

UNIVERSITATEA DIN CRAIOVA
FACULTATEA DE ȘTIINȚE
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENȚĂ

Îndrumător științific:

Asist. Dr. Tudorache Cristina

Absolvent:

Stoentel Alexandru-Eduard

CRAIOVA

2024

**UNIVERSITATEA DIN CRAIOVA
FACULTATEA DE ȘTIINȚE
SPECIALIZAREA INFORMATICĂ**

**Platformă de blogging realizată cu tehnologii web și
baze de date**

Îndrumător științific:

Asist. Dr. Tudorache Cristina

Absolvent:

Stoentel Alexandru-Eduard

CRAIOVA

2024

Cuprins

1. INTRODUCERE	5
1.1 Scop principal	5
1.2 Motivație	5
2. TEHNOLOGII ȘI FRAMEWORK-URI FOLOSITE.....	7
2.1 C#.....	7
2.2 .NET.....	8
2.3 ASP.NET Web API	8
2.4 Entity Framework	9
2.5 Identity	13
2.6 LINQ.....	13
2.7 HyperText Markup Language (HTML).....	14
2.8 Cascading Style Sheets (CSS).....	15
2.9 Blazor.....	15
2.10 Blazorise	17
2.11 Bootstrap.....	17
2.12 Refit API.....	18
2.13 Structured query language (SQL)	18
2.14 NuGet Package Console	19
3. ELEMENTE SOFTWARE FOLOSITE	20
3.1 Visual Studio.....	20
3.2 SQL Server Management Studio 20	20
3.3 Postman	21
4. DESFĂȘURAREA APLICAȚIEI	22
5. SPECIFICAȚII ȘI REPREZENTAREA APLICAȚIEI.....	33
5.1 Specificații funcționale	33
5.2 Specificații tehnice.....	35
5.3 Diagramele cazurilor de utilizare	42
5.4 Organizarea bazei de date	43
6. DEZVOLTAREA APLICAȚIEI	44
6.1 Popularea bazei de date.....	44
6.2 Modele de baze de date Entity Framework.....	46
6.3 Sistemul de autentificare.....	51
6.4 Sistemul de blogging.....	54
6.5 Sistemul de comentarii.....	59
7. CONCLUZIE	60

8. BIBLIOGRAFIE.....	61
----------------------	----

1. INTRODUCERE

1.1 Scop principal

Această documentație reprezintă rezultatul eforturilor mele de cercetare pe parcursul a trei ani de studiu în domeniul Informaticii, la Facultatea de Științe din Craiova. Scopul principal al acestei documentații este să ofere informații detaliate despre arhitectura proiectului, tehnologiile utilizate și funcționalitățile de bază ale acestuia.

Proiectul este un sistem blogging, bazat pe scrierea unei postări, cu scopul de a oferi informații, divertisment și relaxare utilizatorilor. Platforma este concepută pentru a oferi utilizatorilor o experiență simplă și intuitivă de redactare a articolelor, facilitând procesul de creare și publicare a conținutului pe internet. Scopul principal al acestui sistem este să ofere o modalitate eficientă de comunicare și distribuire a informațiilor, promovând schimbul de idei și experiențe între utilizatori și contribuind la dezvoltarea unei comunități active și informate.

Pe lângă aspectele conceptuale, documentația urmărește să explice și motivul tehnic al începerii proiectului, precum și modul în care acesta poate aduce valoare comunității noastre. Consider că această aplicație va testa abilitățile mele în diverse domenii ale dezvoltării software, oferindu-mi motivația necesară pentru a găsi soluții eficiente și pentru a adopta noi tehnologii atunci când este necesar. Este o provocare care ne împinge întotdeauna să ne dăm tot ce e mai bun din noi și să ne depășim limitele.

1.2 Motivație

Studiile și analizele desfășurate pe parcursul timpului au demonstrat că platformele de blogging au un impact semnificativ în mediul online, fiind considerate surse de informații esențiale și instrumente de comunicare eficiente pentru utilizatori.

Unul dintre principalele avantaje ale unui sistem de blogging este accesibilitatea și ușurința în a distribui informații. Comparativ cu alte forme de comunicare și distribuire a conținutului, un blog oferă posibilitatea de a crea și partaja conținut într-un mod rapid și simplu, fără a fi nevoie de cunoștințe tehnice avansate.

De asemenea, blogging-ul oferă oportunitatea de a construi și de a consolida comunități online, aducând împreună persoane cu interese comune și creând un spațiu propice pentru schimbul de idei, opinii și experiențe. Astfel, platforma noastră de blogging este concepută

pentru a sprijini acest proces de conectare și colaborare între utilizatori, facilitând interacțiunea și schimbul de cunoștințe într-un mediu virtual prietenos și deschis.

În final, platforma noastră de blogging este menită să ofere o experiență plăcută și valoroasă utilizatorilor, oferindu-le un spațiu dedicat pentru exprimare și comunicare, încurajându-i să împărtășească idei, să dezbată subiecte importante și să contribuie la dezvoltarea unei comunități active și informate.

2. TEHNOLOGII ȘI FRAMEWORK-URI FOLOSITE

2.1. C#

C# este un limbaj de programare modern, orientat pe obiect, dezvoltat de Microsoft ca parte a platformei .NET. Este utilizat pe scară largă pentru dezvoltarea de aplicații enterprise, aplicații web, jocuri și multe altele. În contextul dezvoltării acestei aplicații de blogging, C# joacă un rol central în realizarea componentelor back-end și a logicii aplicației, oferind o serie de avantaje.

C# este un limbaj complet orientat pe obiect, ceea ce înseamnă că suportă concepte OOP precum clase, obiecte, moștenire, polimorfism și încapsulare. Acest lucru facilitează scrierea unui cod modular, reutilizabil și ușor de întreținut.

C# este un limbaj puternic tipizat, ceea ce înseamnă că fiecare variabilă trebuie declarată cu un tip specific. Acest lucru ajută la detectarea erorilor la timp de compilare, îmbunătățind astfel fiabilitatea și siguranța codului.

Programarea Orientată pe Obiecte (OOP) este un paradigmă de programare care utilizează "obiecte" și "clase" pentru a structura și organiza codul. Aceasta paradigmă este esențială pentru dezvoltarea de software scalabil și modular. OOP este fundamentată pe patru principii de bază: încapsularea, moștenirea, polimorfismul și abstractizarea.

Principiile de baza ale programării orientată pe obiecte sunt:

- Încapsularea: se referă la restricționarea accesului la anumite componente ale unui obiect și la protejarea stării interne a acestuia. Aceasta se realizează prin declararea variabilelor de clasă ca private și furnizarea de metode publice pentru accesarea și modificarea acestor variabile.
- Moștenirea: permite crearea de noi clase (clase derivate) care împrumută atribute și metode de la o clasă existentă (clasa de bază)
- Polimorfismul: permite utilizarea unei metode într-o clasă de bază pentru a fi redefinită sau suprascrisă într-o clasă derivată. Aceasta poate fi de două tipuri: polimorfism la timp de compilare (suprasarcină) și polimorfism la timp de execuție (suprascrivere).

- Abstractizarea: se referă la ascunderea detaliilor complexe de implementare și la expunerea doar a funcționalităților esențiale prin intermediul claselor abstrakte și al interfețelor

2.2. .NET

.NET este un framework software dezvoltat de Microsoft care oferă un mediu unificat pentru dezvoltarea și rularea aplicațiilor. Acesta suportă o gamă largă de aplicații, inclusiv aplicații desktop, web, mobile, cloud, jocuri și IoT (Internet of Things). .NET este cunoscut pentru interoperabilitatea sa, performanța ridicată și suportul puternic pentru limbajele de programare moderne.

.NET, este un framework versatil pentru dezvoltarea aplicațiilor, lansat inițial în 2002 și evoluat semnificativ odată cu introducerea .NET Core în 2016 și unificarea sub .NET 5 în 2020. Aceasta include CLR (Common Language Runtime) pentru execuția codului și BCL (Base Class Library) pentru funcționalități esențiale. Suportă limbaje precum C#, VB.NET, F# și C++/CLI, și permite dezvoltarea de aplicații desktop (Windows Forms, WPF), web (ASP.NET), mobile (Xamarin) și cloud. Avantajele sale includ portabilitatea multiplatformă, performanța ridicată, productivitatea sporită prin instrumente precum Visual Studio, un ecosistem bogat de pachete disponibile prin NuGet și securitatea integrată. .NET facilitează dezvoltarea de aplicații moderne și robuste, fiind o alegere populară în industrie.

2.3. ASP.NET Web API

ASP.NET Web API¹ este un framework² pentru construirea și consumul de servicii HTTP care pot fi accesate de la diverse clienți, inclusiv browsere web, aplicații mobile și aplicații desktop. ASP.NET Web API este o parte a platformei .NET și oferă un set robust de instrumente și funcționalități pentru crearea API-urilor web RESTful. REST este un stil de arhitectură pentru a dezvolta servicii web, care utilizează protocolul HTTP ca interfață de comunicare pentru a transfera date prin metode HTTP.

¹ Application Programming Interface (API) reprezintă un set de definiții de sub-programe, protocoale și unele pentru programarea de aplicații și software.

² În dezvoltarea de software un framework este o structură conceptuală și reprezintă o arhitectură de software care modelează relațiile generale ale entităților domeniului (site-ului).

ASP.NET Web API este proiectat pentru a construi servicii RESTful, care se bazează pe principiile de transfer de stare reprezentativ (REST). Aceste servicii sunt caracterizate prin utilizarea metodelor HTTP standard (GET, POST, PUT, DELETE) pentru operațiuni CRUD (Create, Read, Update, Delete).

Framework-ul oferă mecanisme puternice pentru legarea datelor din cererile HTTP la modele de date și pentru validarea acestor date. Decorarea modelelor cu atrbute de validare asigură că datele primite respectă regulile specificate înainte de a fi procesate.

ASP.NET Web API suportă diverse metode de autentificare și autorizare, inclusiv utilizarea de token-uri JWT (JSON Web Token), Identity, OAuth și alte mecanisme de securitate.

2.4. Entity Framework

Entity Framework (EF) este un Object-Relational Mapper (ORM) open-source pentru .NET, care simplifică accesul la baze de date și manipularea datelor prin maparea obiectelor din aplicație la tabelele din baza de date³. EF permite dezvoltatorilor să interacționeze cu baza de date folosind obiecte .NET, eliminând necesitatea de a scrie cod SQL manual pentru majoritatea operațiunilor de date.

EF facilitează maparea obiectelor din modelul de date (entități) la tabelele din baza de date. Aceasta permite utilizarea limbajului de programare C# pentru a efectua operațiuni de date.

Entity Framework permite utilizarea Language Integrated Query (LINQ) pentru interogarea bazei de date. LINQ oferă o sintaxă intuitivă și puternică pentru scrierea interogărilor.

EF suportă trei metode principale de dezvoltare:

- Model First: Dezvoltarea începe cu un model conceptual care este utilizat pentru a genera baza de date.
- Database First: Baza de date existentă este utilizată pentru a genera modelul de date.
- Code First: Modelul de date este definit folosind clase C#, iar baza de date este generată din acest model.

³ O bază de date este o colecție organizată de date care sunt stocate și accesate electronic. Bazele de date sunt esențiale pentru gestionarea și stocarea datelor în diverse aplicații și sisteme informatici, facilitând accesul rapid și eficient la informații.

Migrația este o caracteristică care permite evoluția schemei bazei de date în timp. Dezvoltatorii pot adăuga, modifica sau elimina tabele și coloane, iar EF va genera scripturi SQL necesare pentru aplicarea acestor modificări.

Entity Framework facilitează crearea relațiilor dintre entități, astfel avem relații: one-to-one, one-to-many și many-to-many.

O relație *one-to-one* apare atunci când fiecare rând dintr-o tabelă este asociat cu un singur rând dintr-o altă tabelă. De exemplu, un utilizator poate avea un profil unic asociat. În C# acest lucru se poate face fie prin proprietățile de navigare, fie prin referențierea virtuală dintre cele două entități.

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    public virtual StudentAddress Address { get; set; }
}

public class StudentAddress
{
    public int StudentAddressId { get; set; }
    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public int Zipcode { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

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

Figură 2.1. One-to-one folosind entitățile

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    public virtual StudentAddress Address { get; set; }
}

public class StudentAddress
{
    [ForeignKey("Student")]
    public int StudentAddressId { get; set; }

    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public int Zipcode { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

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

Figură 2.2. One-to-one folosind entitățile și specificând ForeignKey-ul

```

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    // Configure Student & StudentAddress entity
    modelBuilder.Entity<Student>()
        .HasOptional(s => s.Address) // Mark Address property optional in Student entity
        .WithRequired(ad => ad.Student); // mark Student property as required in StudentAddress entity. Cannot save Student without address
}

```

Figură 2.3. One-to-one folosind Fluent API

O relație *one-to-many* apare atunci când un rând dintr-o tabelă poate fi asociat cu mai multe rânduri dintr-o altă tabelă. De exemplu, un user poate avea mai multe blog-uri.

```

public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeID { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

    public ICollection<Student> Student { get; set; }
}

```

Figură 2.4. One-to-many folosind entitățile

```

public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public int GradeId { get; set; }
    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }

    public ICollection<Student> Student { get; set; }
}

```

Figură 2.5. One-to-many folosind entitățile și specificând ForeignKey-ul

```
modelBuilder.Entity<Grade>()
    .HasMany<Student>(g => g.Students)
    .WithRequired(s => s.CurrentGrade)
    .HasForeignKey<int>(s => s.CurrentGradeId);
```

Figură 2.6. one-to-one folosind Fluent API

O relație *many-to-many* apare atunci când mai multe rânduri dintr-o tabelă pot fi asociate cu mai multe rânduri dintr-o altă tabelă. De exemplu, userii pot avea mai multe roluri, iar aceleși roluri pot fi atribuite mai multor useri.

```
public class Student
{
    public Student()
    {
        this.Courses = new HashSet<Course>();
    }

    public int StudentId { get; set; }
    [Required]
    public string StudentName { get; set; }

    public virtual ICollection<Course> Courses { get; set; }
}

public class Course
{
    public Course()
    {
        this.Students = new HashSet<Student>();
    }

    public int CourseId { get; set; }
    public string CourseName { get; set; }

    public virtual ICollection<Student> Students { get; set; }
}
```

Figură 2.7. one-to-many folosind entitățile

```
modelBuilder.Entity<Student>()
    .HasMany<Course>(s => s.Courses)
    .WithMany(c => c.Students)
    .Map(cs =>
    {
        cs.MapLeftKey("StudentRefId");
        cs.MapRightKey("CourseRefId");
        cs.ToTable("StudentCourse");
    });

```

Figură 2.8. one-to-one folosind Fluent API

2.5. Identity

ASP.NET Core Identity este un sistem de autentificare și autorizare care permite dezvoltatorilor să gestioneze utilizatorii, parolele, rolurile și permisiunile într-o aplicație web. Identity este proiectat pentru a fi extensibil și personalizabil, oferind în același timp o securitate robustă și funcționalități avansate pentru gestionarea identităților utilizatorilor.

Identity oferă mecanisme pentru autentificarea utilizatorilor în aplicație și autorizarea acestora pentru accesarea resurselor protejate. Aceste funcționalități sunt integrate cu middleware-ul de autentificare al ASP.NET Core.

Identity gestionează detaliile utilizatorilor, inclusiv numele de utilizator, parolele și alte informații relevante. Parolele sunt stocate în mod securizat folosind hashing și salting.

Identity pune la dispoziție un set prestabilit de endpoint-uri, disponibile pentru Web API. El vine cu un service-uri și repository-uri deja predefinite pentru a administra utilizatorii și rolurile:

```
16  private readonly SignInManager<DataBaseLayout.Models.User> _signinManager;
17  private readonly UserManager<DataBaseLayout.Models.User> _userManager;
```

Figură 2.9. Identity helpers pentru a administra userii și login-ul

2.6. LINQ

Language Integrated Query (LINQ) este o componentă puternică a limbajului C# și a platformei .NET, care oferă o sintaxă coerentă și ușor de utilizat pentru interogarea și manipularea colecțiilor de date. LINQ permite dezvoltatorilor să scrie interogări puternice și eficiente direct în codul lor C#, fie că lucrează cu baze de date, colecții în memorie, XML, sau alte surse de date.

LINQ oferă o sintaxă unificată pentru interogarea diferitelor surse de date. Aceasta face ca interogările să fie mai intuitive și mai ușor de citit, indiferent de tipul de date cu care se lucrează.

LINQ este integrat direct în limbajul C#, ceea ce înseamnă că dezvoltatorii pot folosi operatori familiari și expresii lambda pentru a construi interogări.

LINQ oferă un set bogat de operatori standard pentru filtrare, proiecție, grupare, unire și alte operații comune. Acești operatori includ *Where*, *Select*, *GroupBy*, *Join*, *OrderBy*, *Sum*, *Average*, *Count*, și altele.

2.7. HyperText Markup Language (HTML)

HyperText Markup Language (HTML) este limbajul standard folosit pentru a crea și structura paginile web⁴. HTML folosește un set de elemente (denumite și tag-uri) pentru a defini diferitele părți ale unei pagini web, cum ar fi paragrafele, imaginile, linkurile și multe altele. Aceste elemente sunt înconjurate de acolade unghiulare (< >) și sunt în general pereche, cu un tag de deschidere și unul de închidere.

Cele mai importante tag-uri care se regăsesc și în proiectul meu sunt:

- <html> - elementul rădăcină care înconjoară întregul document HTML.
- <head> - conține meta-informării despre document, cum ar fi titlul paginii, linkurile către foi de stil și scripturi.
- <title> - definește titlul paginii, afișat în bara de titlu a browserului sau pe fila paginii.
- <body> - conține conținutul vizibil al paginii web, cum ar fi textul, imaginile, formularele și alte elemente.
- <p> - definirea paragrafelor.
- - includerea imaginilor. Atributul src specifică calea imaginii, iar alt oferă o descriere a acesteia.
- <div> și - elemente generice pentru blocuri și inline, utilizate pentru a grupa alte elemente și a aplica stiluri CSS.

```

4 4<!DOCTYPE html>
5 5<html lang="en">
6 6<head>
7 7    <meta charset="utf-8" />
8 8    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
9 9    <base href="/" />
10 10   <link href="~/_content/Blazorise(Icons.FontAwesome/v6/css/all.min.css" rel="stylesheet">
11 11   <link href="~/_content/Blazorise/blazorise.css" rel="stylesheet" />
12 12   <link href="~/_content/Blazorise.Bootstrap/bootstrap.css" rel="stylesheet" />
13 13   <link href="~/_content/Blazorise.Snackbar/blazorise.snackbar.css" rel="stylesheet" />
14 14   <link href="~/_content/Blazorise.SpinKit/blazorise.spinkit.css" rel="stylesheet" />
15 15   <link href="~/_content/Blazorise.LoadingIndicator/blazorise.loadingindicator.css" rel="stylesheet" />
16 16
17 17

```

Figură 2.10. Exemplu de tag-uri html

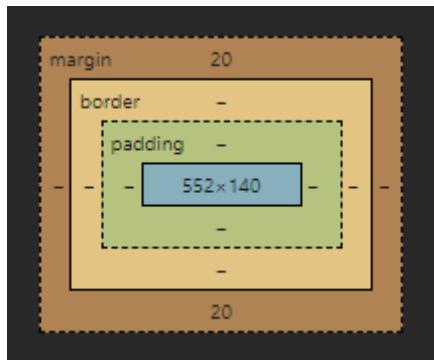
⁴ World Wide Web (pe scurt web sau www) este un sistem hipertext care operează pe Internet. Hipertextul este vizualizat cu un program numit browser, care descarcă paginile web de pe un server web (sau site web) și îl afișează pe ecran.

2.8. Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) este un limbaj de stil folosit pentru a descrie prezentarea unui document scris într-un limbaj de marcăre, cum ar fi HTML. CSS controlează aspectul și formatarea elementelor HTML pe o pagină web, inclusiv layout-ul, culorile, fonturile și alte aspecte vizuale. Prin separarea conținutului de prezentare, CSS permite o gestionare mai ușoară și flexibilă a designului web.

Layout-ul unui tag HTML la care se aplică CSS este modul în care acesta este poziționat și aranjat în cadrul DOM-ului. Aceasta este determinat de regulile CSS aplicate tag-ului, care pot controla dimensiunea, poziția, spațierea și alte aspecte ale elementului.

Layout-ul este format din: margin, border, padding și content:



Figură 2.11. Layout-ul unui element afișat în DOM

2.9. Blazor

Blazor este un framework open-source dezvoltat de Microsoft pentru crearea de aplicații web interactive, single-page applications (SPA), utilizând .NET și C#. Blazor permite dezvoltatorilor să construiască interfețe de utilizator web moderne și interactive fără a folosi JavaScript pentru logica de client. Blazor folosește C# atât pe server, cât și pe client, împreună cu componente reutilizabile, care sunt asemănătoare cu componentele din alte framework-uri de frontend, cum ar fi React sau Angular.

Blazor permite dezvoltarea de aplicații web folosind C# și .NET, atât pe partea de client, cât și pe cea de server. Aceasta oferă avantajul de a utiliza un singur limbaj de programare pentru întregul stack al aplicației.

