

Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

Curs 6

Cuprins

Baze de Date – Notiuni generale	3
Ce este o baza de date	3
Ce este un SGBD/DBMS.....	3
Cerinte minimale ale bazelor de date.....	3
Ce este cheia primara	4
Ce este cheia externa.....	4
Ce este o cheie primara compusa.....	4
Ce este o entitate	4
Ce este o relatie	5
Ce este un atribut.....	5
Exemple – Chei, Entitati, Relatii	5
Diagrama Entitate/Relatie	6
Reguli de proiectare a diagramei E/R	6
Exemplu practic pentru proiectarea Diagramei E/R	6
Diagrama Conceptuala.....	10
Reguli de transformare a diagramei E/R in Diagrama Conceptuala	10
Exemplu pentru proiectarea Diagramei Conceptuale	10
Adaugarea sistemului de autentificare	12
Implementare cereri folosind Entity Framework Core si LINQ.....	15
Implementarea relatiilor din baza de date	27
Relatia many-to-many – Varianta 1 – utilizand proprietatile	27
Relatia many-to-many – Varianta 2.....	27

Relatia one-to-many – Varianta 1 – utilizand proprietatile	30
Relatia one-to-many – Varianta 2	30
Relatia one-to-one – Varianta 1 - utilizand proprietatile	32
Relatia one-to-one – Varianta 2	33

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 concurent.

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 Database, SQL Server, MySQL.

Cerinte minimale ale bazelor de date

O baza de date trebuie sa indeplineasca urmatoarele cerinte minimale:

- 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 entitatile devin tabele.

Nu pot exista in aceeași diagrama două entități cu același nume sau aceeași entitate având un nume diferit.

Ce este o relație

O **relație** este o asocieră dintre două sau mai multe entități. În modelul Entitate/Relație acestea sunt verbe. În modelul relational 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.

Exemple – Chei, Entități, Relații

Un exemplu detaliat se află în documentul numit **Chei_Entități_Relatii** aflat pe site în secțiunea **Săptămâna 6**.

Diagrama Entitate/Relatie

Diagrama Entitate/Relatie este un mod de reprezentare a unui sistem din lumea reala, fiind un model neformalizat. Este compus din entitati si relatiile dintre acestea.

Reguli de proiectare a diagramei E/R

1. Identificarea entitatilor care fac parte din aplicatia pe care dorim sa o implementam
2. Identificarea relatiilor dintre entitati si scrierea acestora in cadrul diagramei
3. Identificarea cardinalitatilor minime si maxime pentru fiecare relatie in parte
4. Identificarea atributelor asociate fiecarei entitati
5. Stabilirea cheilor primare (atributele care identifica in mod unic fiecare entitate)

Exemplu practic pentru proiectarea Diagramei E/R

Sa se proiecteze o baza de date care modeleaza o aplicatie de tip **Engine de stiri** avand urmatoarele reguli de proiectare:

- 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;

