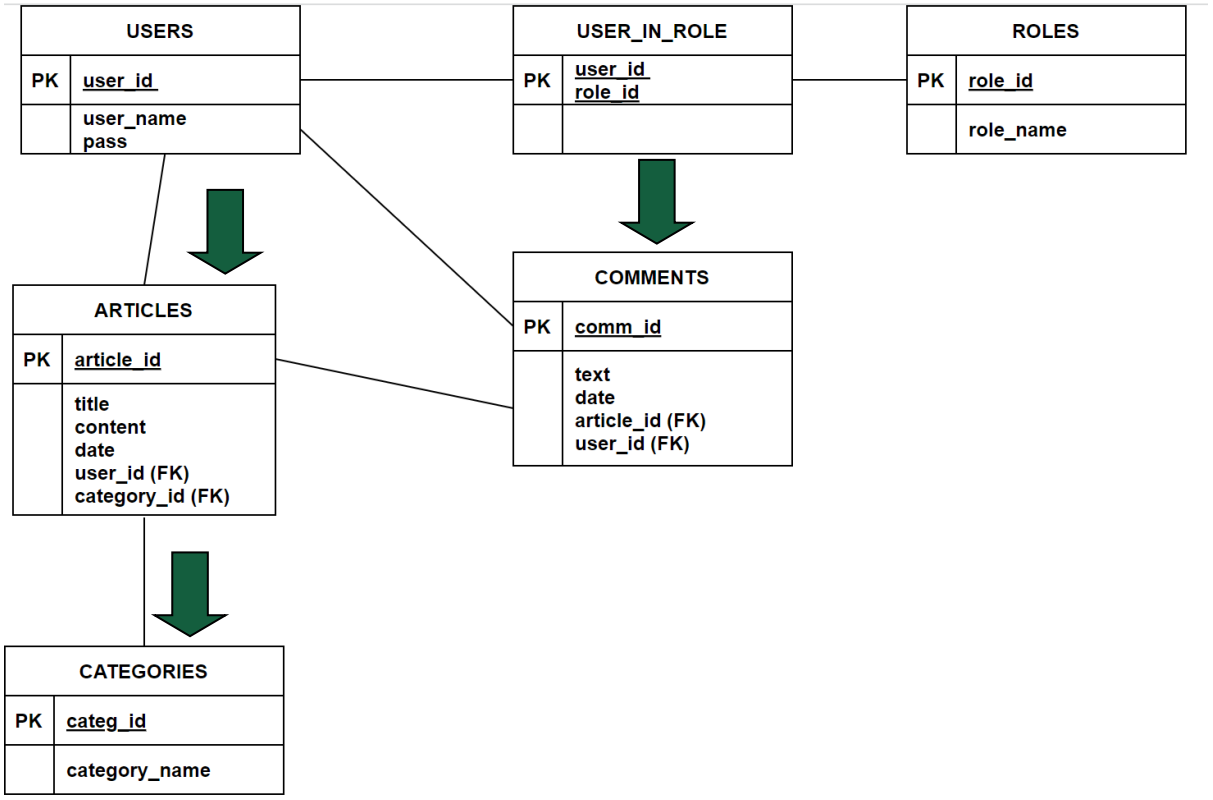


În proiect → SQL Server Object Explorer → se pot vizualiza tabelele create pentru sistemul de autentificare și pentru rolurile pe care le pot avea utilizatorii.



Implementare cereri folosind Entity Framework Core și LINQ

Pe site, în cadrul secțiunii SAPTAMANA 6, este un document cu mai multe exemple de implementare, utilizând LINQ. Documentul se numește **Linq-Cheatsheet.pdf**



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

Article

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

Category:

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

Comment:

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

Implementarea claselor folosind Entity Framework Core

Clasa ARTICLE:

```
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; }
}
```

Clasa CATEGORY:

```
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; }
}
```

Clasa COMMENT:

```
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; }
}
```

/!\ OBSERVAȚIE

După implementarea claselor, se execută migrațiile:

- Add-Migration NumeMigratie
- Update-Database

JOIN

1. Să se afișeze articolele împreună 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 în consolă:

```
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]
```

2. Să se afișeze articolele, categoria din care fac parte și comentariile asociate. Să se afișeze titlul articolului, numele categoriei, comentariul și 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` și `Comments` sunt proprietățile din clasa Article

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

Codul asociat generat în consolă:

```
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]
```

Lazy Loading

Aceste proprietăți sunt **virtuale** pentru a se executa **lazy loading**.