Blazor folosește un model bazat pe componente, unde fiecare componentă este o unitate reutilizabilă de interfață de utilizator, ce poate include atât markup HTML, cât și logică de interacțiune în C#.

Blazor oferă două modele principale de hosting:

- Blazor Server: Execută logica aplicației pe server și folosește SignalR⁵ pentru a comunica actualizările UI cu browserul.
- Blazor WebAssembly: Execută aplicația complet în browser folosind WebAssembly. Aceasta permite aplicațiilor Blazor să ruleze direct pe client fără a depinde de server pentru logica de interacțiune.

Blazor suportă binding bidirectional al datelor, permisând sincronizarea automată a datelor între componenta UI și modelul de date. În Blazor, o componentă este reprezentată printr-o clasă C# care extinde clasa *ComponentBase*. Această clasă reprezintă o unitate autonomă și reutilizabilă de interfață utilizator, care poate fi utilizată în cadrul aplicației Blazor pentru a afișa conținut și pentru a interacționa cu utilizatorul.

O componentă Blazor este definită într-un fișier cu extensia *.razor*⁶, care conține atât cod C# cât și markup HTML. Acest fișier combină logica și prezentarea componentei într-un singur loc. În interiorul fișierului *.razor*, este definită o clasă C# care extinde clasa *ComponentBase*. Această clasă conține logica componentei, inclusiv metodele de inițializare, manipulare a evenimentelor și alte funcționalități specifice. În fișierul *.razor*, este inclus și markup-ul HTML care reprezintă aspectul vizual al componentei. Acest markup poate conține elemente HTML standard, împreună cu directivelor specifice Blazor pentru atașarea de evenimente, legături de date și alte funcționalități.

Proprietăți și Parametri: Componentele Blazor pot avea proprietăți și parametri, care sunt utilizate pentru a transmite date către componentă și pentru a personaliza comportamentul acesteia. Acestea pot fi definite ca proprietăți ale clasei componentei și pot fi utilizate în interiorul markup-ului HTML pentru a influența afișarea și funcționarea componentei.

```
[Parameter]
public EventCallback<Guid> ItemClicked { get; set; }
```

⁵ SignalR este o bibliotecă inclusă în ASP.NET și ASP.NET Core care facilitează adăugarea de funcționalități de comunicație în timp real în aplicațiile web. Utilizând SignalR, dezvoltatorii pot crea aplicații care necesită o comunicare bidirectională între server și client, cum ar fi chaturile live, tablourile de bord de monitorizare, jocurile online și alte aplicații interactive.

⁶ Razor este un motor de vizualizare folosit în cadrul ASP.NET pentru a genera pagini web dinamice. Este utilizat în principal în ASP.NET Core și ASP.NET MVC pentru a crea interfețe web prin combinarea codului C# cu markup-ul HTML.

Figură 2.12. Parametru în Blazor



Figură 2.13. Exemplu de utilizare a unui parametru în Blazor

2.10. Blazorise

Blazorise este o bibliotecă de componente UI⁷ pentru Blazor, care oferă un set de componente stilizate și interactive predefinite pentru dezvoltarea rapidă a aplicațiilor web Blazor. Blazorise simplifică crearea de interfețe de utilizator atractive și funcționale, eliminând nevoie de a scrie cod CSS și JavaScript personalizat pentru fiecare componentă.

Blazorise oferă o gamă largă de componente UI, cum ar fi butoane, casete de text, dropdown-uri, liste, paginatoare, meniuri și multe altele, toate stilizate și gata de utilizare.

Fiecare componentă Blazorise vine cu funcționalitate încorporată, cum ar fi gestionarea *evenimentelor de click, hover și focus, validarea datelor de intrare, gestionarea paginării și sortării*, facilitând dezvoltarea aplicațiilor web interactive.

Biblioteca Blazorise permite crearea de teme personalizate sau utilizarea unor teme predefinite, precum *Bootstrap* sau *Material Design*, oferind o experiență coerentă și stilizată pentru aplicațiile Blazor.

2.11. Bootstrap

Bootstrap este unul dintre cele mai populare framework-uri front-end utilizate pentru dezvoltarea rapidă a interfețelor de utilizator web. Creat inițial de către Twitter, Bootstrap este acum o resursă open-source și este utilizat pe scară largă de dezvoltatori din întreaga lume pentru construirea de site-uri web responsive și atrăgătoare.

Bootstrap oferă un sistem de grilă flexibil, bazat pe un sistem de coloane și rânduri, care facilitează crearea de layout-uri responsive și adaptabile pentru diverse dimensiuni de ecran.

⁷ UI este prescurtarea de la user interface (interfață de utilizator). Este aspectul grafic al unei aplicații sau al unui website și se referă la toate elementele care vin în contact cu utilizatorii: butoane, text, imagini, câmpuri pentru introducere text, slide-uri și aşa mai departe.

Bootstrap include o serie de componente UI predefinite, cum ar fi butoane, casete de text, dropdown-uri, bare de navigare, carduri, alerte și multe altele, care sunt stilizate și gata de utilizare.

Bootstrap oferă o gamă largă de clase utilitare CSS pentru gestionarea marginilor, padding-urilor, alinierii, culorilor și multe altele, care facilitează stilizarea și formatarea rapidă a elementelor HTML.

Bootstrap permite personalizarea aspectului și stilului componentelor prin intermediul temelor predefinite sau prin crearea de teme personalizate, permitând dezvoltatorilor să creeze designuri unice și distinctive.

Bootstrap este compatibil cu cele mai recente versiuni ale principalelor browsere web și asigură o experiență consistentă pentru utilizatorii din întreaga lume.

2.12. Refit API

Refit este o bibliotecă .NET care simplifică integrarea API-urilor REST în aplicațiile .NET. Creată de Paul Betts, Refit oferă o modalitate elegantă și ușor de utilizat pentru definirea și apelarea serviciilor web bazate pe HTTP, fără a fi nevoie să se scrie cod boilerplate pentru comunicare sau serializare/deserializare a datelor JSON⁸.

Refit permite definirea interfețelor de serviciu în stilul .NET, care declară metode pentru fiecare endpoint al API-ului. Aceste interfețe servesc ca contracte pentru comunicarea cu serviciul web.

Folosind anotările de atribut, dezvoltatorii pot specifica detalii precum URL⁹-ul, metoda HTTP și alte opțiuni pentru fiecare metodă din interfața de serviciu.

Refit automat gestionează serializarea și deserializarea datelor JSON, fără a fi nevoie să se scrie cod suplimentar pentru aceste operațiuni.

Refit suportă toate metodele HTTP standard, inclusiv GET, POST, PUT, DELETE etc., facilitând comunicarea cu API-uri RESTful.

2.13. Structured query language (SQL)

Structured Query Language (SQL) este un limbaj de programare specializat utilizat pentru gestionarea datelor în sisteme de management al bazelor de date relaționale (RDBMS).

⁸ JSON este un format text, inteligibil pentru oameni, utilizat pentru reprezentarea obiectelor și a altor structuri de date și este folosit în special pentru a transmite date structurate prin rețea, procesul purtând numele de serializare

⁹ URL - localizator uniform de resurse este o secvență de caractere standardizată, folosită pentru denumirea, localizarea și identificarea unor resurse de pe Internet, inclusiv documente text, imagini, clipuri video, expuneri de diapositive etc.

SQL permite utilizatorilor să efectueze diverse operațiuni asupra datelor, inclusiv interogări, inserări, actualizări și ștergeri.

În cadrul unei baze de date relaționale, relațiile dintre tabele sunt definite prin intermediul cheilor străine și primare. Iată câteva tipuri de relații comune:

- *One-to-One*: Acest tip de relație apare atunci când fiecare înregistrare dintr-o tabelă este asociată cu exact o înregistrare dintr-o altă tabelă și invers.
- *One-to-Many*: Într-o relație unu-la-mulți, fiecare înregistrare dintr-o tabelă este asociată cu mai multe înregistrări dintr-o altă tabelă.
- *Many-to-Many*: Într-o relație mulți-la-mulți, mai multe înregistrări dintr-o tabelă sunt asociate cu mai multe înregistrări dintr-o altă tabelă. Pentru a implementa această relație, este necesară o tabelă de legătură care să asocieze înregistrările din cele două tabele.

Foreign Keys și Primary Keys sunt concepte fundamentale în proiectarea bazelor de date relaționale și sunt utilizate pentru a stabili relațiile între tabele.

Primary Key este o coloană sau un set de coloane într-o tabelă care identifică unic fiecare înregistrare din acea tabelă. Cheia primară este folosită pentru a asigura integritatea datelor și pentru a permite accesul rapid și eficient la înregistrările din tabelă. O cheie primară trebuie să fie unică pentru fiecare înregistrare din tabelă.

Foreign Key este o coloană sau un set de coloane într-o tabelă care stabileste o legătură între două tabele. Foreign Key este folosită pentru a crea relații între tabele, permitând asocierea datelor dintr-o tabelă cu înregistrările corespunzătoare dintr-o altă tabelă. O Foreign Key este de obicei legată de Primary Key a unei alte tabele, stabilind astfel relația dintre ele.

2.14. NuGet Package Console

NuGet Package Console este o interfață de linie de comandă (CLI) integrată în Visual Studio care permite dezvoltatorilor să instaleze, să actualizeze și să gestioneze pachetele NuGet în proiectele lor. Este o unealtă utilă pentru lucrul cu dependințele și bibliotecile externe în aplicațiile lor .NET.

3. ELEMENTE SOFTWARE FOLOSITE

3.1. Visual Studio

Visual Studio este un mediu integrat de dezvoltare (IDE) dezvoltat de Microsoft, utilizat pentru dezvoltarea diverselor tipuri de aplicații, de la aplicații desktop și web la aplicații mobile și jocuri. Este una dintre cele mai populare și puternice suite de dezvoltare pentru platforma .NET și alte tehnologii.

Visual Studio oferă o interfață utilizator prietenoasă, cu un set bogat de instrumente și opțiuni de personalizare, permitând dezvoltatorilor să lucreze eficient și confortabil.

Visual Studio oferă un puternic set de instrumente pentru debugging¹⁰ și profilare a aplicațiilor, inclusiv suport pentru breakpoint-uri¹¹, evaluarea expresiilor, analiza performanței și depanarea la distanță.

IDE-ul are integrată suportul pentru controlul versiunilor cu Git, permitând dezvoltatorilor să gestioneze și să colaboreze la codul lor folosind funcționalități precum controlul versiunilor, ramificările și solicitările de tragere.

Visual Studio suportă o varietate de limbaje de programare, inclusiv C#, VB.NET, F#, C++, Python, JavaScript, TypeScript și altele, oferind dezvoltatorilor o platformă unificată pentru dezvoltarea diferitelor tipuri de aplicații.

Visual Studio oferă un set complet de instrumente pentru dezvoltarea aplicațiilor web și mobile, inclusiv şabloane de proiect, instrumente de testare, emulatoare și suport pentru tehnologii precum ASP.NET, Blazor, Angular, React, Xamarin și multe altele.

3.2. SQL Server Management Studio 20

SQL Server Management Studio (SSMS) este o aplicație de gestionare și dezvoltare a bazelor de date, dezvoltată de Microsoft, care servește ca interfață grafică pentru administrarea și manipularea bazelor de date Microsoft SQL Server. Este una dintre cele mai utilizate și puternice aplicații pentru administrarea bazelor de date SQL Server.

¹⁰ Debugging, în inginerie, este procesul de găsire a cauzei rădăcină și a soluțiilor și a posibilelor remedieri pentru erori.

¹¹ Breakpoint-ul în dezvoltarea de software, este un loc de oprire sau pauză intenționată într-un program, pus în aplicare în scopuri de depanare. De asemenea, uneori este denumită pur și simplu o pauză.

SSMS oferă o interfață utilizator familiară și ușor de utilizat, cu un set bogat de instrumente și opțiuni pentru administrarea bazelor de date.

Utilizatorii pot crea, edita și executa interogări SQL direct în SSMS, beneficiind de facilități precum colorarea sintaxei, completarea automată a codului și sugestii de comenzi. SSMS permite gestionarea obiectelor bazei de date, cum ar fi tabelele, vizualizările, funcțiile, procedurile stocate, indecșii și altele, prin intermediul unei interfețe grafice intuitive.

SSMS oferă capacitați puternice de scripting și automatizare, permitând utilizatorilor să scrie și să execute scripturi SQL complexe și să automatizeze sarcini comune de administrare a bazei de date.

3.3. Postman

Postman este o platformă pentru dezvoltatorii de API-uri care permite testarea, dezvoltarea și documentarea API-urilor. Este o aplicație puternică și ușor de utilizat, disponibilă ca aplicație desktop și extensie de browser, care oferă un set complet de instrumente pentru gestionarea și testarea API-urilor.

Postman oferă o interfață utilizator prietenoasă, cu un design intuitiv și o navigare simplă, permitând utilizatorilor să lucreze eficient și să acceseze rapid funcționalitățile aplicației.

Utilizatorii pot crea și trimite cereri HTTP personalizate, inclusiv cereri GET¹², POST¹³, PUT¹⁴, DELETE¹⁵ etc., folosind un editor de cereri ușor de utilizat, cu suport pentru parametri, antete, corpuri de cerere și multe altele. Postman permite crearea și gestionarea mediilor de testare, permitând dezvoltatorilor să organizeze și să execute seturi de teste pentru a valida și a verifica funcționalitatea API-urilor lor.

Postman oferă funcționalități avansate de colaborare și partajare, permitând dezvoltatorilor să lucreze împreună la dezvoltarea și testarea API-urilor și să partajeze rapid și ușor cererile și seturile de teste cu alte echipe.

¹² Metoda GET este folosită pentru a prelua date de la o adresă URL

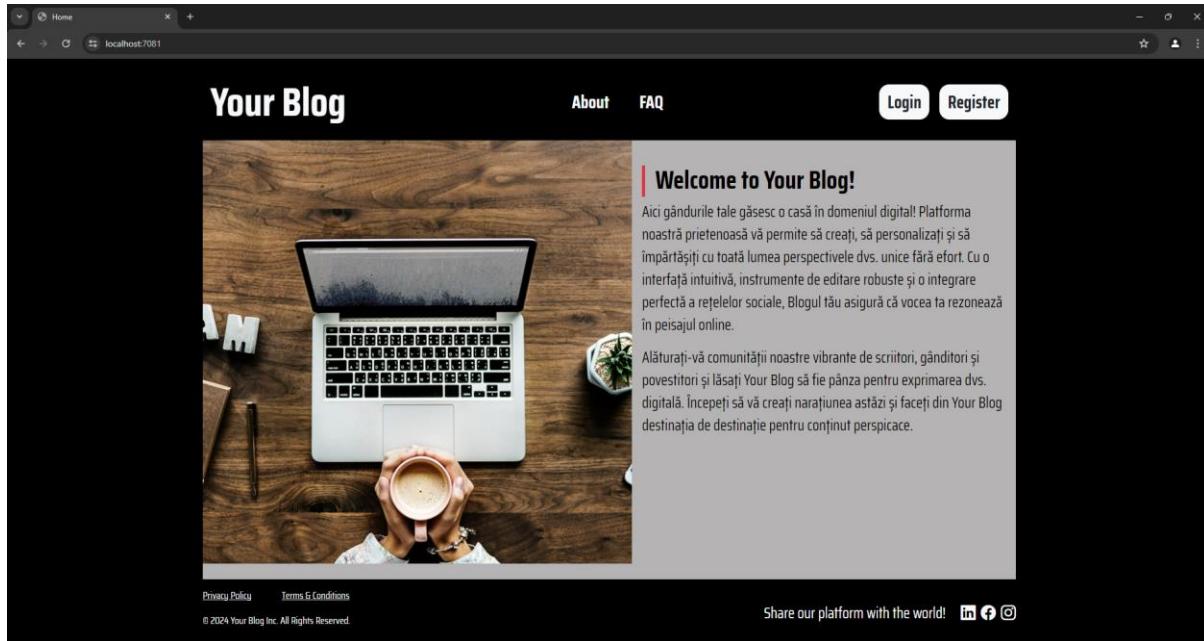
¹³ Metoda HTTP POST trimite date către server.

¹⁴ Metoda PUT creează o nouă resursă sau înlocuiește o reprezentare a resursei țintă cu sarcina utilă de solicitare.

¹⁵ Metoda HTTP DELETE șterge resursa specificată.

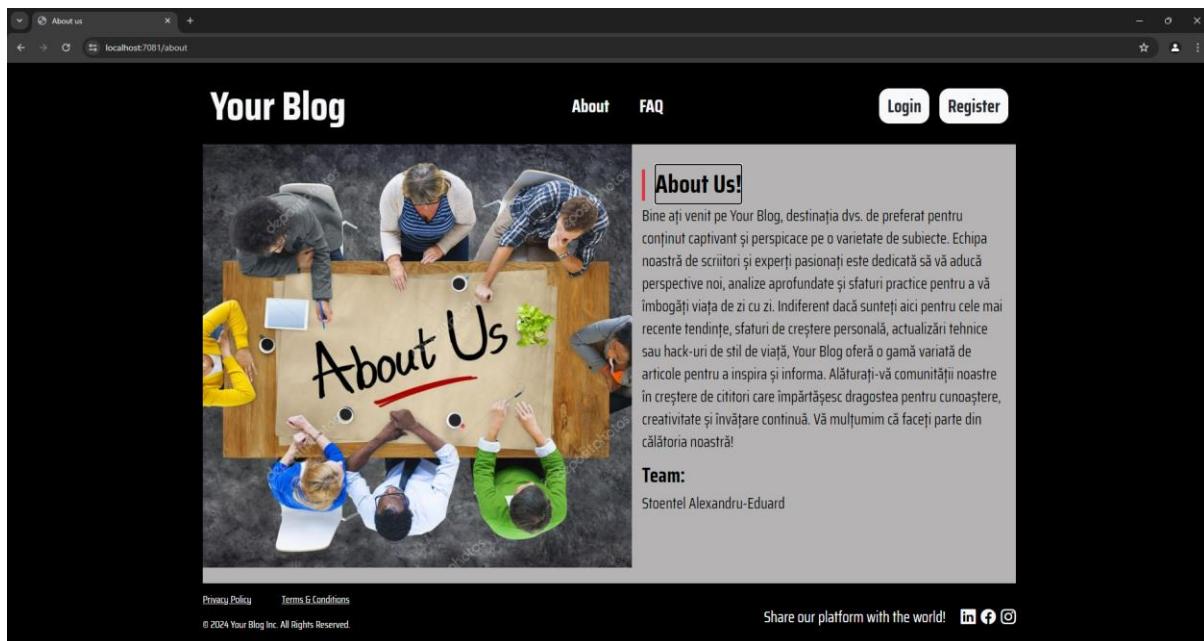
4. DESFĂȘURAREA APLICAȚIEI

În primă fază, când utilizatorul intră pe platformă, este redirectionat pe pagina principală unde găsește un scurt rezumat al proiectului:



Figură 4.1. Homepage când user-ul este delegat

El poate vedea mult mai multe informații la secțiunea *About*:



Figură 4.2. Pagina about us

Sau poate vedea întrebările și răspunsurile adminilor, la secțiunea *FAQ*¹⁶:

The screenshot shows a web browser window with the URL 'localhost:7081/faq'. The page has a dark header with the title 'Your Blog' and navigation links for 'About', 'FAQ', 'Login', and 'Register'. Below the header is a section titled 'Frequently Asked Questions' containing the following questions:

- 1. Ce este este Your Blog?**
Your Blog este o platformă de blogging care oferă conținut captivant și perspicace pe o varietate de subiecte, inclusiv cele mai recente tendințe, sfaturi de dezvoltare personală, actualizări tehnice și hack-uri privind stilul de viață.
- 2. Cum îmi pot crea un cont?**
Pentru a crea un cont, faceți clic pe butonul „Register” din colțul din dreapta sus al paginii de pornire. Completați informațiile necesare, inclusiv numele dvs., adresa de e-mail și o parolă.
- 3. Cum pot scrie o postare?**
Pentru scriere o postare, faceți clic pe butonul „Write” și completați toate informațiile.
- 4. Cum raportează conținut neadecvat?**
Dacă întâlniți orice conținut despre care credeți că încalcă termenii și condițiile noastre sau este inadecvat, vă rugăm să ni-l raportați făcând trimijându-ne un e-mail la your-blog@blog.ro cu detaliile problemei.
- 5. Pot face publicitate pe Your Blog?**
Da, oferim diverse oportunități de publicitate pe Your Blog. Pentru mai multe informații despre opțiunile noastre de publicitate, vă rugăm să contactați echipa noastră de publicitate la your-blog@blog.ro.
- 6. Cum îmi pot actualiza informațiile contului?**
Pentru a vă actualiza informațiile contului, conectați-vă la contul dvs. și accesați pagina „Profile”. Aici puteți să vă actualizați informațiile personale, să vă schimbați parola și să vă actualizați imaginea de profil.