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

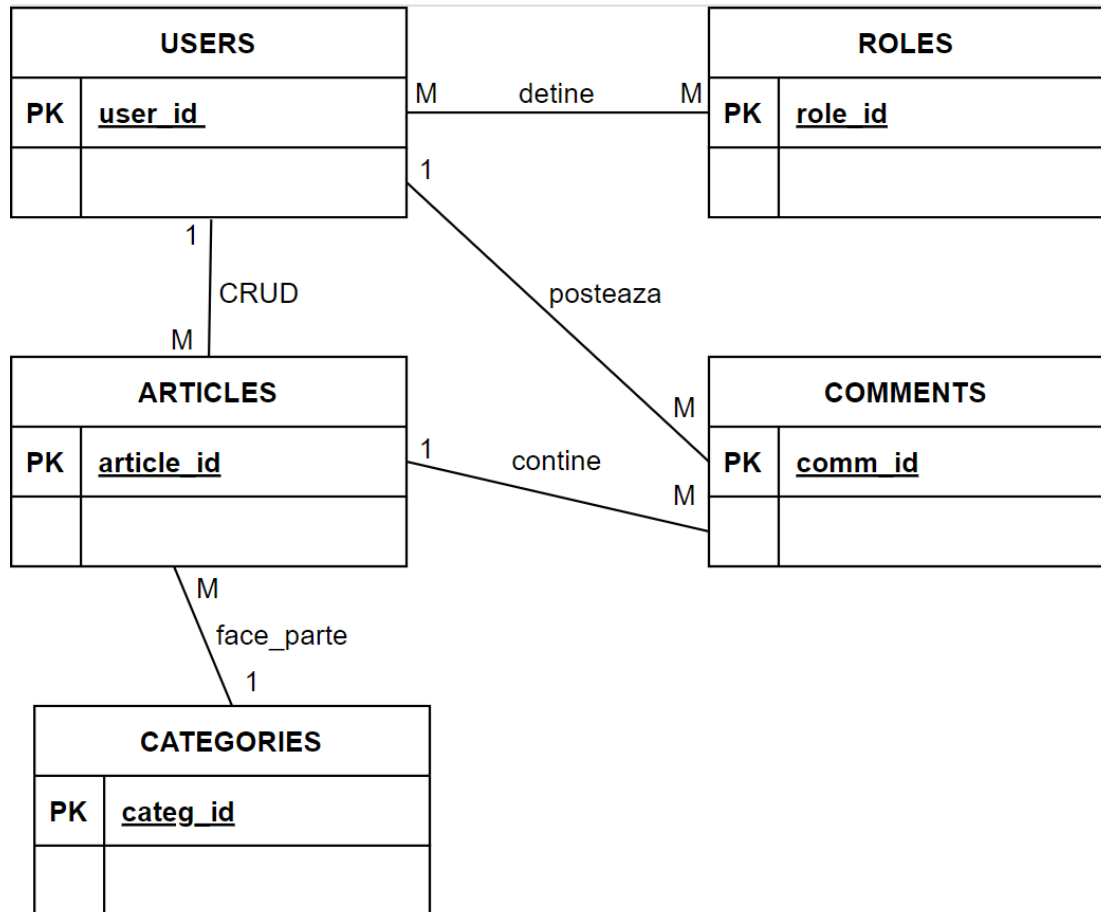


Diagrama Conceptuala

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

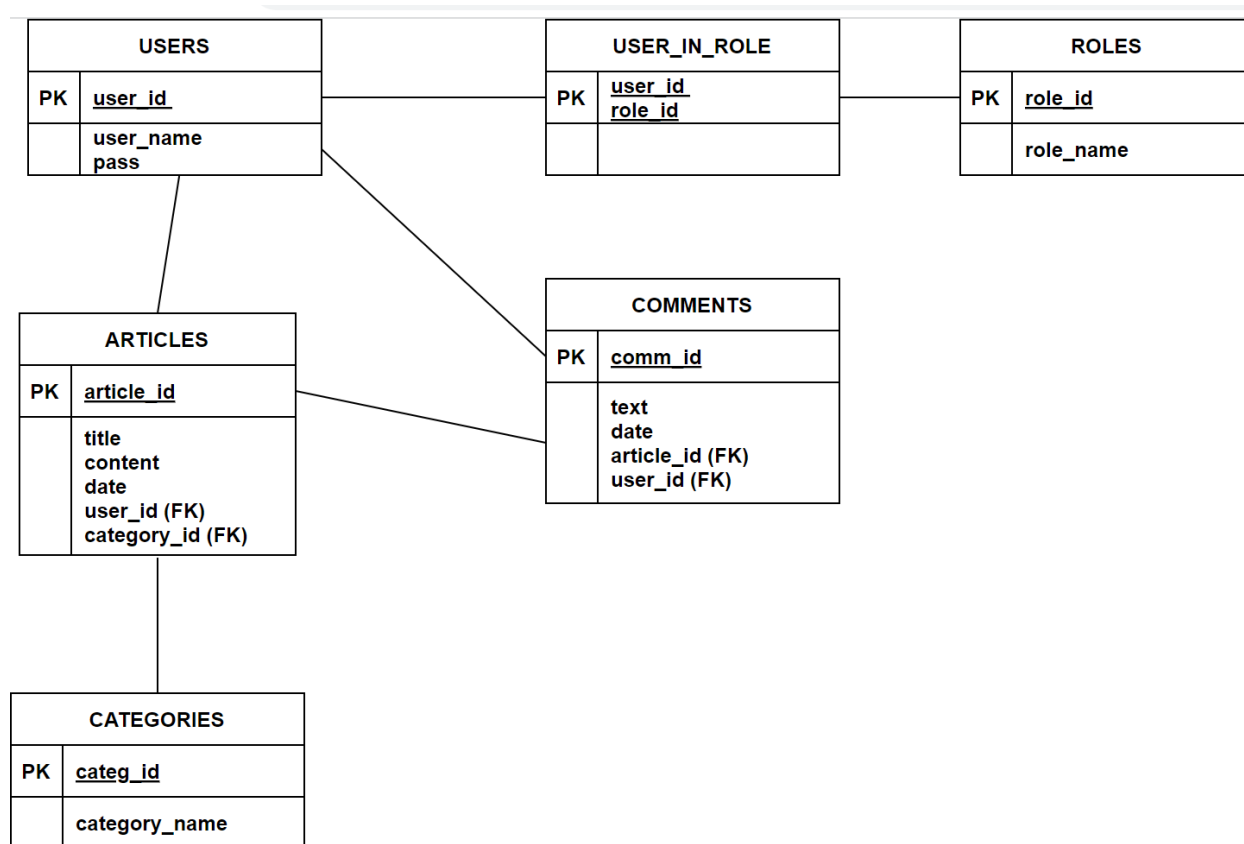
Pentru transformarea Diagramei Entitate/Relatie in Diagrama Conceptuala se urmeaza urmatoarele reguli:

- 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:

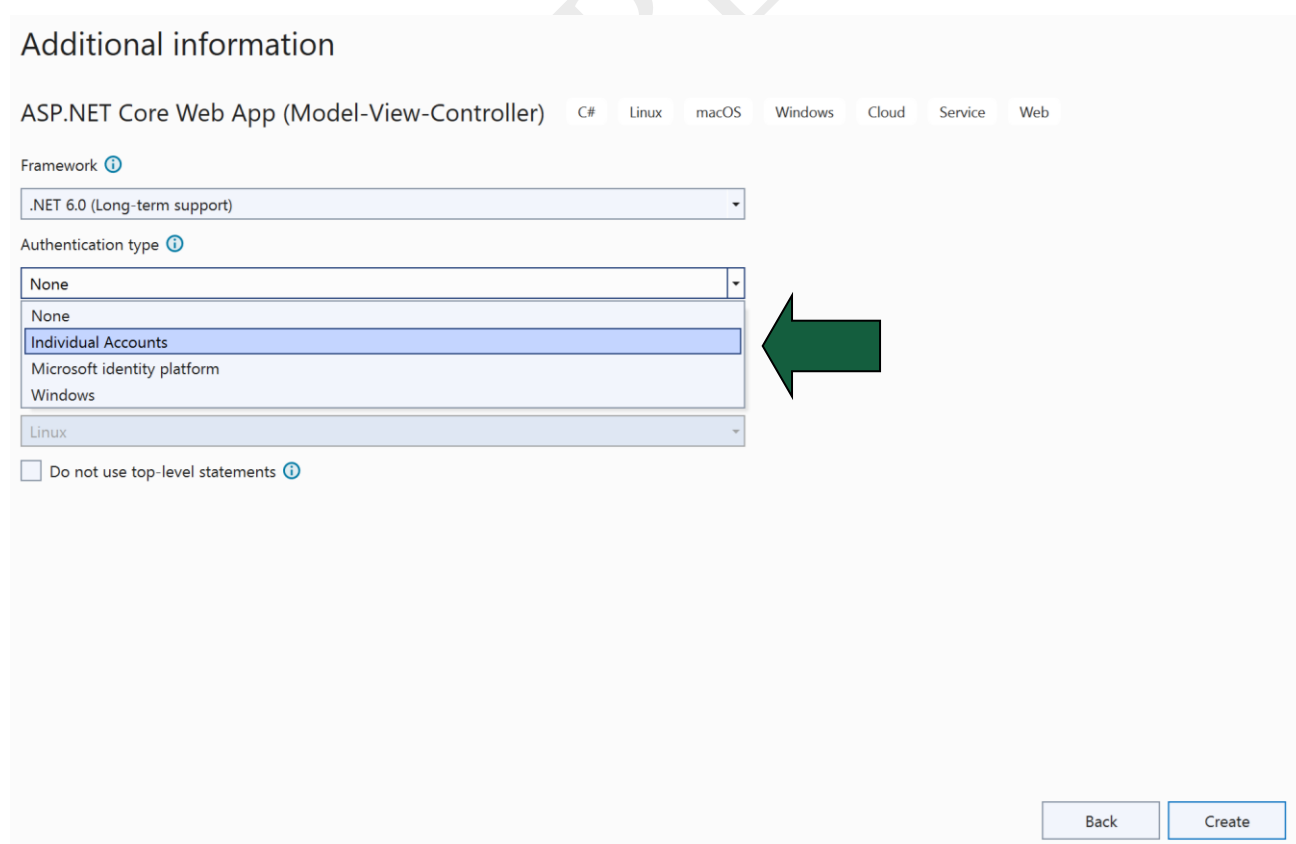


Adaugarea sistemului de autentificare

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

Identity este compus dintr-o suita de clase si secvente 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 3rd 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**.