Lazy loading este un design pattern folosit în dezvoltarea web, care întârzie încărcarea anumitor resurse pentru a îmbunătăți performanța.

În cazul nostru, lazy loading se realizează prin declararea proprietăților de tip **virtual** și duce la încărcarea claselor doar în momentul în care sunt folosite proprietățile respective. Un exemplu în acest sens ar fi utilizarea joinului folosind **Include**, dar nefolosirea proprietății **Category**. În acest caz vom observa că, indiferent dacă există în join, proprietatea Category nu o să fie încărcată, deoarece nu a fost utilizată pentru afișare în query.

```
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]
```


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

GROUP BY și funcții grup (de exemplu COUNT)

Pentru fiecare categorie să se afișeze numărul de articole care fac parte din categoria respectivă. O să se afișeze id-ul și numele categoriei, împreună cu numărul de articole care fac parte din categoria respectivă.

```
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 în consolă:

```
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]
```

În același mod, se pot selecta în continuare informațiile despre articolele asociate, adăugând următoarea linie de cod în secvența anterioară:

```
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 consolă:

```
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
                }
            ]
        },
```

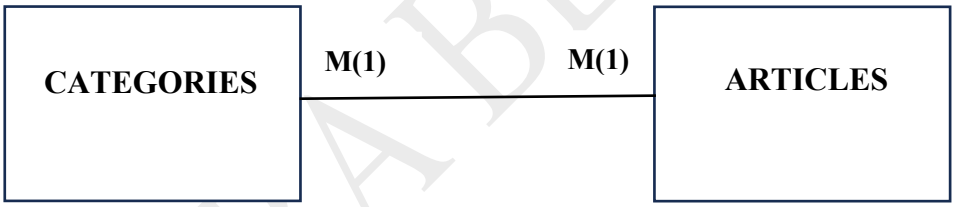
Implementarea relațiilor din baza de date

Implementările următoare au la bază două variante. Ambele sunt utilizate în practică, în funcție de scopul avut.

Relația many-to-many – Varianta 1

În acest caz, relația many-to-many se implementează de la zero, prin suprascrierea convențiilor existente. Această metodă oferă mai multe opțiuni de configurare, având acces facil la atributele modelelor. În exemplul următor este utilizat ***Entity Framework Core – Fluent API***.

Relația este → un articol are mai multe categorii, iar o categorie are mai multe articole



Clasa Article.cs

```
public class Article

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

    public string Title { get; set; }

    public virtual ICollection<ArticleCategory>? ArticleCategories
{ get; set; }

}
```

Clasa Category.cs

```
public class Category

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

    public string CategoryName { get; set; }

   public virtual ICollection<ArticleCategory>? ArticleCategories
{ get; set; }

}
```

Clasa ArticleCategory.cs – o să fie tabelul asociativ din baza de date

```
public class ArticleCategory

{

   public int? ArticleId { get; set; }

   public int? CategoryId { get; set; }

   public virtual Article? Article { get; set; }

   public virtual Category? Category { get; set; }

}
```