Figură 4.3. Pagina FAQ

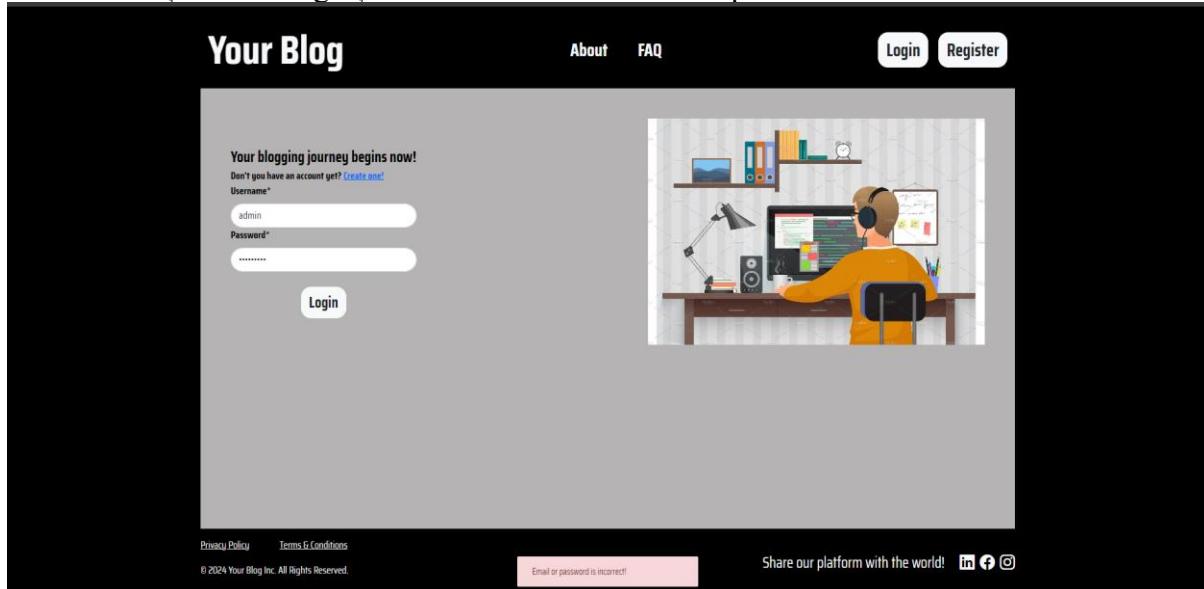
Pentru a se autentifica poate naviga către pagina *Login*:

The screenshot shows a web browser window with the URL 'localhost:7081/login'. The page has a dark header with the title 'Your Blog' and navigation links for 'About', 'FAQ', 'Login', and 'Register'. The main content area features a message 'Your blogging journey begins now!' followed by a 'Create one!' link. It includes fields for 'Username*' and 'Password*', both with placeholder text 'Enter your username here...' and 'Enter your password here...'. A large 'Login' button is positioned below the fields. To the right of the form is a cartoon illustration of a person wearing headphones and working at a desk with a computer monitor, keyboard, and various office supplies. At the bottom of the page are links for 'Privacy Policy' and 'Terms & Conditions', a copyright notice '© 2024 Your Blog Inc. All Rights Reserved.', and social sharing icons for LinkedIn, Facebook, and Instagram.

Figură 4.4. Pagina Login

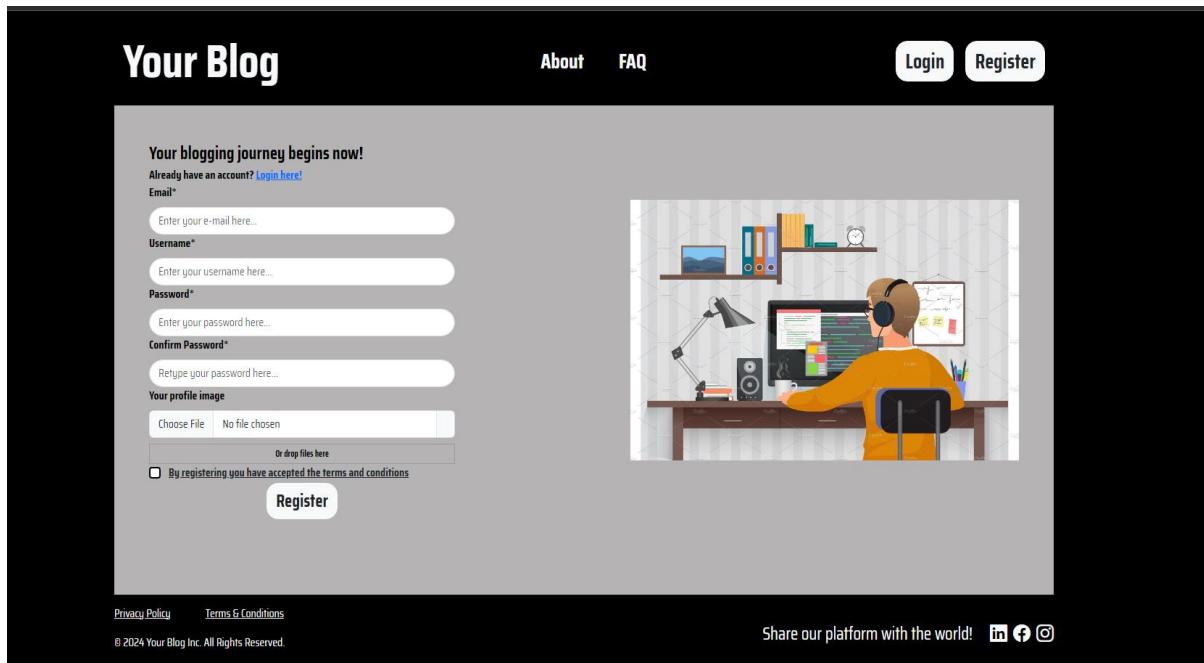
¹⁶ FAQ - Frequently Asked Questions

Dacă credențialele sunt gresite utilizatorul este avertizat printr-un toast:

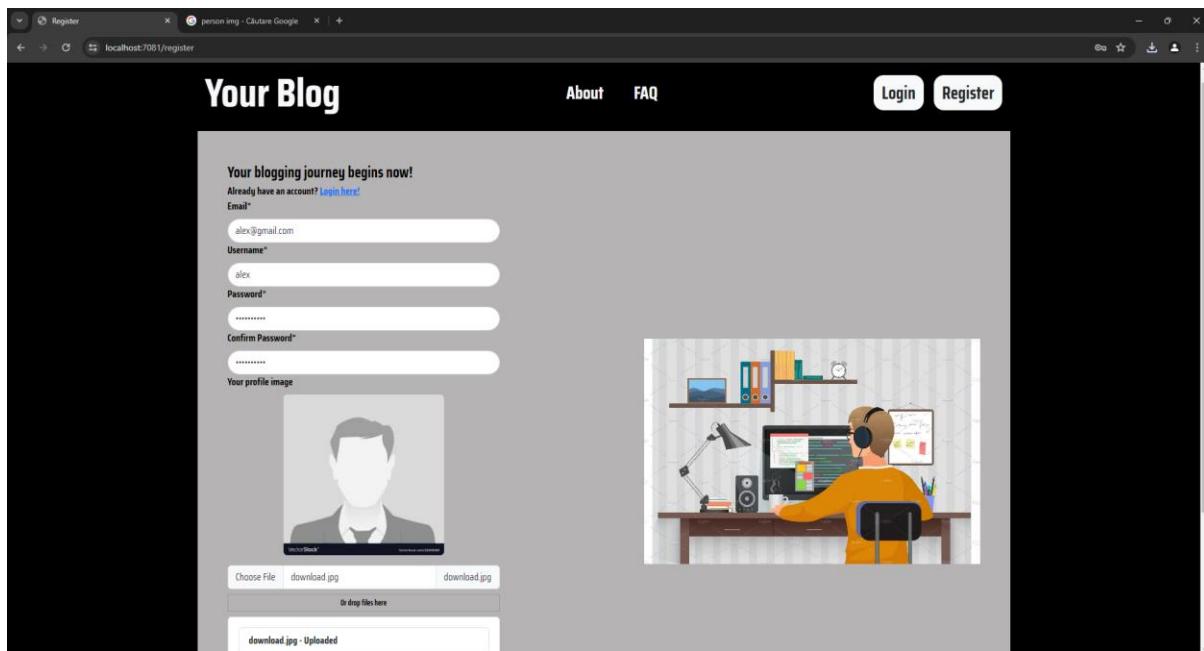


Figură 4.5. Pagina Login cu credențiale greșite

Dacă utilizatorul dorește să își creeze un cont, trebuie să navigheze către *Register* și că completeze informațiile de acolo. Imaginea de profil este opțională, caz în care este completată cu una predefinată, dacă este omisă.

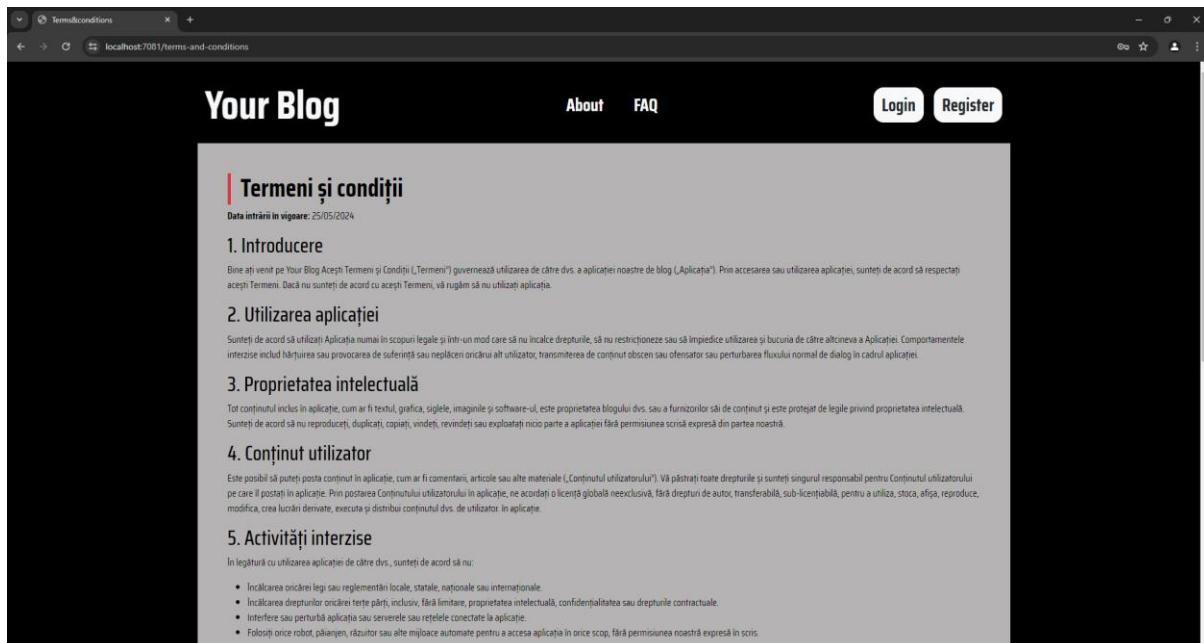


Figură 4.5. Pagina Register

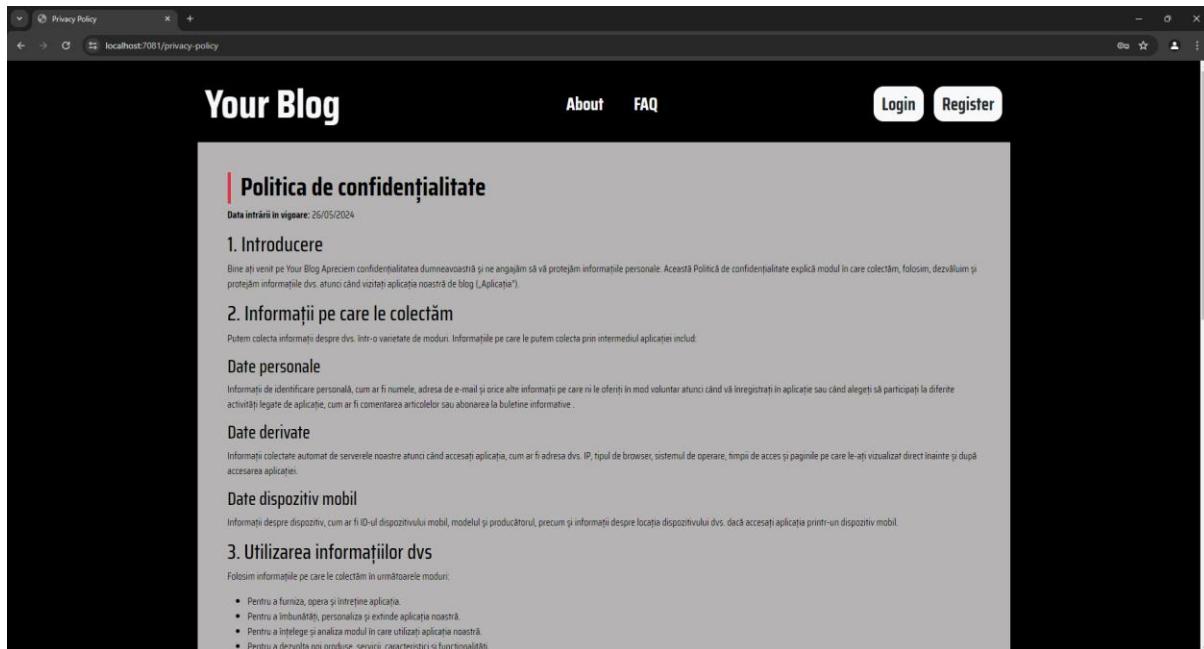


Figură 4.6. Pagina Register cu field-urile completate

Utilizatorul trebuie să accepte termenii și condițiile aplicației. Poate găsi mai multe informații despre aceștia facând click pe link-ul din pagina de register sau din secțiunea *footer*.



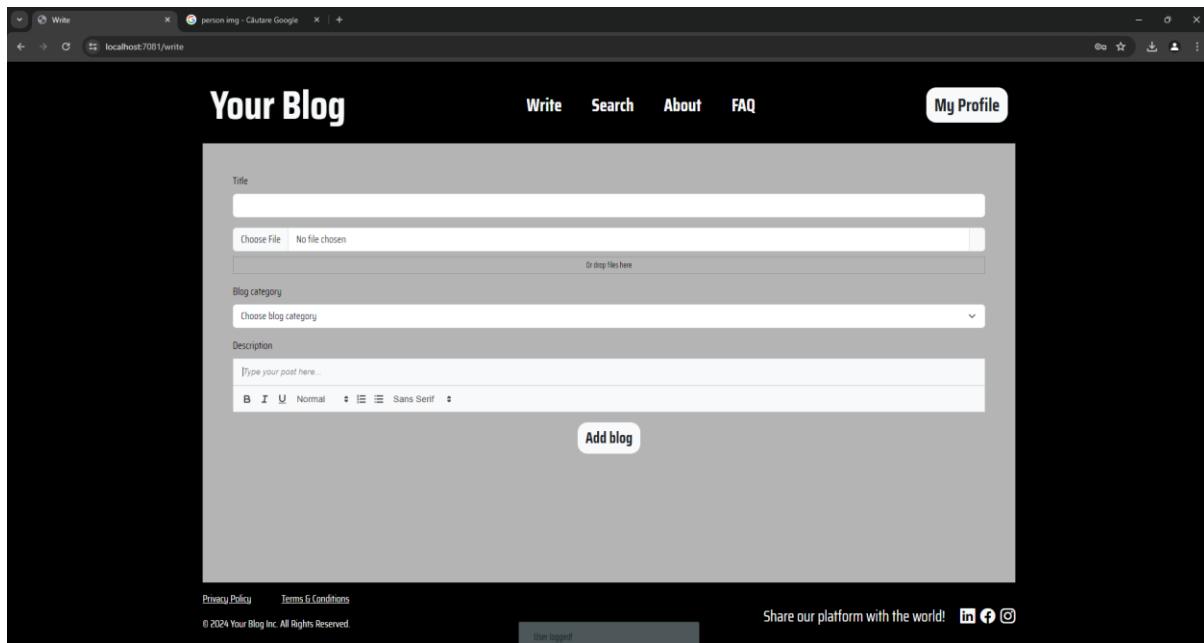
Figură 4.7. Termeni și condiții



Figură 4.8. Politica de confidențialitate

După înregistrare, utilizatorul este redirecționat către pagina de Login pentru a se autentifica. După autentificare, utilizatorului i se deblochează posibilitatea de a vedea profilul lui sau al celorlalți utilizatori, de își edita profilul, de a vedea bloguri, de a crea bloguri și de a comenta.

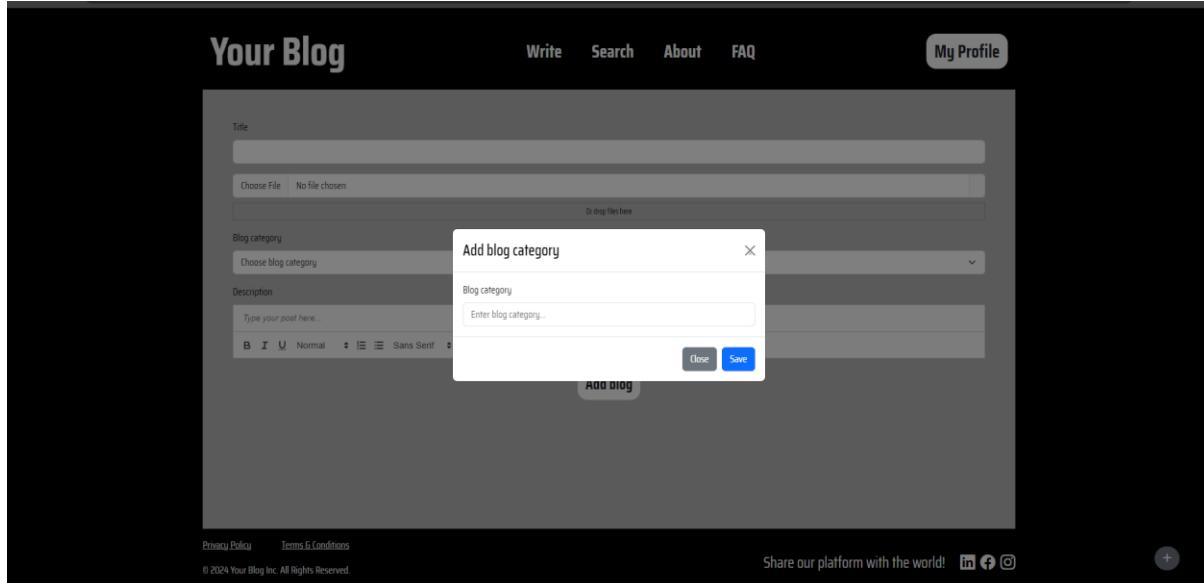
Pentru a scrie un blog, user-ul trebuie să facă click, din bara de navigare pe *Write*.



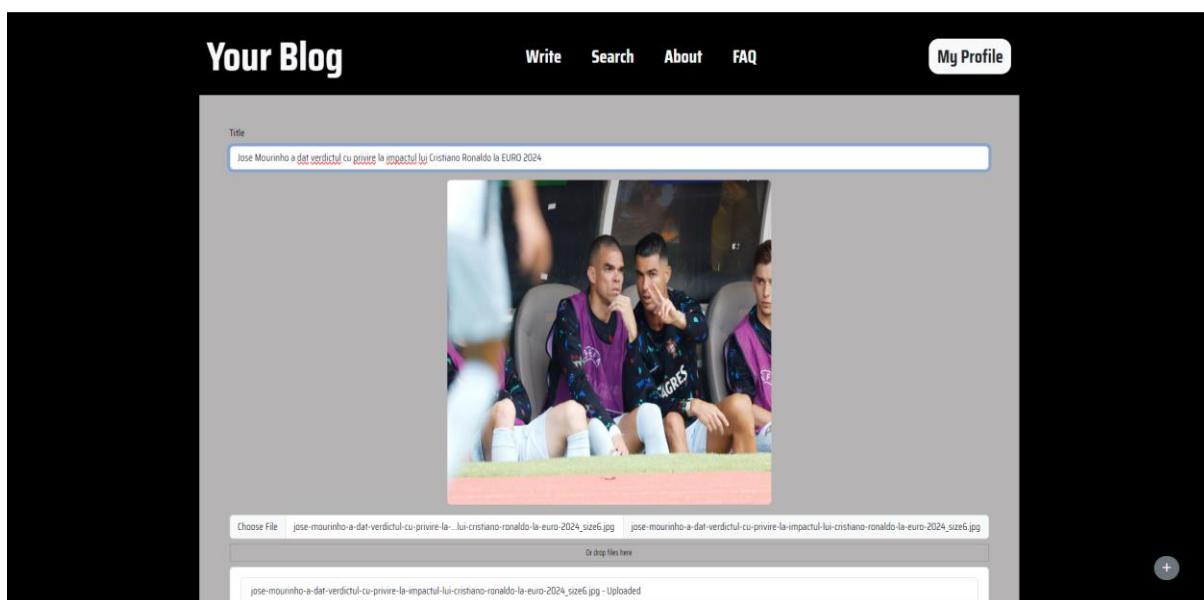
Figură 4.9. Write page

Aici, editorul mimează un *WYSIWYG*¹⁷editor și oferă posibilitatea user-ului să adauge un titlu, o imagine, o descriere și să selecteze o categorie. User-ul poate schimba font-ul, mărimea font-ului dar și decorațiile acestuia (**bold**, *italic*, underline).