Additional information

ASP.NET Core Web App (Model-View-Controller) C# Linux macOS Windows Cloud Service Web

Framework ⓘ

.NET 6.0 (Long-term support)

Authentication type ⓘ

None

None

Individual Accounts

Microsoft identity platform

Windows

Linux

☐ Do not use top-level statements ⓘ

Back Create

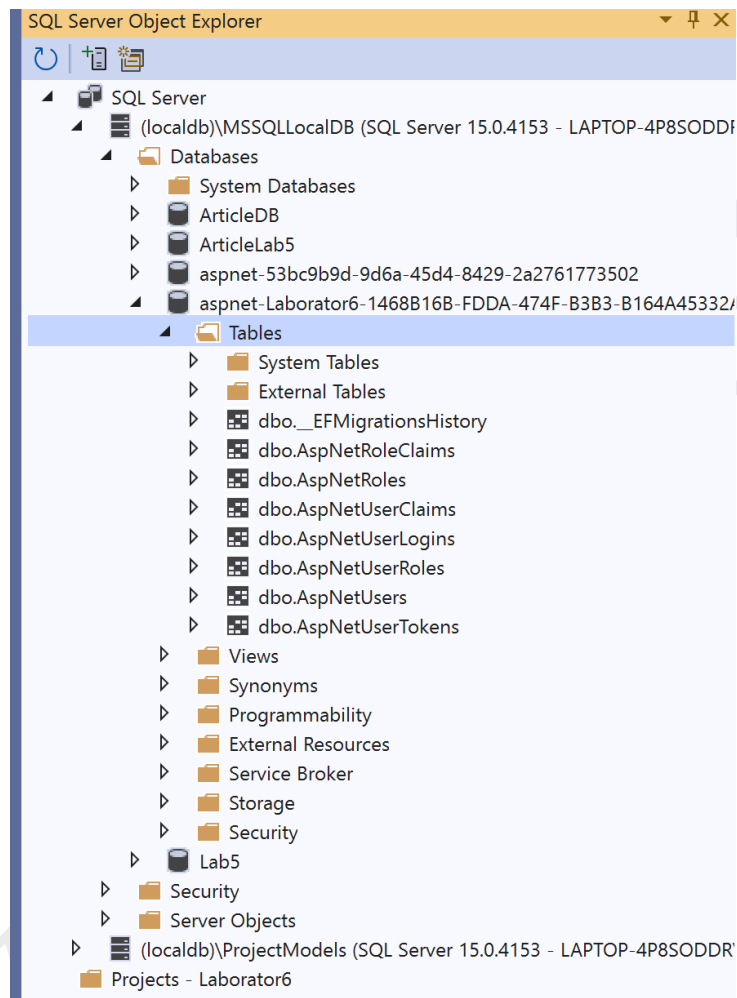
Proiectul nou creat contine **Identity Framework** si toate mecanismele aferente autentificarii.