Clasa ApplicationDbContext.cs – contextul bazei de date

```
public class ApplicationDbContext : DbContext

{
   public
ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
       : base(options)
   {
   }
```

```
        public DbSet<Article> Articles { get; set; }

        public DbSet<Category> Categories { get; set; }

        public DbSet<ArticleCategory> ArticleCategories { get; set; }


        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {

            base.OnModelCreating(modelBuilder);

            // definire primary key compus

            modelBuilder.Entity<ArticleCategory>()
                .HasKey(ac => new { ac.ArticleId, ac.CategoryId });


            // definire relatii cu modelele Category si Article (FK)

            modelBuilder.Entity<ArticleCategory>()
                .HasOne(ac => ac.Article)
                .WithMany (ac => ac.ArticleCategories)
                .HasForeignKey(ac => ac.ArticleId);

            modelBuilder.Entity<ArticleCategory>()
                .HasOne(ac => ac.Category)
                .WithMany(ac => ac.ArticleCategories)
                .HasForeignKey(ac => ac.CategoryId);
        }
      }
```

Relația many-to-many – Varianta 2 – utilizând proprietățile

În cazul în care se dorește implementarea unei **relații de tip many-to-many,** utilizând convențiile din framework, și anume utilizarea proprietăților în cadrul claselor, se procedează ca în exemplul următor. În acest caz framework-ul **generează automat** tabelul asociativ.

Utilizând această variantă este suficient ca:

- în clasa **Article** să existe o proprietate
  - **public virtual ICollection <Category> Categories {get; set;}**

- iar în clasa **Category** să existe o proprietate:
  - **public virtual ICollection <Article> Articles {get; set;}**

Relația one-to-many – Varianta 1

Dacă varianta suprascrierii convențiilor este mai potrivită, atunci se procedează ca în exemplul următor.

Clasa Article.cs

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

   public string Title { get; set; }

  public int CategoryId { get; set; }

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

Clasa Category.cs

```
public class Category

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

    public string CategoryName { get; set; }

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

}
```

ApplicationDbContext.cs

```
public class ApplicationDbContext : DbContext

{
   public
ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
   {
   }


   public DbSet<Article> Articles { get; set; }

   public DbSet<Category> Categories { get; set; }


   protected override void OnModelCreating(ModelBuilder
modelBuilder)
   {

       base.OnModelCreating(modelBuilder);


       // definire relatii cu modelele Category si Article (FK)

       modelBuilder.Entity<Article>()
           .HasOne<Category>(a => a.Category)
           .WithMany(c => c.Articles)
           .HasForeignKey(a => a.CategoryId);

   }
 }
```

Relația one-to-many – Varianta 2 – utilizând proprietățile

Există și în acest caz două variante de implementare. Cel mai des este utilizată varianta prezentă în secțiunea **Implementare cereri folosind Entity Framework Core si LINQ** din cursul curent, și anume cea în care se utilizează convențiile din cadrul framework-ului. Aceste convenții sunt proprietățile adăugate în cadrul fiecărei clase.

Clasa Article.cs

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

	…

	public virtual Category Category { get; set; }

	…

}
```

Clasa Category.cs

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

	…

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

	…
}
```

Relația one-to-one – Varianta 1

Clasa Article.cs

```
public class Article

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

   public string Title { get; set; }

   public int CategoryId { get; set; }

   public virtual Category Category { get; set; }

}
```

Clasa Category.cs

```
public class Category

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

    public string CategoryName { get; set; }

   public virtual Article Article { get; set; }

}
```

ApplicationDbContext.cs

```
public class ApplicationDbContext : DbContext

{
   public
ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
   {
   }

   public DbSet<Article> Articles { get; set; }

   public DbSet<Category> Categories { get; set; }
```

```
        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {

            base.OnModelCreating(modelBuilder);


            // definire relatii cu modelele Category si Article (FK)

            modelBuilder.Entity<Article>()
                .HasOne<Category>(a => a.Category)
                .WithOne(c => c.Article)
                .HasForeignKey<Article>(a => a.CategoryId);

        }
    }
```

Relația one-to-one – Varianta 2 - utilizând proprietățile

Article.cs

```
public class Article

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

   public string Title { get; set; }

   public int CategoryId { get; set; }

   public virtual Category Category { get; set; }

}
```

Clasa Category.cs

```
public class Category

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

    public string CategoryName { get; set; }

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