Pentru *Admini*, apare un buton suplimentar, de unde poate crea o categorie:

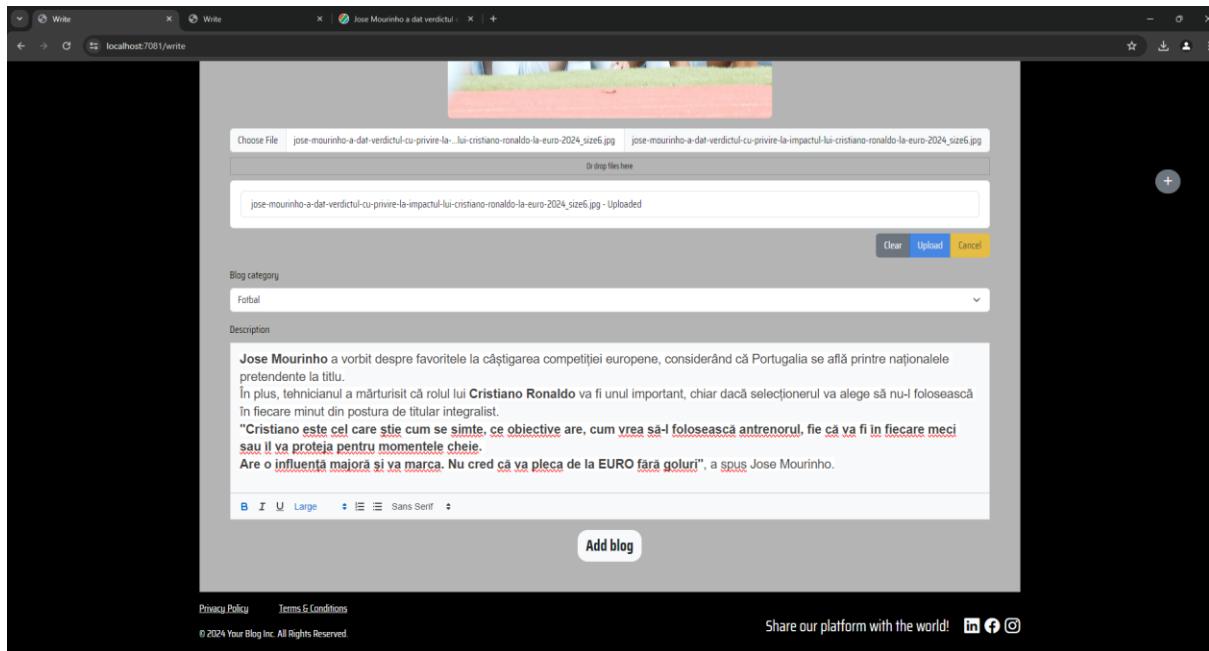


Figură 4.10. Popup-ul folosit pentru a adăuga o categorie de blog



Figură 4.11.a. Exemplu de blog

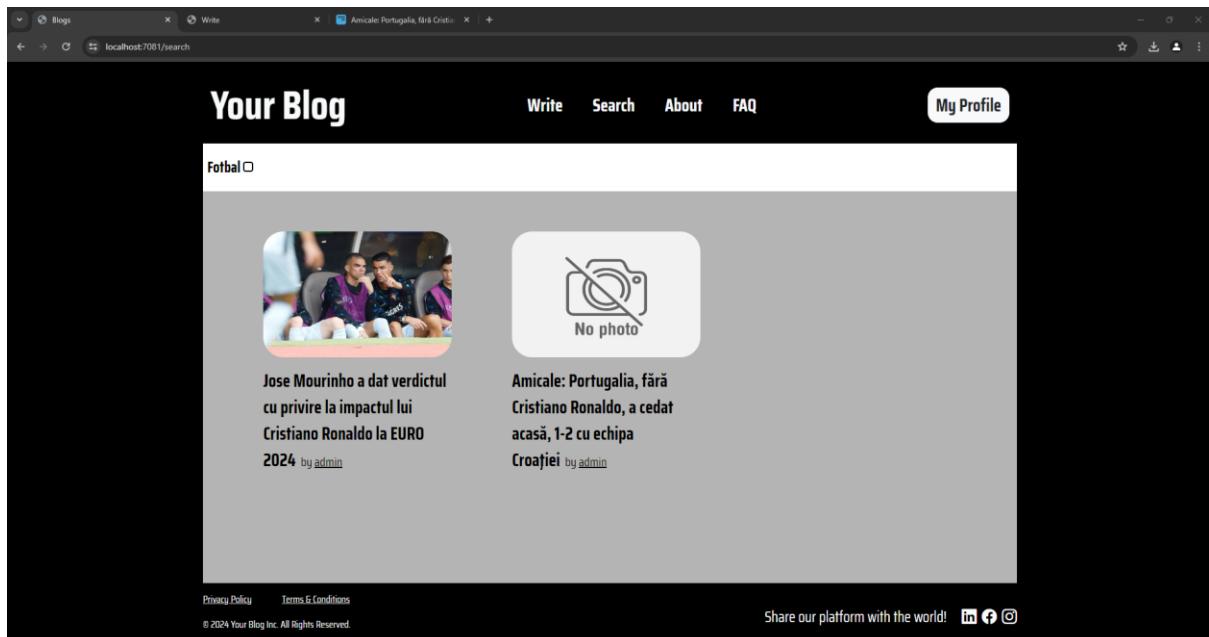
¹⁷ WYSIWYG - what you see is what you get



Figură 4.11.b. Exemplu de blog

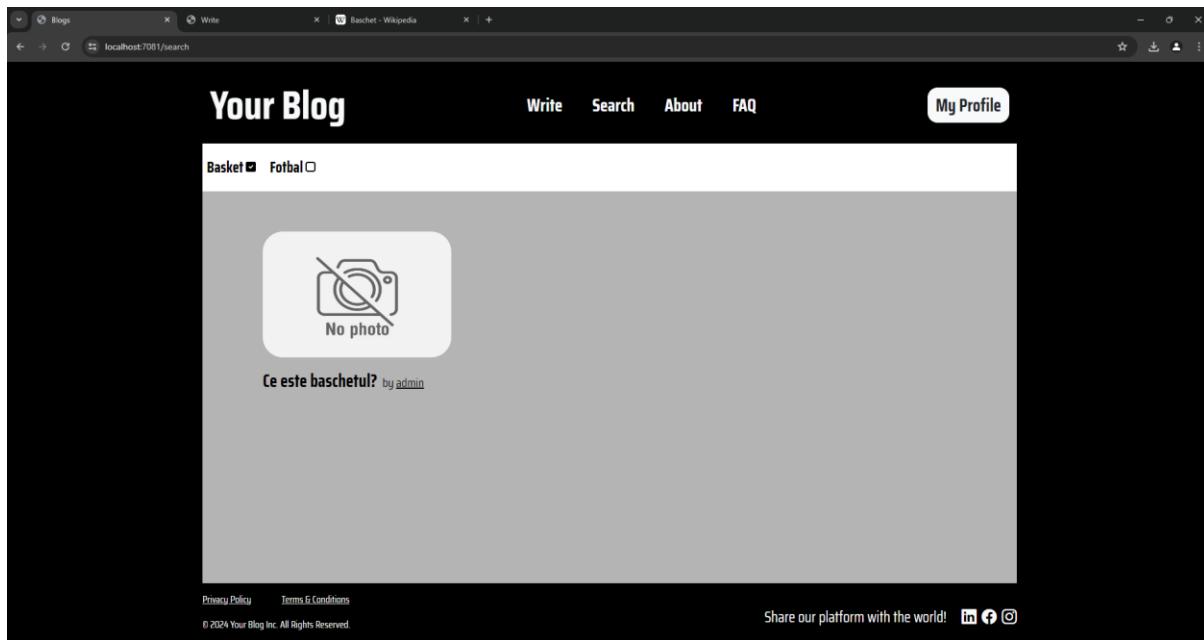
După completarea informațiilor utilizatorul apasă butonul *Add blog* și este notificat că blogul a fost adăugat cu succes. Platforma atribuie o imagine predefinată în cazul în care nu există una.

Apoi, user-ul este redirecționat către pagina de search, unde poate vedea toate blogurile existente. Acestea pot fi filtrate după categorie sau pot fi căutate din bara de navigare după titlu sau utilizatorul care le-a postat:



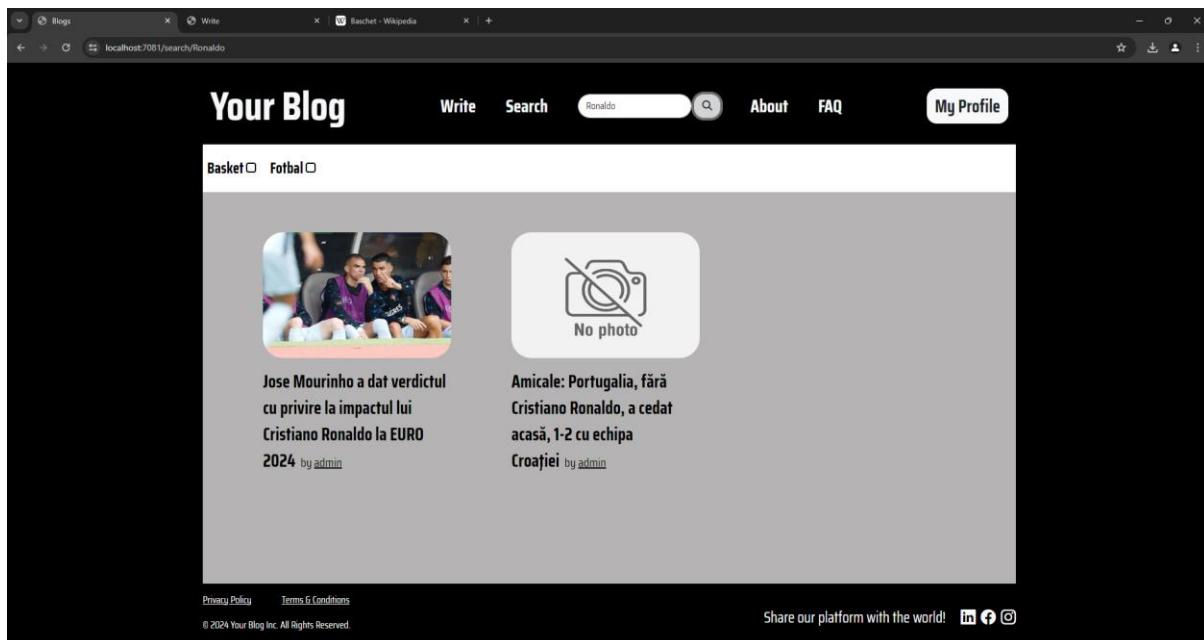
Figură 4.12. Search page

Toate căutările pot fi filtrate folosind selecția din topul paginii:



Figură 4.13. Exemplu de filtrare

Utilizatorul poate să și caute titluri sau postări a altor utilizatori, făcând click pe *Search* și căutând în bara de search apărută:



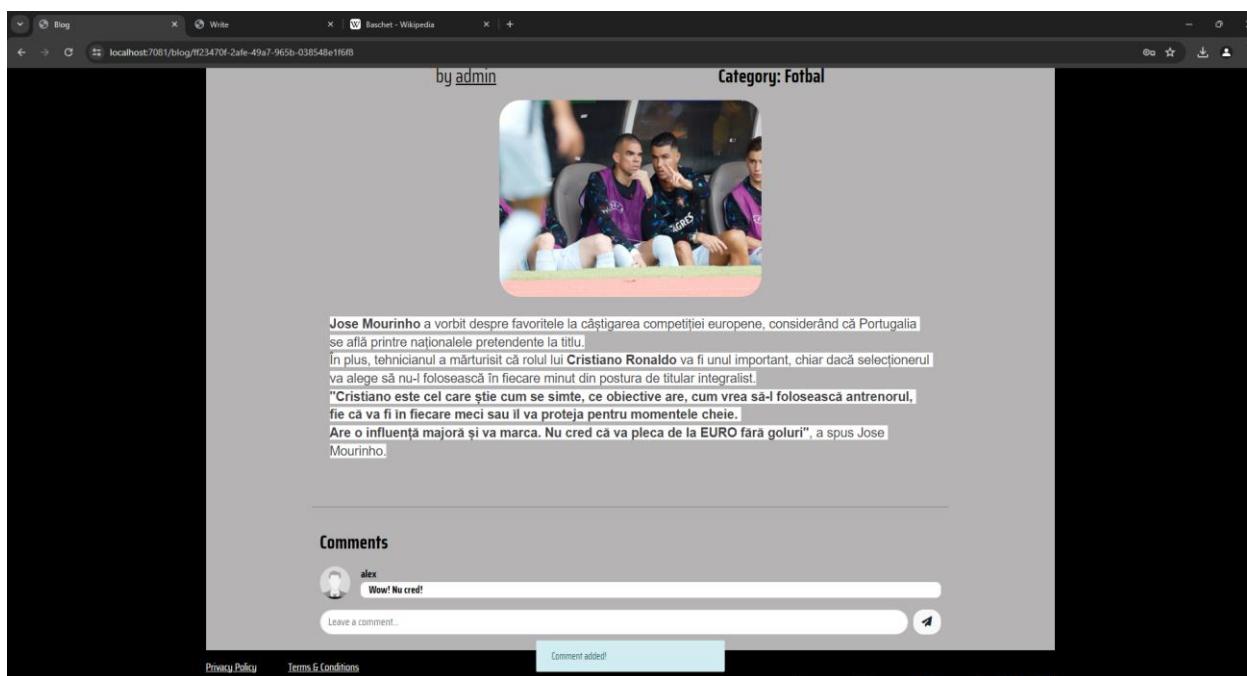
Figură 4.14. Exemplu de căutare blog

Pentru fiecare postare se găsește titlul, imaginea și creatorul. Prin apăsarea unei dintre bloguri, se deschide o pagina cu detalii blog-ului, unde, în plus, se regăsește descrierea blog-ului și comentariile:



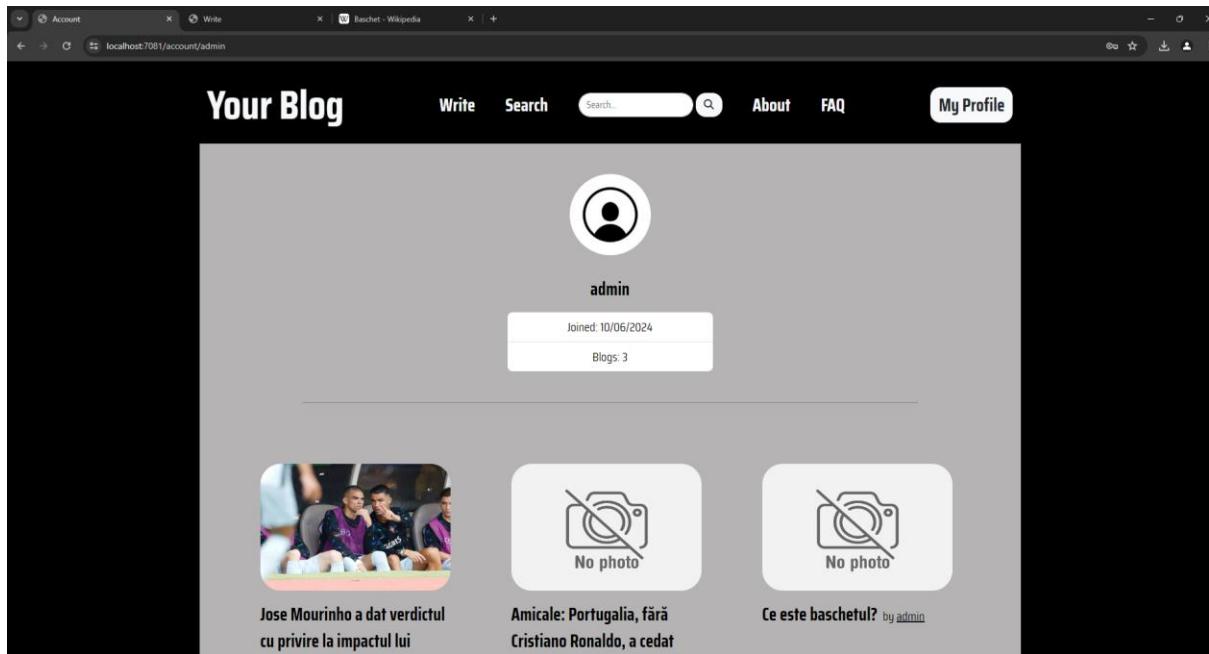
Figură 4.15. Detaliile unui blog

Aici se pot plasa comentarii și vizualiza celelalte comentarii referitoare la această postare.



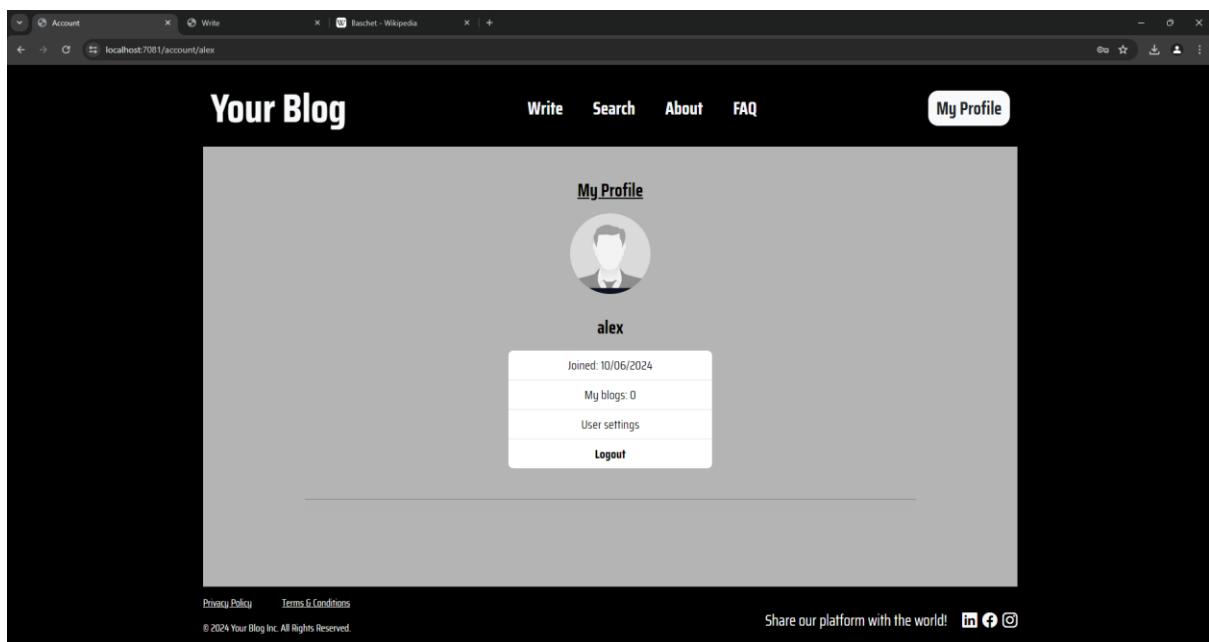
Figură 4.16. Exemplu comentarii

Pentru a vedea detalii despre utilizatorul care a postat blog-ul, utilizatorul poate face click pe username și este navigat către pagina de profil, unde regăsește informații despre utilizator și toate blog-urile postate de acesta.



Figură 4.17. Pagina de vizualizare a profilului al altui utilizator

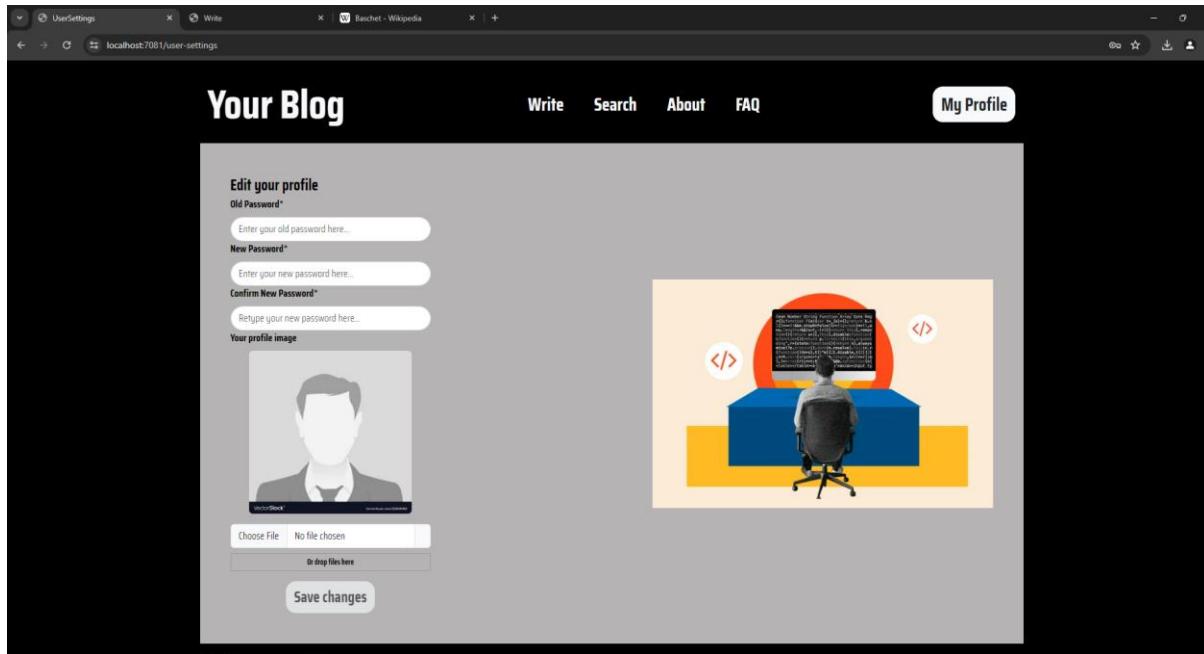
Dacă profilul este al utilizatorului care este logat, atunci aceasta poate să și editeze informațiile.