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

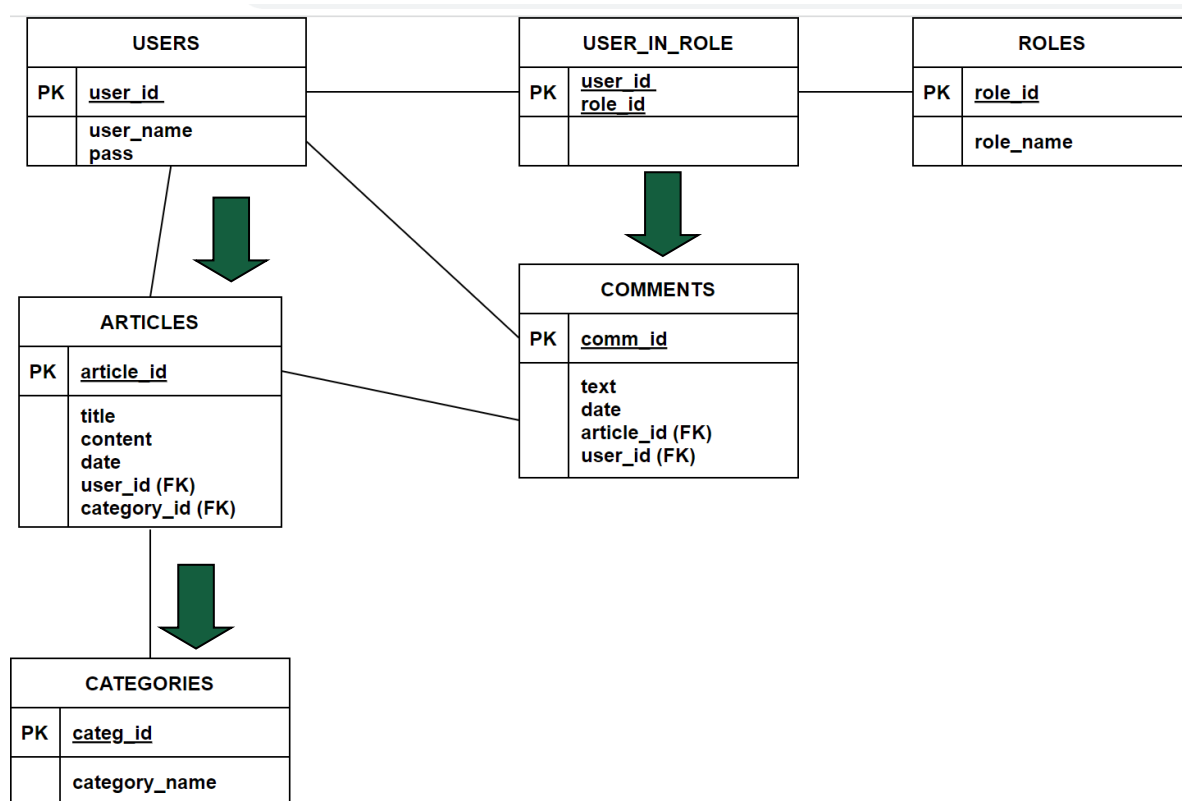
Dupa rularea comenzii, ne putem inregistra cu un cont.

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

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 entitatile **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)
- ArticleId (int – cheie externa – articolul caruia ii apartine 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 = "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; }
}

```

!/ 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": []
  }
]

```

Aceste proprietati sunt **virtuale** pentru a se executa **lazy loading**. Lazy loading este un design pattern folosit in dezvoltarea web care intarzie incarcarea anumitor 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]

```

```

[
  {
    "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 care 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 in 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": "Cate1",
      "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 relatiilor din baza de date

Relatia many-to-many – Varianta 1 – utilizand proprietatile

In cazul in care se doreste implementarea unei **relatii de tip many-to-many**, utilizand conventiile din framework, si anume utilizarea proprietatilor in cadrul claselor, se procedeaza ca in exemplul urmator. In acest caz 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 mai multi student

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; }**

Relatia many-to-many – Varianta 2

In acest caz, relatia many-to-many se implementeaza de la zero, prin suprascrierea conventiilor existente. Aceasta metoda ne ofera mai multe optiuni de configurare, avand acces facil la attributele modelelor. In exemplul urmator este utilizat Entity Framework Core – Fluent API.

Pentru implementarea claselor se procedeaza astfel:

Article.cs

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

    public string Title { get; set; }

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

Category.cs

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

    public string CategoryName { get; set; }

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

ArticleCategory.cs

```
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; }
}
```

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; }
    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);
    }
}

```

Relatia one-to-many – Varianta 1 – utilizand proprietatile

Exista si in acest caz doua variante de implementare. Cel mai des este utilizata varianta prezenta in sectiunea **Implementare cereri folosind Entity Framework Core si LINQ** din cursul curent, si anume cea in care se utilizeaza conventiile din cadrul framework-ului. Aceste conventii sunt proprietatile adaugate in cadrul fiecarei clase.

Relatia one-to-many – Varianta 2

De asemenea, daca varianta suprascrierii conventiilor este mai potrivita, atunci se procedeaza ca in exemplul urmator. Varianta potrivita se alege in functie de ceea ce dorim sa modelam.

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; }
}
```

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);
    }
}
```

Relatia one-to-one – Varianta 1 - utilizand proprietatile

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; }
}
```

Category.cs

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

    public string CategoryName { get; set; }

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


Relatia one-to-one – Varianta 2

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; }
}
```

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);
}
}
```