Figură 4.18. Pagina de vizualizare a profilului

Din această interfață, user-ul se poate deloga.

Pentru a edita informațiile profilului, utilizatorul trebuie să facă click pe *User settings* și poate schimba imaginea de profil și parola. Pentru parolă, acesta trebuie să reintroducă parola veche, parola nouă și confirmarea parolei noi pentru a nu exista riscul de typo.



Figură 4.19. Pagina de editare a profilului

5. SPECIFICAȚII ȘI REPREZENTAREA APLICAȚIEI

5.1. Specificatii functionale

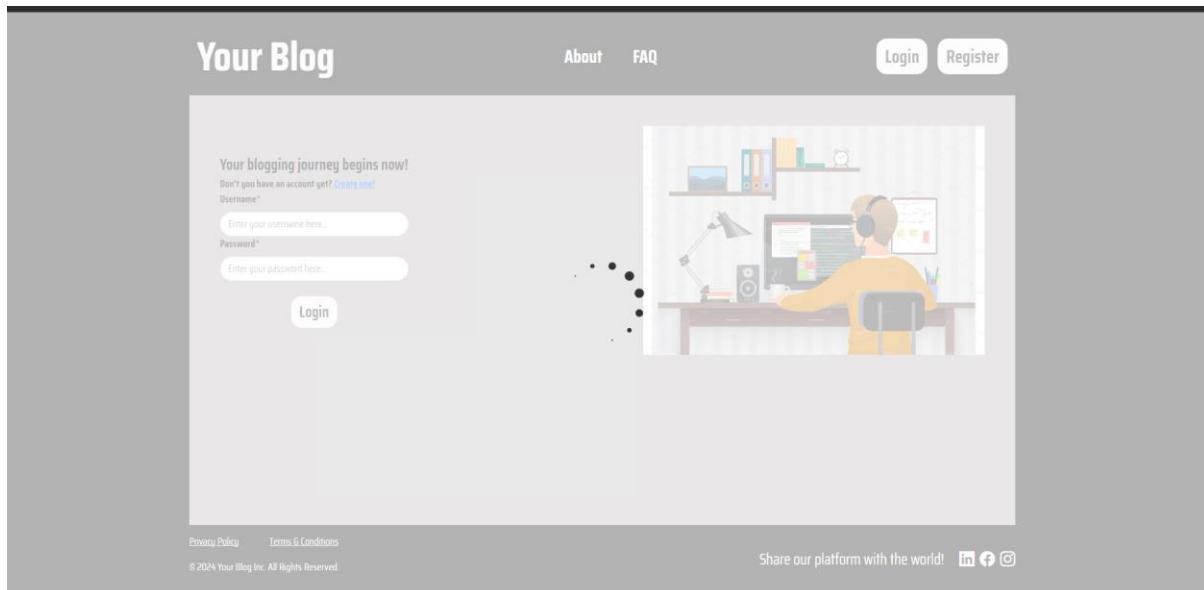
Platforma mea de blogging sau *Your blog*, sub numele în care se regăsește în proiect, este o aplicație web ce permite utilizatorilor să creeze, să publice și să vizualizeze postări ce pot conține o descriere, un titlu și o imagine. Acestea sunt grupate pe categorii, cum ar fi: fotbal, IT, music, etc. Utilizatorii sunt de 2 tipuri: useri normali și administratori. Administratorii au, în plus de utilizatorii normali, posibilitatea de a șterge blogurile și comentariile celorlalți și posibilitatea de a crea categorii de bloguri noi.

În aplicație există un singur administrator. Ceilalți useri pot fi creați prin sistemul de înregistrare al unui cont nou.

Detaliile despre utilizatori sunt: nume de utilizator (username), email, poză de profil, rol și parolă. Toate informațiile private sunt codate în baza de date.

Un user poate să adauge și comentarii la blogurile celorlalți, poate să vizualizeze profilul celorlalți, să își vizualizeze propriul profil dar și să îl editeze. Dacă utilizatorul nu își alege o imagine de profil, platforma o să îi atribuie una predefinită.

Toate comunicările dintre interfață și API sunt așteptate, timp în care utilizatorului nu îi este permis să facă vreo modificare pe pagină, prin apariția unui *ecran de încarcare*.

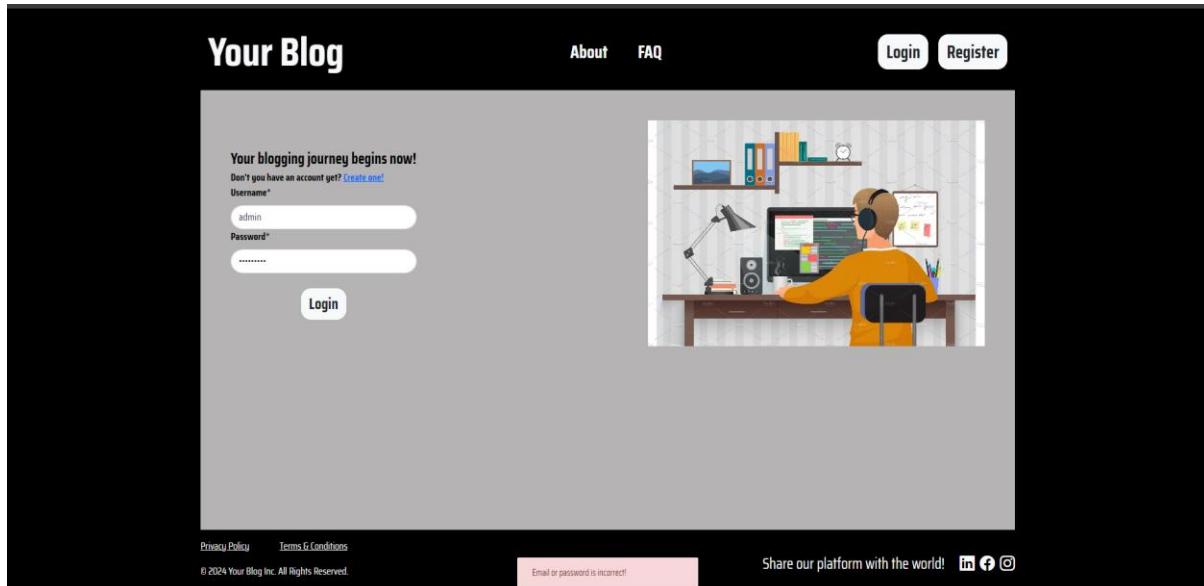


Figură 5.1. Loading screen-ul din aplicație

Utilizatorii sunt informați despre eventualele erori sau informări despre cererile lor către API printr-un *toast*, care apare în jocul paginii:



Figură 5.2. Toast-ul cu succes din aplicație



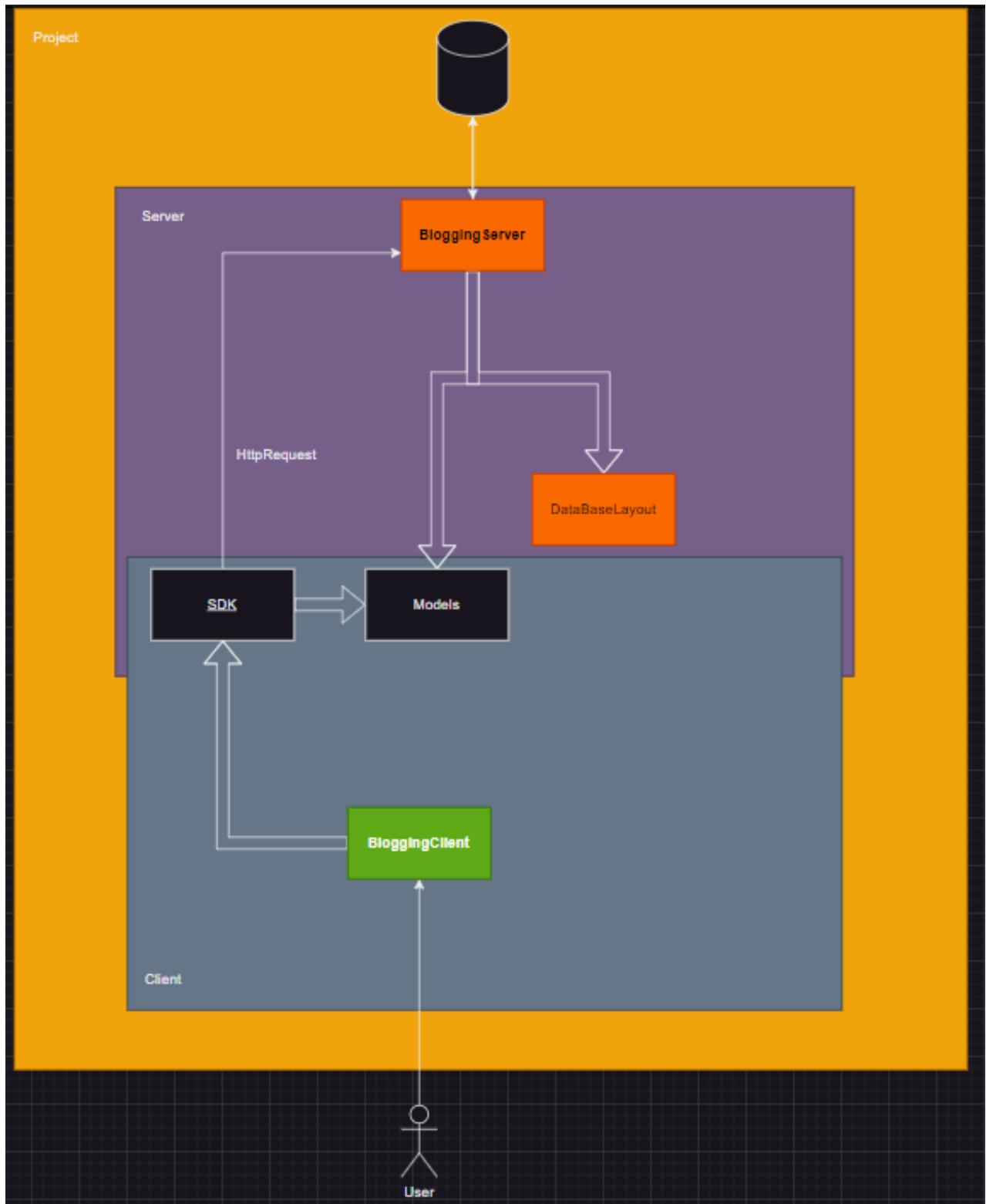
Figură 5.3. Toast-ul cu eroare din aplicație

Acestea se numesc *Snackbar* în Blazorise, sub formă de stack¹⁸ și rămân active 3 secunde.

¹⁸ O stivă (engleză stack) este o structură de date ale cărei elemente sunt considerate a fi puse unul peste altul, astfel încât orice element adăugat se pune în vârful stivei, iar extragerea unui element se poate face numai din vârful acesteia, în ordinea inversă celei în care elementele au fost introduse.

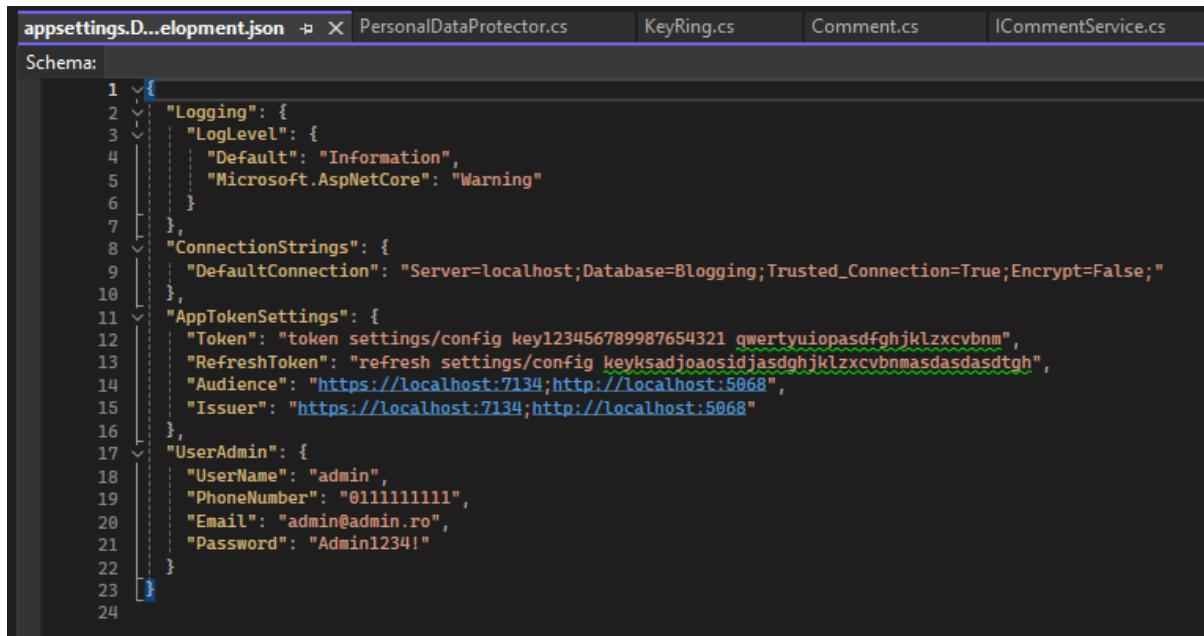
5.2. Specificații tehnice

Aplicația este împărțită în două proiecte: *BloggingServer* și *BloggingClient*.



Figură 5.4. Diagrama de reprezentare a dependenților între proiecte

BloggingServer reprezintă API-ul, proiectul prin care interfața comunică cu baza de date. Comunicarea se face prin Refit. Această soluție conține mai multe proiecte formând o structură care facilitează extinderea funcționalităților. Aceasta comunică cu baza de date folosind Entity Framework. Toate setările referitoare legate de baza de date, dar și partea de token se găsesc în fișierul *appsettings.json*:



```

1  "Logging": {
2    "LogLevel": {
3      "Default": "Information",
4      "Microsoft.AspNetCore": "Warning"
6    },
8    "ConnectionStrings": {
9      "DefaultConnection": "Server=localhost;Database=Blogging;Trusted_Connection=True;Encrypt=False;"
10   },
11   "AppTokenSettings": {
12     "Token": "token settings/config key123456789987654321 qwertyuiopasdfghjklzxcvbnm",
13     "RefreshToken": "refresh settings/config keyksadjaoasidjasdghjklzxcvbnmasdasdasdtgh",
14     "Audience": "https://localhost:7134;http://localhost:5068",
15     "Issuer": "https://localhost:7134;http://localhost:5068"
16   },
17   "UserAdmin": {
18     "UserName": "admin",
19     "PhoneNumber": "0111111111",
20     "Email": "admin@admin.ro",
21     "Password": "Admin1234!"
22   }
23 }

```

Figură 5.5. *appsettings.json*

Entity Framework folosește un *DbContext*¹⁹ pentru a accesa toate entitățile. Restul serviciilor nu trebuie să aibă acces la toate informațiile, de aceea se folosește *Repository Pattern*²⁰. Avem implementată o interfață generică care acoperă toate nevoile consumatorilor:

¹⁹ Clasa *DbContext* este o parte integrantă a Entity Framework. O instanță a lui *DbContext* reprezintă o sesiune cu baza de date care poate fi folosită pentru a interoga și a salva instanțe ale entităților într-o bază de date. *DbContext* este o combinație a modelelor Unit Of Work și Repository.

²⁰ Repository pattern este un set de reguli și practici care îmbunătățesc software-ul, oferă o abstractizare a persistenței datelor, astfel încât aplicația să poată funcționa cu o abstracție simplă (pe care o deține modelul de domeniu) care are o interfață care se aproximează pe cea a unei colecții. Adăugarea, eliminarea, actualizarea și selectarea elementelor din această colecție se face printr-o serie de metode simple, fără a fi nevoie să se ocupe de preocupările bazei de date, cum ar fi conexiunile, comenzi, cursoarele sau cititorii.

```

 9  <references>
10  public interface IRepositoryBase<T> where T : class
11  {
12      /// <summary>
13      4 references
14      void Add(T objModel);
15
16      /// <summary>
17      1 reference
18      void AddRange(IEnumerable<T> objModel);
19
20      /// <summary>
21      1 reference
22      T Get(Expression<Func<T, bool>> predicate, Func<IQueryable<T>, IIIncludableQueryable<T, object>> include = null);
23
24      /// <summary>
25      5 references
26      Task<T> GetAsync(Expression<Func<T, bool>> predicate, Func<IQueryable<T>, IIIncludableQueryable<T, object>> include = null);
27
28      /// <summary>
29      1 reference
30      IEnumerable<T> GetList(Expression<Func<T, bool>> predicate, Func<IQueryable<T>, IIIncludableQueryable<T, object>> include = null);
31
32      /// <summary>
33      4 references
34      Task<IEnumerable<T>> GetListAsync(Expression<Func<T, bool>> predicate, Func<IQueryable<T>, IIIncludableQueryable<T, object>> include = null);
35
36      /// <summary>
37      1 reference
38      int Count();
39
40      /// <summary>
41      1 reference
42      Task<int> CountAsync();
43
44      /// <summary>
45      4 references
        void Update(T objModel);
        void Remove(T objModel);
    }

```

Figură 5.6. Interfața repository-ului generic

Implementarea acestei interfețe este o clasă generică care folosește un *TEntity* ce poate fi populat cu orice model ce mapează o entitate din baza de date:

```

12  <public class RepositoryBase<TEntity> : IRepositoryBase<TEntity> where TEntity : class
13  {
14      protected readonly Context Context;
15
16      0 references
17      public RepositoryBase(IContext context)
18      {
19          Context = context as Context;
20      }
21
22      /// <inheritdoc />
23      4 references
24      public void Add(TEntity model)
25      {
26          Context.Set<TEntity>().Add(model);
27          Context.SaveChanges();
28      }
29
30      /// <inheritdoc />
31      1 reference
32      public void AddRange(IEnumerable<TEntity> model)
33      {
34          Context.Set<TEntity>().AddRange(model);
35          Context.SaveChanges();
36      }
37
38      /// <inheritdoc />
39      1 reference
40      public TEntity Get(Expression<Func<TEntity, bool>> predicate, Func<IQueryable<TEntity>, IIIncludableQueryable<TEntity, object>> include = null)
41      {
42          IQueryable<TEntity> query = Context.Set<TEntity>();
43          if (include != null)
44          {
45              query = include(query);
46          }
47
48          return query.FirstOrDefault(predicate);
49      }
50
51      /// <inheritdoc />
52      5 references
53      public async Task<TEntity> GetAsync(Expression<Func<TEntity, bool>> predicate, Func<IQueryable<TEntity>, IIIncludableQueryable<TEntity, object>> include = null)
54      {
55          IQueryable<TEntity> query = Context.Set<TEntity>();
56          if (include != null)
57          {
58              query = include(query);
59          }
60
61          return await query.FirstOrDefaultAsync(predicate);
62      }

```

Figură 5.7. Implementarea interfeței repository-ului generic

Toate serviciile și repository-urile sunt înregistrate folosind *dependency injection*.

Dependency Injection este un concept de programare care implică furnizarea dependențelor necesare unei componente din exterior, în loc să le creeze intern. Este folosit pentru a separa responsabilitățile, crește flexibilitatea și ușurează testarea aplicațiilor software.

Codul respectă principiile *SOLID* și *design patterns*²¹, pentru a ușura extinderea, înțelegerea și complexitatea acestuia.

- Single Responsibility Principle sau Priniciul Responsabilității Unice:

- Fiecare clasă ar trebui să aibă o singură responsabilitate și, prin urmare, un singur motiv pentru a se schimba. Aceasta înseamnă că o clasă ar trebui să aibă o responsabilitate bine definită și să nu fie suprasolicitată cu multiple funcționalități.

Exemplu: În aplicația noastră de blogging, clasa Blog este responsabilă de manipularea datelor legate de postări, nu și pentru gestionarea comentariilor sau a utilizatorilor.

- Open/Closed Principle sau Priniciul Deschiderii/Închiderii:

- Entitățile (clase, module, funcții) ar trebui să fie deschise pentru extensie, dar închise pentru modificare. Acest principiu impune dezvoltatorii să extindă funcționalitățile platformei fără a modifica codul existent.

Exemplu: Dacă dorim să adăugăm un nou tip de autentificare în aplicație, putem crea o nouă clasă care extinde funcționalitatea de autentificare fără a modifica clasele existente.

- Liskov Substitution Principle sau Priniciul Substituirii Liskov:

- Obiectele de tipul unei clase derivate ar trebui să poată înlocui obiectele de tipul clasei de bază fără a strica funcționalitatea aplicației.

Exemplu: Clasa derivată User ar trebui să poată fi folosită oriunde se folosește clasa de bază IdentityUser fără a afecta funcționalitatea.

- Interface Segregation Principle sau Priniciul Segregării Interfeței:

- O interfață ar trebui să fie specifică unui client, și clienții nu ar trebui să fie forțați să implementeze interfețe pe care nu le folosesc.

Exemplu: În loc să avem o interfață mare *IBlogOperations* care include metode pentru postări, comentarii și utilizatori, ar fi mai bine să avem interfețe mai mici și mai specifice: *IBlogService*, *ICommentService* și *IUserService*.

²¹ Design pattern-urile reprezintă soluții generale și reutilizabile ale unei probleme comune în design-ul software. Un design pattern este o descriere a soluției sau un template ce poate fi aplicat pentru rezolvarea problemei, nu o bucată de cod ce poate fi aplicată direct. În general pattern-urile orientate pe obiect arată relațiile și interacțiunile dintre clase sau obiecte, fără a specifica însă forma finală a claselor sau a obiectelor implicate.

- Dependency Inversion Principle sau Prinzipiul Inversării Dependenței:
 - Modulele de nivel înalt nu ar trebui să depindă de modulele de nivel scăzut.
 - Atât modulele de nivel înalt, cât și cele de nivel scăzut ar trebui să depindă de abstracții (interfețe). Abstracțiile nu ar trebui să depindă de detalii.
 - Detaliile ar trebui să depindă de abstracții.

Exemplu: În loc ca clasa BlogController să depindă direct de o clasă BlogService, ar trebui să depindă de o interfață IBlogService. Acest lucru permite schimbarea implementării IBlogService fără a afecta BlogController.

Proiectul SDK conține toate endpoint-urile disponibile în API (BloggingServer). Toate modelele de cerere sau de răspuns sunt puse în proiectul de *Models*.

BloggingClient injectează un *ApiClient* de fiecare dată când are nevoie să facă un request către Server:

```
27
28     [Inject]
29     private ILoggingApiClient BloggingApiClient { get; set; }
```

Figură 5.8. Injectarea *ApiClient*-ului în Blazor page

Pentru a pune în așteptare pagina, până se execută cererea către API, dar și pentru a notifica rezultatul în urma finalizării request-ului, se folosesc *state*-uri. Clase injectate *singleton*, care folosesc un *Action* pentru a notifica toate componente care sunt abonate la acestea:

```
6
7     public class SnackbarState
8     {
9         public SnackbarStack Snackbar { get; set; }
10
11        public event Action OnStateChanged;
12
13        public async Task PushAsync(string message, bool isError = false)
14        {
15            await Snackbar.PushAsync(
16                message,
17                isError ? SnackbarColor.Danger : SnackbarColor.Info,
18                options: options => { options.IntervalBeforeClose = 3000; } ); // Task
19
20            NotifyStateChanged();
21        }
22
23        private void NotifyStateChanged() => OnStateChanged?.Invoke();
24    }
```

Figură 5.9. Exemplu de state (SnackBar state)

În momentul în care are loc o acțiune, componentele folosesc metoda *StateHasChanged* pentru a notifica contextul despre schimbari. Folosim interfața *IDisposable*, care apelează ca un destructor metoda *Dispose*, pentru a dezabona componente de la state-uri și pentru a evite *memory leaks*.

```

15
44     public void Dispose()
45     {
46         SnackbarState.OnStateChange -= StateHasChanged;
47         LoadingState.OnStateChange -= StateHasChanged;
48     }
49
50     protected override async Task OnInitializedAsync()
51     {
52         SnackbarState.OnStateChange += StateHasChanged;
53         LoadingState.OnStateChange += StateHasChanged;
54
55         await GetBlogCategoriesAsync();
56     }

```

Figură 5.10. Exemplu de injectare a state-urilor

Proiectul începe de la componenta `_Host.cshtml` care folosește ca și layout: `_Layout.cshtml`. Aici se injectează toate referințele de la toate librăriile și se randează un `body`. `Body`, care este reprezentat de un routing pus la dispoziție de către Blazor, prin care trec toate paginile:

```

1  @inherits ComponentBase
2  @implements IDisposable
3
4  <SnackbarStack @ref="@SnackbarState.Snackbar"></SnackbarStack>
5
6  <Router AppAssembly="@typeof(App).Assembly">
7      <Found Context="routeData">
8          <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
9          <FocusOnNavigate RouteData="@routeData" Selector="h1" />
10     </Found>
11     <NotFound>
12         <PageTitle>Not found</PageTitle>
13         <LayoutView Layout="@typeof(MainLayout)">
14             <p role="alert">Sorry, there's nothing at this address.</p>
15         </LayoutView>
16     </NotFound>
17 </Router>
18

```

Figură 5.11. App.razor – Routing-ul în aplicație

Această componentă apelează un `MainLayout` predefinit, ce centrează conținutul folosind clasa din bootstrap `container`.

```

1  @inherits LayoutComponentBase
2
3  <PageTitle>Blogging</PageTitle>
4
5  <LoadingIndicator @ref="LoadingState.Loading">
6      <ChildContent>
7          <div class="bg-black text-light">
8              <div class="container">
9                  <header>
10                     <NavMenu></NavMenu>
11                 </header>
12                 <div class="background-grey background-height">
13
14                     @Body
15                 </div>
16
17                 <Footer></Footer>
18             </Footer>
19
20         </div>
21     </ChildContent>
22     <IndicatorTemplate>
23         <Animate Animation="Animations.ZoomIn" Auto Duration="TimeSpan.FromMilliseconds(700)">
24             <Div>
25                 <SpinKit Type="SpinKitType.Circle" Size="100px" />
26             </Div>
27         </Animate>
28     </IndicatorTemplate>
29 </LoadingIndicator>

```

Figură 5.12. MainLayout.razor – Layout-ul pe care îl folosește aplicația

Pe toate paginile există o bară de navigare și un footer, unde putem naviga către pagina de scriere a unui blog, de a vizualiza toate blogurile, profilul utilizatorului, dar și de a te loga și a te înregistra:

```

1  <nav class="navbar pt-3">
2      <div class="container d-flex flex-row flex-alignment-between">
3          <Link class="navbar-brand text-light fw-bold logo" To="/">Your Blog</Link>
4          <div class="navbar-nav fs-3 fw-bold d-flex flex-row gap-5">
5              <AuthorizeView>
6                  <Authorized>
7                      <li class="nav-item">
8                          <Link class="nav-link text-light" To="/write">Write</Link>
9                      </li>
10                     <li class="nav-item">
11                         <Link class="nav-link text-light" To="/search">Search</Link>
12                     </li>
13                 </Authorized>
14             </AuthorizeView>
15             <li class="nav-item">
16                 <Link class="nav-link text-light" To="/about">About</Link>
17             </li>
18             <li class="nav-item">
19                 <Link class="nav-link text-light" To="/faq">FAQ</Link>
20             </li>
21         </div>
22         <ul class="navbar-nav d-flex flex-row gap-3">
23             <AuthorizeView>
24                 <Authorized>
25                     <li class="nav-item">
26                         <Button To="@($"/account/{@_authState.User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.Name).Value}")" Type="ButtonType.Link" Color="Color.Light" >
27                         </Button>
28                     </Authorized>
29                 <NotAuthorized>
30                     <li class="nav-item">
31                         <Button To="/login" Type="ButtonType.Link" Color="Color.Light" TextWeight="TextWeight.Bold" TextColor="TextColor.Dark" TextSize="TextSize.Heading3" Class="nav-link">
32                         </Button>
33                     <li class="nav-item">
34                         <Button To="/register" Type="ButtonType.Link" Color="Color.Light" TextWeight="TextWeight.Bold" TextColor="TextColor.Dark" TextSize="TextSize.Heading3" Class="nav-link">
35                         </Button>
36                 </NotAuthorized>
37             </AuthorizeView>
38         </ul>
39     </div>
40 </nav>

```

Figură 5.13. NavMenu.razor – implementarea bării de navigare

Din footer putem accesa și termenii și condițiile aplicației.

```

1  <Div Flex="Flex.Row.JustifyContent.Between.AlignItems.Center" Height="@Height.Px(size:107)">
2      <Div Flex="Flex.Column.JustifyContent.Center" Gap="Gap.Is3">
3          <Div Flex="Flex.Row" Gap="Gap.Is5">
4              <a href="/privacy-policy" class="text-light">Privacy Policy</a>
5              <a href="/terms-and-conditions" class="text-light">Terms & Conditions</a>
6          </Div>
7          <Paragraph>© 2024 Your Blog Inc. All Rights Reserved.</Paragraph>
8      </Div>
9      <Div Flex="Flex.Row.JustifyContent.Between.AlignItems.Center" Gap="Gap.Is4">
10         <span class="fs-4">Share our platform with the world!</span>
11         <Div Flex="Flex.Row" Gap="Gap.Is2">
12             <i class="bi bi-linkedin fs-4"></i>
13             <i class="bi bi-facebook fs-4"></i>
14             <i class="bi bi-instagram fs-4"></i>
15         </Div>
16     </Div>
17 </Div>

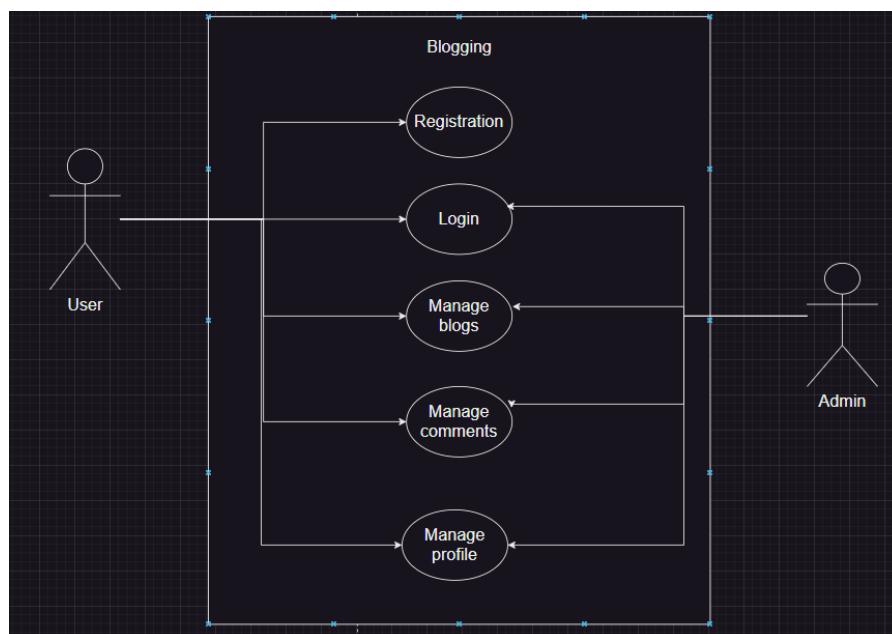
```

Figură 5.14. Footer.razor – implemtarea footer-ului

5.3. Diagramele cazurilor de utilizare

O diagramă a cazurilor de utilizare (use case diagram) prezintă o colecție de cazuri de utilizare și actori care:

- oferă o descriere generală a modului în care va fi utilizat sistemul
- furnizează o privire de ansamblu a funcționalităților ce se doresc a fi oferite de sistem
- arată cum interacționează sistemului cu unul sau mai mulți actori
- asigură faptul că sistemul va produce ceea ce s-a dorit.

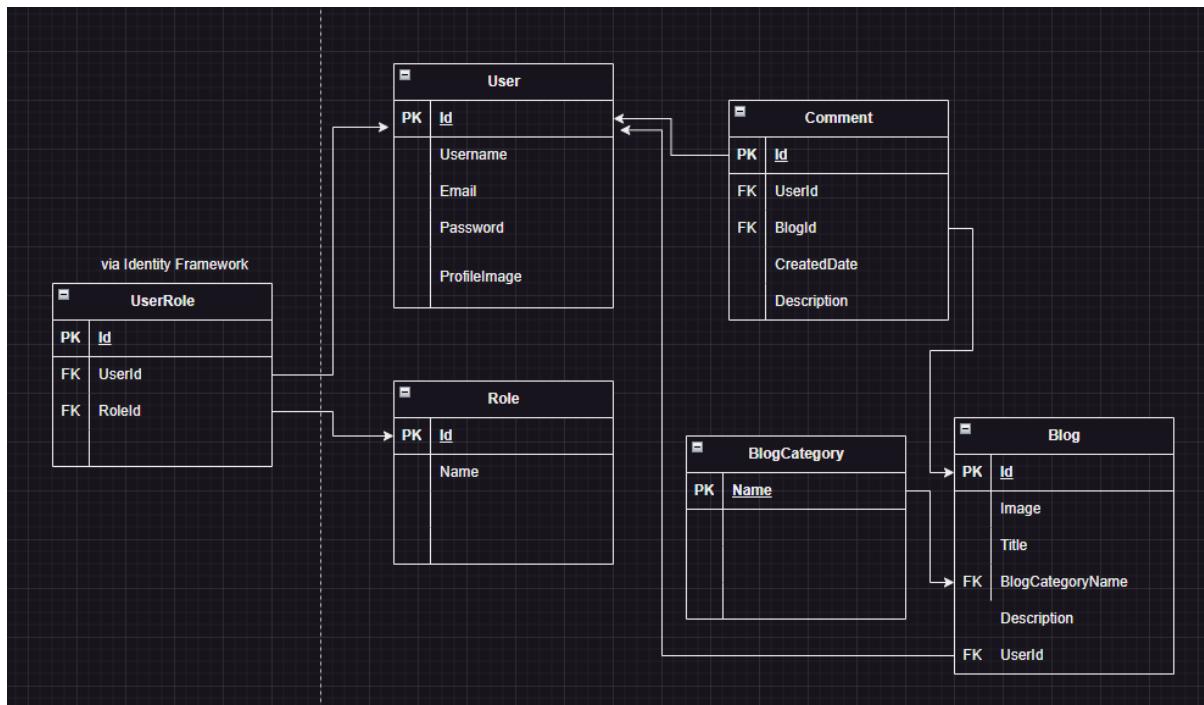


Figură 5.15. Diagrama use-case

Un actor este un stereotip al unei clase. Actorii sunt reprezentați de utilizatori sau entități care pot interacționa cu sistemul. Ei nu fac parte din sistem și definesc mulțimi de roluri în comunicarea cu acesta.

5.4. Organizarea bazei de date

Diagrama ERD este o diagramă care prezintă structura unei baze de date în termeni de entități și relațiile dintre ele. Entitățile sunt obiectele sau conceptele distincte despre care se stochează date în baza de date, iar relațiile definesc modul în care aceste entități sunt conectate sau interacționează între ele. Diagrama ERD este utilă pentru proiectarea și modelarea bazei de date, oferind o vedere vizuală asupra entităților și relațiilor lor.



Figură 5.16. Diagrama relații între entități (ERD)

Cu această structură orice entitate poate avea acces la referința ei: un blog își poate vedea creatorul, comentariile, rolul creatorului, categoria, dar și celelalte componente pot face același lucru.

6. DEZVOLTAREA APLICAȚIEI

6.1. Popularea bazei de date

Popularea bazei de date este esențială în configurarea aplicație. În cazul de față folosim SQL Server împreună cu Entity Framework și Identity pentru a gestiona autentificarea și operațiile pe date.

Entity Framework (EF) este un ORM (Object-Relational Mapper) dezvoltat de Microsoft care permite dezvoltatorilor să lucreze cu o bază de date folosind obiecte .NET. Una dintre cele trei abordări principale ale EF este Code First, care permite dezvoltatorilor să definească modelul de date folosind clase C# obișnuite (POCO - Plain Old CLR Objects) și apoi să genereze schema bazei de date pe baza acestor clase.

O entitate în Entity Framework (EF) reprezintă o clasă C# care este mapată la o tabelă din baza de date. Fiecare instanță a acestei clase corespunde unei rând din tabelă.

Code First permite dezvoltatorilor să creeze modelul de date prin scrierea de cod C#. Aceasta înseamnă că nu este necesar să existe o bază de date preexistentă, deoarece EF poate crea baza de date și tabelele pe baza modelului definit în cod. Acest lucru oferă flexibilitate și control complet asupra designului modelului de date.

Folosind *Entity Framework*, avem definit contextul de date, prin clasa *Context*, care extinde *IdentityDbContext* pentru gestiona utilizatorii, folosind *ASP.NET Core Identity*. Contextul de date definește seturile de entități care vor fi folosite în baza de date:

```

7  namespace DataBaseLayout.DbContext;
8
9  public class Context : IdentityDbContext<User, Role, string>, IContext
10 {
11     public DbSet<Blog> Blogs { get; set; }
12
13     public DbSet<BlogCategory> BlogCategories { get; set; }
14
15     public DbSet<Comment> Comments { get; set; }
16
17     public Context(DbContextOptions<Context> options)
18         : base(options) { }
19
20     public async Task<int> SaveChangesAsync()
21     {
22         return await base.SaveChangesAsync();
23     }
24 }
```

Figură 6.1. Implementarea DbContext-ului

Pentru a genera baza de date, mai întâi trebuie generate migrațiile. Migrațiile în Entity Framework (EF) sunt un mecanism prin care modificările aduse modelului de date în cod sunt reflectate în schema bazei de date. Acestea permit gestionarea și aplicarea modificărilor astfel încât baza de date este sincronizată cu modelul de date definit în cod. Pentru acest lucru se vor executa în Package Manager Console următoarele comenzi:

```
Package Manager Console
Package source: All | Default project: BloggingServer
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it
dependencies.

Package Manager Console Host Version 6.9.1.3

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

PM> add-migration Initial|
```

Figură 6.2. Comanda de adăugare a migrațiilor

Iar pentru a rula aceste migrații se va executa:

```
Package Manager Console
Package source: All | Default project: Bloggi
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it
dependencies.

Package Manager Console Host Version 6.9.1.3

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

PM> update-database
```

Figură 6.3. Comanda de rulare a migrațiilor

Toate entitățile din aplicație sunt populate din partea de Client a proiectului, de către toți utilizatorii care și-au creat un cont. În aplicație există 2 roluri: *User* și *Admin*. Acestea sunt adăugate automat de fiecare dată când server-ul rulează, dacă acestea nu există. Cu rolul *Admin* există un singur cont, care, de asemenea, este generat la fiecare runtime, dacă nu există:

```
57
58     var app : WebApplication = builder.Build();
59
60     if (app.Environment.IsDevelopment())
61     {
62         app.UseSwagger();
63         app.UseSwaggerUI();
64     }
65
66     app.UseHttpsRedirection();
67
68     app.UseAuthorization();
69
70     app.MapControllers();
71
72     await DefaultDataAsync();
73
74     app.Run();
75
76     return;
```

Figură 6.4. Apelarea populării bazei dacă este prima dată când se rulează din Program.cs

```

78     T reference
79     <async Task DefaultDataAsync()
80     {
81         var serviceProvider = builder.Services.BuildServiceProvider();
82         var roleManager = serviceProvider.GetService<RoleManager<Role>>();
83         var userManager = serviceProvider.GetService<UserManager<User>>();
84
85         var userRole = await roleManager.Roles.FirstOrDefaultAsync(x => x.Id == Roles.User);
86         if (userRole == null)
87         {
88             var result = await roleManager.CreateAsync(
89                 new Role()
90                 {
91                     Id = Roles.User,
92                     Name = Roles.User
93                 });
94             if (!result.Succeeded)
95             {
96                 throw new Exception(result.Errors.First().Description);
97             }
98
99         var adminRole = await roleManager.Roles.FirstOrDefaultAsync(x => x.Id == Roles.Admin);
100        if (adminRole == null)
101        {
102            var result = await roleManager.CreateAsync(
103                new Role()
104                {
105                    Id = Roles.Admin,
106                    Name = Roles.Admin
107                });
108            if (!result.Succeeded)
109            {
110                throw new Exception(result.Errors.First().Description);
111            }
112        }
113    }

```

Figură 6.5.a. Popularea bazei dacă este rulată prima dată

```

115     var adminUser:IList<User> = await userManager.GetUsersInRoleAsync(Roles.Admin);
116     if (!adminUser.Any())
117     {
118         var profileImage:byte[] = await File.ReadAllBytesAsync(path: @"./DataBaseLayout/Data/default-image-profile.jpg");
119         var user = new User
120         {
121             ProfileImage = profileImage,
122             JoinedDate = DateTime.UtcNow,
123             PhoneNumberConfirmed = true,
124             TwoFactorEnabled = false,
125             AcceptTerms = true,
126             EmailConfirmed = true
127         };
128         builder.Configuration.GetSection(key: "UserAdmin").Bind(user);
129
130         var result = await userManager.CreateAsync(user, builder.Configuration.GetSection(key: "UserAdmin:Password").Value);
131
132         if (!result.Succeeded)
133         {
134             throw new Exception(result.Errors.First().Description);
135         }
136
137         result = await userManager.AddToRolesAsync(user, roles: new List<string>() { Roles.User, Roles.Admin });
138
139         if (!result.Succeeded)
140         {
141             throw new Exception(result.Errors.First().Description);
142         }
143     }
144 }

```

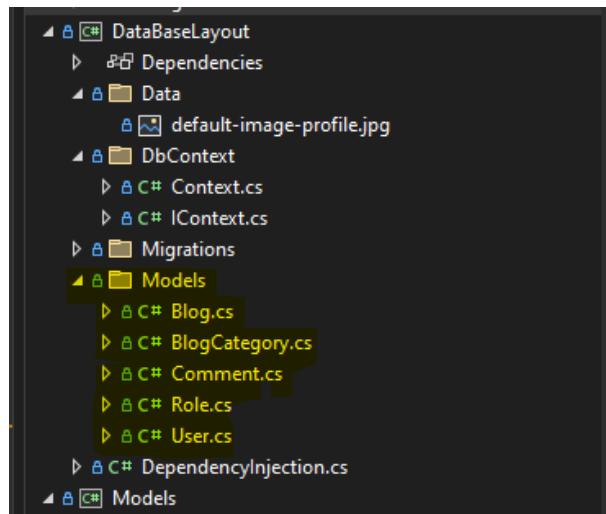
Figură 6.5.b. Popularea bazei dacă este rulată prima dată

Administratorul este configurat din *Application Configuration*, pentru a facilita deployment-ul pe mai multe environment-uri.

6.2. Modele de baze de date Entity Framework

Modelele de date în Entity Framework (EF) sunt reprezentări ale entităților din aplicație, utilizate pentru a interacționa cu baza de date. Aceste modele sunt adesea clase C#

care definesc structura și relațiile datelor pe care le gestionăm. Prin intermediul acestor modele, EF poate traduce operațiunile efectuate asupra obiectelor în comenzi SQL care, mai apoi sunt trimise către baza de date și interpretate. Aplicația noastră dispune de cinci astfel de modele:



Figură 6.6. Totalitatea entităților din aplicație scrise sub formă de clasă C#

```

1  using Microsoft.EntityFrameworkCore;
2  using System;
3  using System.Collections.Generic;
4  using System.ComponentModel.DataAnnotations;
5
6  namespace DataBaseLayout.Models;
7
8  [PrimaryKey(nameof(Id))]
9
10 public class Blog
11 {
12     public Guid Id { get; set; }
13
14     [Required]
15     public string Title { get; set; }
16
17     public string BlogCategoryName { get; set; }
18
19     [Required]
20     public BlogCategory BlogCategory { get; set; }
21
22     [Required]
23     public byte[] Image { get; set; }
24
25     public string Description { get; set; }
26
27     public DateTime CreatedTime { get; set; }
28
29     public string UserId { get; set; }
30
31     [Required]
32     public User User { get; set; }
33
34     public ICollection<Comment> Comments { get; set; }
35
36 }
```

Figură 6.7. Definiția entității unui blog

Un blog conține id-ul, titlul, o categorie, o imagine, o descriere, data creării, utilizatorul care a inițiat crearea blogului dar și mai multe comentarii.

```

4  namespace DataBaseLayout.Models;
5
6  [PrimaryKey(nameof(Name))]
7
7  public class BlogCategory
8  {
9      4 references
9      public string Name { get; set; }
10
10     2 references
11     public ICollection<Blog> Blogs { get; set; }
12 }
```

Figură 6.8. Definiția entității unei categorii de blog

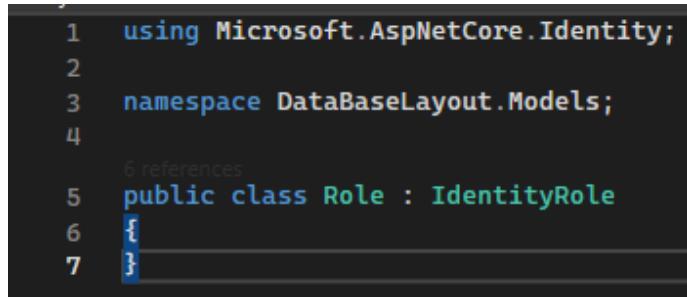
Categoria unui blog conține doar numele și blogurile care aparțin acelei categorii. Acestea sunt unice prin nume.

```

1  using System;
2  using System.ComponentModel.DataAnnotations;
3  using Microsoft.EntityFrameworkCore;
4
5  namespace DataBaseLayout.Models;
6
7  [PrimaryKey(nameof(Id))]
8
8  public class Comment
9  {
10
10     public Guid Id { get; set; }
11
11     [Required]
12
13     public string Description { get; set; }
14
14     [Required]
15
16     public DateTime CreatedDate { get; set; }
17
17
18     public string UserId { get; set; }
19
19
20     [Required]
21
21     public User User { get; set; }
22
22
23     public Guid BlogId { get; set; }
24
24
25     [Required]
25
26     public Blog Blog { get; set; }
27 }
```

Figură 6.9. Definiția entității unui comentariu

Un comentariu conține un Id, o descriere, data creării, utilizatorul care a creat comentariul și blog-ul la care a fost atribuit.



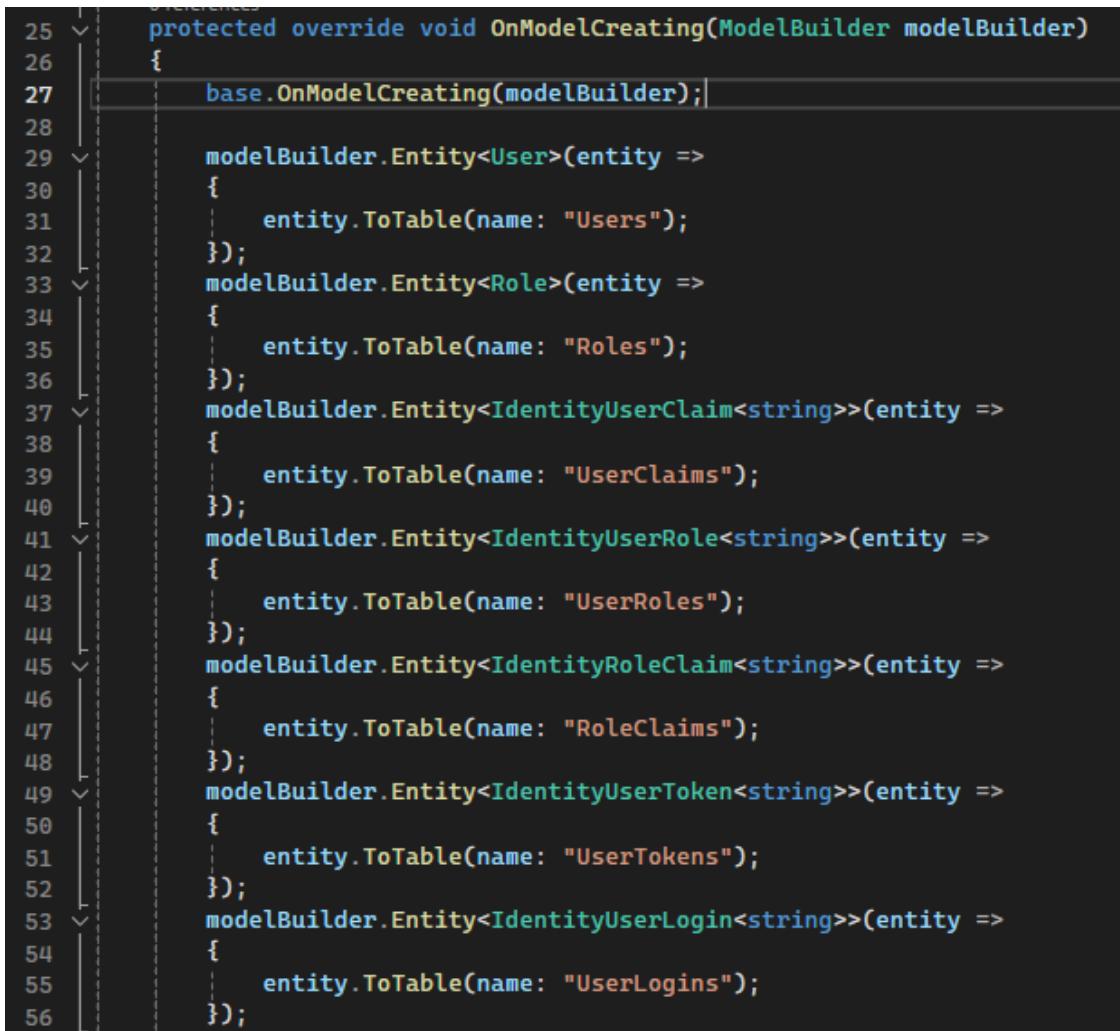
```

1  using Microsoft.AspNetCore.Identity;
2
3  namespace DataBaseLayout.Models;
4
5  public class Role : IdentityRole
6  {
7

```

Figură 6.10. Definiția entității unui rol

Role este un *IdentityRole*, ce conține numele rolului. Acesta are referințe de many-to-many cu *User*, făcute de către Identity. Această referință este suprascrisă în *Context*, pentru a păstra convențiile de nume din proiect:



```

25  protected override void OnModelCreating(ModelBuilder modelBuilder)
26  {
27      base.OnModelCreating(modelBuilder);
28
29      modelBuilder.Entity<User>(entity =>
30      {
31          entity.ToTable(name: "Users");
32      });
33      modelBuilder.Entity<Role>(entity =>
34      {
35          entity.ToTable(name: "Roles");
36      });
37      modelBuilder.Entity<IdentityUserClaim<string>>(entity =>
38      {
39          entity.ToTable(name: "UserClaims");
40      });
41      modelBuilder.Entity<IdentityUserRole<string>>(entity =>
42      {
43          entity.ToTable(name: "UserRoles");
44      });
45      modelBuilder.Entity<IdentityRoleClaim<string>>(entity =>
46      {
47          entity.ToTable(name: "RoleClaims");
48      });
49      modelBuilder.Entity<IdentityUserToken<string>>(entity =>
50      {
51          entity.ToTable(name: "UserTokens");
52      });
53      modelBuilder.Entity<IdentityUserLogin<string>>(entity =>
54      {
55          entity.ToTable(name: "UserLogins");
56      });

```

Figură 6.11. Suprascrierea numelui tabelelor create automat de Identity Framework

```

1  using System;
2  using System.Collections.Generic;
3  using Microsoft.AspNetCore.Identity;
4
5  namespace DataBaseLayout.Models;
6
7  public class User : IdentityUser
8  {
9      public byte[] ProfileImage { get; set; }
10     public bool AcceptTerms { get; set; }
11     public DateTime JoinedDate { get; set; }
12     public ICollection<Comment> Comments { get; set; }
13     public ICollection<Blog> Blogs { get; set; }
14 }
15
16
17 }
18

```

Figură 6.12. Definiția entității unui user

User definește utilizatorul și conține în plus, pe lângă proprietățile standard oferite de *IdentityUser* (username, email, etc.), imaginea de profil, dacă a acceptat termeni și condițiile, data la care s-a înregistrat în platformă și relația de one-to-many dintre acesta și *Comments*, plus *Blogs*. Astfel, un utilizator poate scrie mai multe comentarii și bloguri, însă un comentariu, respectiv un blog poate fi scris de mai mulți utilizatori.

În caz de ștergere sau actualizare, *Context*-ul este configurat să nu facă operația în cascadă pentru că toate entitățile noastre pot fi create și pot exista independent.

```

57
58     modelBuilder.Entity<Blog>().HasOne(navigationExpression: x :Blog => x.User) // ReferenceNavigationBuilder<Blog,User>
59     .WithMany(navigationExpression: x :User => x.Blogs)
60     .OnDelete(DeleteBehavior.NoAction)
61     .HasForeignKey(x :Blog => x.UserId);
62
63     modelBuilder.Entity<Blog>().HasOne(navigationExpression: x :Blog => x.BlogCategory) // ReferenceNavigationBuilder<Blog,BlogCategory>
64     .WithMany(navigationExpression: x :BlogCategory => x.Blogs)
65     .OnDelete(DeleteBehavior.NoAction)
66     .HasForeignKey(x :Blog => x.BlogCategoryName);
67
68     modelBuilder.Entity<Comment>().HasOne(navigationExpression: x :Comment => x.Blog) // ReferenceNavigationBuilder<Comment,Blog>
69     .WithMany(navigationExpression: x :Blog => x.Comments)
70     .OnDelete(DeleteBehavior.NoAction)
71     .HasForeignKey(x :Comment => x.BlogId);
72
73     modelBuilder.Entity<Blog>().HasMany(navigationExpression: x :Blog => x.Comments) // CollectionNavigationBuilder<Blog,Comment>
74     .WithOne(navigationExpression: x :Comment => x.Blog)
75     .OnDelete(DeleteBehavior.Cascade)
76     .HasForeignKey(x :Comment => x.BlogId);
77
78     modelBuilder.Entity<Comment>().HasOne(navigationExpression: x :Comment => x.User) // ReferenceNavigationBuilder<Comment,User>
79     .WithMany(navigationExpression: x :User => x.Comments)
80     .OnDelete(DeleteBehavior.NoAction)
81     .HasForeignKey(x :Comment => x.UserId);

```

Figură 6.13. Configurarea bazei în cazul de ștergere a entităților cu referință în alte tabele

Prin aceste modele se evidențiază tipul pe care fiecare proprietate ar trebui să îl aibă, valorile pe care le pot avea, dar și relațiile dintre entități: one-to-one, one-to-many, many-to-many. Astfel se facilitează mult mai ușor accesarea acestora în cod.

Pentru ca aceste referințe să fie accesate se folosește noțiunea de *AutoInclude*. *AutoInclude* este o caracteristică introdusă în Entity Framework Core 6 care permite încorporarea automată a relațiilor la interogările LINQ fără a fi nevoie de includerea explicită a acestora în cod.

```

77     modelBuilder.Entity<Blog>().Navigation(t:Blog => t.Comments).AutoInclude();
78     modelBuilder.Entity<Blog>().Navigation(t:Blog => t.User).AutoInclude();
79     modelBuilder.Entity<User>().Navigation(t:User => t.Blogs).AutoInclude();
80     modelBuilder.Entity<BlogCategory>().Navigation(t:BlogCategory => t.Blogs).AutoInclude();
81   }
82 }
```

Figură 6.14. Configurare *AutoInclude*

6.3. Sistemul de autentificare

Sistemul de autentificare este partea esențială a aplicații deoarece verifică identitatea utilizatorilor și le acordă acces la diferite resurse ale aplicației în funcție de permisiunile lor. Aceasta asigură securitatea și confidențialitatea datelor prin autentificarea utilizatorilor și gestionarea sesiunilor.

Atât autentificarea cât și autorizarea se realizează prin *Bearer Token*.

Autentificarea Bearer (numită și autentificare cu token) este o schemă de autentificare HTTP care implică jetoane de securitate numite Bearer Token. Tokenul este un șir criptic, generat de obicei de server ca răspuns la o solicitare de conectare. Clientul trebuie să trimită acest token în header-ul *Authorization* atunci când face cereri către API.

Pentru generarea acestui token, se folosește *JwtSecurityToken*:

```

31 /// <inheritdoc />
32 public async Task<string> GenerateTokenAsync(string username, int durationMin)
33 {
34   var user = _userManager.Users.FirstOrDefault(u:User => u.UserName == username);
35   var roles:IList<string> = await _userManager.GetRolesAsync(user);
36   var claims = new List<Claim>()
37   {
38     new(type:ClaimTypes.Name, value:username),
39     new(type:ClaimTypes.Email, value:user.Email),
40   };
41   claims.AddRange(collection:roles.Select(role:string => new Claim(type:ClaimTypes.Role, value:role)));
42
43   var key = new SymmetricSecurityKey(
44     Encoding.UTF8.GetBytes(_configuration.GetSection(key: "AppTokenSettings:Token").Value!));
45   var credential = new SigningCredentials(key, algorithm:SecurityAlgorithms.HmacSha512Signature);
46
47   var token = new JwtSecurityToken(claims: claims, expires: DateTime.UtcNow.AddMinutes(durationMin),
48     signingCredentials: credential);
49   return new JwtSecurityTokenHandler().WriteToken(token);
50 }
```

Figură 6.15. Generarea unui token

Acesta este apelat de către controller, în momentul în care utilizatorul inițiază operația de *Login*:

```

30  /// <inheritdoc />
31  2 references
32  public async Task<LoginResponse> SignInAsync(string userName, string password)
33  {
34
35      var user = await _userManager.FindByNameAsync(userName);
36
37      var isLoggedIn :SignInResult = await _signinManager.CheckPasswordSignInAsync(user, password, lockoutOnFailure: false);
38
39      if (isLoggedIn.Succeeded)
40      {
41
42          var token :string = await _tokenService.GenerateTokenAsync(userName, durationMin: 2);
43          var refreshToken :string = await _tokenService.GenerateTokenAsync(userName, durationMin: 8);
44
45          var responseLogin = new LoginResponse
46          {
47              AccessToken = token,
48              RefreshToken = refreshToken
49          };
50
51
52      }
53
54      throw new Exception(message: "Email or password is incorrect!");
55  }

```

Figură 6.16. Autentificarea unui utilizator

Toate rolurile utilizatorului se criptează ca și Claims²² în token. Toți utilizatorii care vor să facă diferite operații sau să acceseze resurse din platformă trebuie să ofere un token pentru a le verifica identitatea. În *backend* acest lucru se face printr-un atribut definit la fiecare request:

```

8  2 references
9  public class BloggingAuthorizationHandler : AuthorizationHandler<AuthorizationRequirement>
10 {
11
12     private readonly IHttpContextAccessor _httpContextAccessor;
13     private readonly ITokenService _tokenService;
14
15
16
17
18     protected override Task HandleRequirementAsync
19     (AuthorizationHandlerContext context, AuthorizationRequirement requirement)
20     {
21
22         var httpRequest = _httpContextAccessor.HttpContext!.Request;
23         var token :string = httpRequest.Headers["Authorization"].ToString().Replace(oldValue: "Bearer ", newValue: string.Empty);
24
25         if (!_tokenService.IsValidToken(token, requirement.RoleName))
26         {
27
28             context.Fail();
29             return Task.CompletedTask;
30
31         }
32
33         context.Succeed(requirement);
34         return Task.CompletedTask;
35     }

```

Figură 6.16. Configurarea policy-ului de verificare a fiecarui request pentru autorizare

²² Claims sunt informații afirmate despre un subiect. De exemplu, un simbol ID (care este întotdeauna un JWT) poate conține o revendicare numită *name* care afirmă numele utilizatorului ce se autentifică

Acesta are o perioadă de expirare. Dacă este expirat accesul este restricționat. Utilizatorul își poate folosi un al doilea token, numit și *refreshToken* pentru a regenera un alt token, fără a fi nevoie să repete pașii de login. Dacă și acest refreshToken este expirat atunci sesiunea se încheie și utilizatorul este nevoie să se logheze din nou. Toate acestea se fac prin intermediul Refit-ului, care în spate pune la dispoziție un *HttpClient*, prin intermediul căruia *frontend*-ul apelează controller-ele din API.

În client-side, token-ul este salvat în *LocalStorage*. Local Storage este o tehnologie de stocare web care permite aplicațiilor web să stocheze date local, direct în browser-ul utilizatorului. Aceasta este parte a specificației Web Storage și oferă o modalitate simplă și eficientă de a păstra datele pe partea clientului fără a fi nevoie de servere sau baze de date externe.

Tot acest flow este susținut de un *AuthenticationStateProvider*. Acesta este folosit ca un *CascadeParameter*, prin care notifică toate componentele că *state*-ul s-a schimbat. Blazor pune la dispoziție un tag care verifică dacă user-ul este autentificat și autorizat:

```

23   <AuthorizeView>
24     <Authorized>
25       <li class="nav-item">
26         <a href="#">@($"account/{@_authState.User.Claims.FirstOrDefault(c : Claim => c.Type == ClaimTypes.Name).Value}"</a>
27         <button Type="ButtonType.Link" Color="Color.Light" TextWeight="TextWeight.Bold" TextColor="TextColor.Dark" TextSize="TextSize.Heading3" Class="rounded-4">Log
28       </li>
29     </Authorized>
30     <NotAuthorized>
31       <li class="nav-item">
32         <a href="/login" Type="ButtonType.Link" Color="Color.Light" TextWeight="TextWeight.Bold" TextColor="TextColor.Dark" TextSize="TextSize.Heading3" Class="rounded-4">Log
33       </li>
34       <li class="nav-item">
35         <a href="/register" Type="ButtonType.Link" Color="Color.Light" TextWeight="TextWeight.Bold" TextColor="TextColor.Dark" TextSize="TextSize.Heading3" Class="rounded-4">Reg
36       </li>
37     </NotAuthorized>
38   </AuthorizeView>

```

Figură 6.17. Utilizarea autorizării în razor

Refit-ul este configurat ca la începutul fiecărui request să acceseze acest storage și să atașeze în header-ul *Authorization* token-ul de acces:

```

    RefitSettings refitSettings = new()
{
    AuthorizationHeaderValueGetter = (_, cancellationToken) => AuthBearerTokenFactory.GetBearerTokenAsync(cancellationToken)
};
services.AddRefitClient<IBloggingApi>(refitSettings)
    .ConfigureHttpClient(c : HttpClient => c.BaseAddress = url);

services.AddSingleton<IBloggingApiClient, BloggingApiClient>();

```

Figură 6.18. Configurare Refit-ului pentru a obține *AccessToken*-ul

Utilizatorul poate face o cerere de înregistrare din interfață. Toate informațiile private sunt codificate folosind inversarea caracterelor, iar parola folosește un HASH prestatibil:

```

5
6 public class PersonalDataProtector : IPersonalDataProtector
7 {
8     public string Protect(string data)
9     {
10        return new string(data?.Reverse().ToArray());
11    }
12
13    public string Unprotect(string data)
14    {
15        return new string(data?.Reverse().ToArray());
16    }
17 }

```

Figură 6.19. Configurare protejării datelor cu caracter personal

În baza de date, aceste valori o să arate de forma:

ProfileImage	AcceptTerms	JoinedDate	UserName	NormalizedUserName	Email
0xFFD8FFE000104A46494600010100025802580000FFDB00...	1	2024-06-09 18:46:17.2761562	nimda	NIMDA	or.nimda@nimda

Figură 6.20. Exemplu de encriptare a datelor în baza de date

6.4. Sistemul de blogging

Sistemul de blogging oferă utilizatorilor să vizualizeze, dar și să creeze blog-uri noi. Pentru toate acestea, *user*-ul trebuie să fie autentificat dar și autorizat cu rolul de User sau Admin:

```

23
24     [HttpGet]
25     [Authorize(policy: Roles.User)]
26
27     public async Task<IActionResult> GetBlogsAsync()
28     {
29         try
30         {
31             var result: List<Blog> = await _blogService.GetBlogsAsync();
32             return ApiServiceResponse.ApiServiceResult(new ServiceResponse<List<Blog>>(result.ToList()));
33         }
34         catch (Exception ex)
35         {
36             return ApiServiceResponse.ApiServiceResult(new ServiceResponse<List<Blog>>(ex));
37         }
38     }
39
40     [HttpGet(template: "username/{username}")]
41
42     [Authorize(policy: Roles.User)]
43
44     public async Task<IActionResult> GetBlogsByUserAsync(string username)
45     {
46         try
47         {
48             var result: List<Blog> = await _blogService.GetBlogsByUserAsync(username);
49             return ApiServiceResponse.ApiServiceResult(new ServiceResponse<List<Blog>>(result.ToList()));
50         }
51         catch (Exception ex)
52         {
53             return ApiServiceResponse.ApiServiceResult(new ServiceResponse<List<Blog>>(ex));
54         }
55     }
56
57     [HttpGet(template: "{id}")]
58     [Authorize(policy: Roles.User)]
59
60     public async Task<IActionResult> GetBlogAsync(string id)
61     {
62         try
63         {

```

Figură 6.21. Controller-ul unui blog

Admin-ul, în acest proces, are dreptul de a șterge și blog-urile celorlalți Useri.

Pentru scrierea unui blog, utilizatorul trebuie să ofere un titlu, o imagine, o descriere și o categorie. Categoriile pot fi adăugate doar de către admini și sunt unice prin nume. Aceste intrări sunt create folosind componentele de care dispune librăria Blazorise:

```

4
5     @attribute [Authorize(Roles = Roles.User)]
6
7     <PageTitle>Write</PageTitle>
8
9     <Validations @ref=" validations" Mode="ValidationMode.Manual" Model="@_addBlogModel">
10    <Div class="d-flex flex-column p-5">
11        <Validation>
12            <Field Class="mb-3">
13                <FieldLabel>Title</FieldLabel>
14                <FieldBody>
15                    <TextEdit @bind-Text=" _addBlogModel.Title">
16                        <Feedback>
17                            | <ValidationError />
18                        </Feedback>
19                    </TextEdit>
20                </FieldBody>
21            </Validation>
22            <Field Class="mb-3">
23                <FieldBody>
24                    &#039;
25                    &#039; If (_addBlogModel.Image != null)
26                    {
27                        <div class="d-flex justify-content-center align-items-center">
28                            <Figure Size="FigureSize.Is512x512">
29                                <FigureImage
29                                    | Width="@Width.Px( size: 512 )"
29                                    | Height="@Height.Px( size: 512 )"
29                                    | Source="@($"data:image/png;base64,{Convert.ToBase64String(_addBlogModel.Image)}")"
29                                    | Rounded>
30                                </FigureImage>
31                            </div>
32                        }
33                    <FilePicker class="rounded-pill" Upload="@OnImageUploaded" ShowMode="FilePickerShowMode.List" Filter="image/*">
34                        <FileTemplate>
35                            <ListGroup Class="b-file-picker_files" Margin="Margin.Is2.OnY">
36                                <ListGroupItem @key="context.File.Id" Class="b-file-picker_file">
37                                    | @context.File.Name - @context.File.Status
38                                </ListGroupItem>
39                            </ListGroup>
40                        </FileTemplate>
41                    </FilePicker>
42                </FieldBody>
43            </Field>
44        </Div>
45    </Validations>

```

Figură 6.22. Implementarea paginii Write

Pentru a adăuga descrierea unei postări, se folosește componenta RichText din Blazorise. Această este o interfață în C# pentru Quill din JavaScript:

Pentru validarea datelor se folosește componenta *Validation* care este trigger-uită de fiecare dată când se apasă butonul de *Save*:

```

93
94     private async Task AddBlogAsync()
95     {
96         if (await _validations.ValidateAll())
97         {
98             await LoadingState.ShowAsync();
99             var authState = await AuthenticationState;
100            var result : ApiResponseMessage = await BloggingApiClient.CreateBlogAsync(
101                new AddBlog()
102                {
103                    Description = _addBlogModel.Description,
104                    BlogCategoryName = _addBlogModel.BlogCategoryName,
105                    Image = _addBlogModel.Image != null ? Convert.ToBase64String(_addBlogModel.Image) : string.Empty,
106                    Title = _addBlogModel.Title,
107                    Username = authState.User.Claims.First(c:Claim => c.Type == ClaimTypes.Name).Value,
108                } // Task<ApiResponseMessage>
109
110            await LoadingState.HideAsync();
111
112            await SnackbarState.PushAsync(
113                result.Success ? "Blog created!" : result.ResponseMessage,
114                !result.Success); // Task
115
116            if (result.Success)
117            {
118                NavigationManager.NavigateTo(url: "/search");
119            }
120        }
121    }
122

```

Figură 6.23. Salvare și validarea datelor înainte de a crea cererea de adăugare a lor în bază

Imaginea se introduce de către utilizator printr-un *FilePicker* care convertește input-ul într-un *byte array*:

```

75  private async Task OnImageUploaded(FileUploadEventArgs e)
76  {
77      try
78      {
79          using var result = new MemoryStream();
80          await e.File.OpenReadStream(maxAllowedSize: long.MaxValue).CopyToAsync(result);
81
82          _addBlogModel.Image = result.ToArray();
83      }
84      catch (Exception exc)
85      {
86          Console.WriteLine(exc.Message);
87      }
88      finally
89      {
90          StateHasChanged();
91      }
92  }
93

```

Figură 6.24. Salvarea unei imagini încarcată din calculatorul utilizatorului

Metoda *StateHasChanged()* notifică toate componentele că s-au efectuat modificări pe contextul curent.

Toate blog-urile se găsesc sub pagina `/search/{parameters}` care, parcurge toate blog-urile înregistrate în platformă, și le afișează într-un format prietenos pentru utilizator:

```

1  @page "/search/{Filter?}"
2  @inherits ComponentBase
3  @implements IDisposable
4
5  @attribute [Authorize(Roles = Roles.User)]
6
7  <PageTitle>Blogs</PageTitle>
8  <div class="d-flex flex-row flex-wrap justify-content-start overflow-auto no-scrollbar w-100 bg-white">
9    <foreach (var blogCategory) .FilterCheck in _blogCategories>
10   {
11     <div class="filter-cell position-relative m-0 index-up">
12       <span class="fs-4">@blogCategory.Name</span>
13       <Check class="check-box"
14         TValue="bool"
15         CheckedChanged="@(async (t:bool) => { blogCategory.IsChecked = t; await FilterBlogsAsync(); })"
16         Checked="@blogCategory.IsChecked"></Check>
17     </div>
18   }
19 </div>
20
21 <div class="d-flex flex-row p-5 flex-wrap w-100">
22   <foreach (var blog in _blogs)>
23   {
24     <div class="blog-card">
25       <BlogCard Item="@blog" ItemClicked="@BlogClicked"></BlogCard>
26     </div>
27   }
28   @if (!_blogs.Any())
29   {
30     <div class="d-flex flex-column gap-1">
31       <div class="fs-1 fw-bold align-items-center">Blogs not found!</div>
32       <Anchor class="fs-3 fw-bold text-dark" To="/write">Write something!</Anchor>
33     </div>
34   }
35 </div>

```

Figură 6.25. Implementarea paginii de search

Pentru a pastra principiul Single Responsability, s-a creat o componentă separată pentru un card blog:

```
1 @inherits ComponentBase
2
3 <Div Flex="Flex.Column" Width="width.Px(size: 300)" Position="Position.Relative" Gap="Gap.Is3">
4   <Image Width="@Width.Px(size: 300)" Height="@Height.Px(size: 200)" Source="@($"data:image/png;base64,{Item.Image}")" class="rounded-5"></Image>
5   <div class="text-wrap">
6     <Span class="fs-3 fw-bold text-wrap me-2">@Item.Title</Span><Span class="fs-5" by <u>@Item.UserName</u></Span>
7   </div>
8   <Link Clicked="@CellClick" Stretched class="cursor-pointer">
9   </Link>
10 </Div>
```

Figură 6.26. Implementarea unui blog card

Această componentă conține un *Link* care este *Stretched*, ceea ce înseamnă că oriunde s-ar apăsa click pe acel card, se va face un fire event către acel *Link*. Acesta v-a naviga user-ul către pagina de detaliu al unui blog.

În pagina de detaliu al unui blog se regăsește în plus, descrierea și comentariile lăsate de către utilizatori:

```
1 @page "/blog/{Id}"
2 @inherits ComponentBase
3 @implements IDisposable
4
5 @attribute [Authorize(Roles = Roles.User)]
6
7 <PageTitle>Blog</PageTitle>
8
9 <div class="d-flex flex-column align-items-center gap-3 pt-5">
10   <span class="text-wrap fw-bold fs-1">@blog.Title</span>
11   <div class="d-flex flex-row justify-content-evenly container-fluid">
12     <span class="fs-3" by <Anchor Class="text-black" To="@($"account/{_blog.UserName}")">@_blog.UserName</Anchor></span>
13     <span class="fs-3 fw-bold" Category: @_blog.BlogCategory> @_blog.BlogCategory</span>
14   </div>
15   @if(!_string.IsNullOrEmpty(_blog.Image))
16   {
17     <Image Width="@Width.Px(size: 400)" Height="@Height.Px(size: 300)" Source="@($"data:image/png;base64,{@_blog.Image}")" class="rounded-5"></Image>
18   }
19   <div class="text-wrap container w-75">
20     <RichTextEdit Theme="RichTextEditTheme.Bubble"
21       ReadOnly="true"
22       TextColor="TextColor.Dark">
23       <Editor>@(_MarkupString)_blog.Description</Editor>
24     <Toolbar></Toolbar>
25   </RichTextEdit>
26 </div>
27
28 <Divider Shadow="Shadow.Default" Class="w-75 fs-3 fw-bold bg-black" style="height: 2px" />
29 <div class="d-flex flex-column container w-75 gap-3 mb-4">
30   <span class="fs-3 fw-bold" Comments: <span>
31   <div class="d-flex flex-column overflow-auto h-50 w-100">
32     @foreach (var comment in _comments)
33     {
34       <div class="d-flex flex-row gap-3">
35         <Image class="rounded-circle"
36           Width="@Width.Px(size: 50)"
37           Height="@Height.Px(size: 50)"
38           Source="@($"data:image/png;base64,{comment.UserImage}")">
39         </Image>
40         <div class="d-flex flex-column w-100">
41           <span class="fw-bold">@comment.UserName</span>
42           <div class="d-flex flex-row rounded-3 bg-white container align-items-start justify-content-start w-100">
43             <span class="text-wrap fw-bold">@comment.Description</span>
44           </div>
45         </div>
46       </div>
47     }
48   </div>
49   <div class="d-flex flex-row gap-2 align-items-center">
50     <TextEdit class="rounded-pill" Placeholder="Leave a comment..." Role="TextRole.Text" @bind-Text="_comment"></TextEdit>
51     <Button
52       Color="Color.Light"
53       TextWeight="TextWeight.Bold"
54       Clicked="@AddCommentAsync"
55       TextColor="TextColor.Dark"
56       TextSize="TextSize.Default"
57       Class="rounded-4">
```

Figură 6.27. Implementarea paginii ce conține detaliile unui blog

6.5. Sistemul de comentarii

Sistemul de comentarii permite oricărui utilizator să își exprime idei despre o anumită postare. Oricine este înregistrat poate să lase sau să vadă un comentariu din secțiunea de detaliu al unui blog:

```

35     @foreach (var comment in _comments)
36     {
37         <div class="d-flex flex-row gap-3 align-items-center">
38             <Image class="rounded-circle"
39                 Width="@Width.Px(size: 50)"
40                 Height="@Height.Px(size: 50)"
41                 Source="@$data:image/png;base64,{comment.UserImage}">
42         </Image>
43         <div class="d-flex flex-column w-100">
44             <span class="fw-bold">@comment.UserName</span>
45             <div class="d-flex flex-row rounded-3 bg-white container align-items-start justify-content-start w-100">
46                 <span class="text-wrap fw-bold">@comment.Description</span>
47             </div>
48         </div>
49         @if (_username == _blog.UserName || _isAdmin)
50         {
51             <Button
52                 Height="Height.Px(size: 40)"
53                 Width="Width.Px(size: 40)"
54                 Background="Background.Light"
55                 Clicked="@async () => { await DeleteCommentAsync(comment.Id); }"
56                 class="border-0">
57                 <i class="bi bi-trash-fill"></i>
58             </Button>
59         }
60     </div>
61 }

```

Figură 6.28.a. Implementarea sistemului de comentarii

```

40     </div>
41     <div class="d-flex flex-row gap-2 align-items-center">
42         <TextEdit class="rounded-pill" Placeholder="Leave a comment..." Role="TextRole.Text" @bind-Text="_comment"></TextEdit>
43         <Button
44             Color="Color.Light"
45             TextWeight="TextWeight.Bold"
46             Clicked="@AddCommentAsync"
47             TextColor="TextColor.Dark"
48             TextSize="TextSize.Default"
49             Class="rounded-4">
50             <Blazorise.Icon Name="@FontAwesomeIcons.PaperPlane"></Blazorise.Icon>
51         </Button>
52     </div>
53 
```

Figură 6.28.b. Implementarea sistemului de comentarii

Comentariile pot fi șterse de către autor sau de către un admin:



Figură 6.29. Comentariu cu funcționalitate de ștergere

7. CONCLUZIE

Realizarea acestui proiect de sistem de blogging a fost o oportunitate de a pune în practică cunoștințele acumulate în domeniul dezvoltării software și de a explora noi tehnologii. Proiectul utilizează Blazor cu ASP.NET Core Web API și SQL Server, facilitând o soluție completă și solidă pentru gestionarea postărilor și a comentariilor utilizatorilor.

În cadrul acestui proiect, am implementat funcționalități esențiale pentru un sistem de blogging, incluzând gestionarea utilizatorilor prin ASP.NET Core Identity, autentificare securizată folosind token-uri Bearer și stocarea datelor în baza de date SQL Server prin Entity Framework. Sistemul permite utilizatorilor să creeze, să șteargă postări, să adauge și să gestioneze comentarii, oferind astfel o platformă interactivă pentru comunicare și partajare de informații.

Pe parcursul dezvoltării, am urmărit respectarea principiilor SOLID pentru a asigura un cod modular, scalabil și ușor de întreținut. Implementarea relațiilor între tabelele bazei de date, inclusiv relațiile de tip one-to-one, one-to-many și many-to-many, a fost realizată utilizând Entity Framework, asigurând astfel o mapare eficientă a datelor.

De asemenea, am utilizat diverse instrumente și tehnologii, cum ar fi Blazor pentru dezvoltarea interfeței de utilizator, Bootstrap pentru stilizarea componentelor UI, și SQL Server Management Studio pentru gestionarea bazei de date. Am integrat și testat API-urile folosind Postman pentru a asigura că toate funcționalitățile sunt corect implementate și funcționează conform așteptărilor.

În concluzie, acest proiect de blogging reprezintă o realizare semnificativă, demonstrând capacitatea de a dezvolta o aplicație web complexă utilizând tehnologii moderne și abordări bune de inginerie software.

8. BIBLIOGRAFIE

C#. Preluat de pe *C# in Depth, Fourth Edition*, Jon Skeet, martie 2019

FullStack Full Stack Web Development. Preluat de pe *The Comprehensive Guide*, Ackermann Philip, 2023

Database. Preluat de pe *Learning SQL: Generate, Manipulate, and Retrieve Data*, Alan Beaulieu, 2020

.NET. (fără an). Preluat de pe <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet>.

ASP.NET Web API. (fără an). Preluat de pe
<https://www.tutorialsteacher.com/webapi/what-is-web-api>.

Autentificarea Bearer. (2024). Preluat de pe
<https://swagger.io/docs/specification/authentication/bearer-authentication/>.

AutoInclude. (fără an). Preluat de pe <https://learn.microsoft.com/en-us/ef/core/querying/related-data/eager>.

Blazor. (2024). Preluat de pe <https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-8.0>.

Blazorise. (2024). Preluat de pe <https://blazorise.com/docs>.

Bootstrap. (2024). Preluat de pe <https://getbootstrap.com/docs/5.3/getting-started/introduction/>.

C#. (2024). Preluat de pe <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview>.

CascadeParameter. (2024). Preluat de pe <https://learn.microsoft.com/en-us/aspnet/core/blazor/components/cascading-values-and-parameters?view=aspnetcore-8.0>.

Cascading Style Sheets (CSS). (2024). Preluat de pe <https://developer.mozilla.org/en-US/docs/Web/CSS>.

Code First. (fără an). Preluat de pe <https://learn.microsoft.com/en-us/ef/ef6/modeling/code-first/workflows/new-database>.

Dependency Injection. (fără an). Preluat de pe <https://stackify.com/dependency-injection/>.

Diagrama ERD. (fără an). Preluat de pe
<https://www.mindonmap.com/ro/blog/relationship-diagram/>.

Entity Framework (EF). (2024). Preluat de pe <https://learn.microsoft.com/en-us/ef/core/>.

entitate. (2024). Preluat de pe [https://learn.microsoft.com/en-us/previous-versions/office/developer/sharepoint-2010/ee536692\(v=office.14\)](https://learn.microsoft.com/en-us/previous-versions/office/developer/sharepoint-2010/ee536692(v=office.14)).

HyperText Markup Language (HTML). (2024). Preluat de pe
<https://en.wikipedia.org/wiki/HTML>.

Language Integrated Query (LINQ) . (2024). Preluat de pe
<https://www.tutorialsteacher.com/linq/what-is-linq>.

Local Storage . (2024). Preluat de pe
https://www.w3schools.com/html/html5_webstorage.asp.

many-to-many . (2024). Preluat de pe <https://www.entityframeworktutorial.net/code-first/configure-many-to-many-relationship-in-code-first.aspx>.

Migrații. (2024). Preluat de pe <https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli>.

NuGet Package Console. (2024). Preluat de pe <https://www.sharpcorner.com/UploadFile/22da8c/package-manager-console-in-visual-studio/>.

one-to-many. (2024). Preluat de pe <https://www.entityframeworktutorial.net/code-first/configure-one-to-many-relationship-in-code-first.aspx>.

one-to-one. (2024). Preluat de pe <https://www.entityframeworktutorial.net/code-first/configure-one-to-one-relationship-in-code-first.aspx>.

Postman. (fără an). Preluat de pe [https://en.wikipedia.org/wiki/Postman_\(software\)](https://en.wikipedia.org/wiki/Postman_(software)).

Refit. (2024). Preluat de pe <https://mwaseemzakir.substack.com/p/ep-32-using-refit-to-consume-apis>.

SOLID. (2024). Preluat de pe <https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>.

SQL Server Management Studio (SSMS) . (2024). Preluat de pe
https://en.wikipedia.org/wiki/SQL_Server_Management_Studio.

Structured Query Language (SQL) . (2024). Preluat de pe
<https://www.techtarget.com/searchdatamanagement/definition/SQL>.

Visual Studio . (2024). Preluat de pe https://en.wikipedia.org/wiki/Visual_Studio.