**UNIVERSITATEA BABEŞ-BOLYAI**

**Facultatea de Științe Economice și Gestiunea Afacerilor**

**Specializarea Informatică Economică**

**LUCRARE DE LICENȚĂ**

Aplicatie web pentru gestionarea informatiilor academice

Absolvent,

Maximilian Mihai Andrei Maximian

Coordonator științific,

Lect.univ.dr. Alexandru Stan

**Capitolul 1: Introducere**

În contextul actual al transformării digitale accelerate, instituțiile de învățământ superior sunt obligate sa modernizeze și sa creasca eficienta proceselor administrative și educaționale. Gestionarea informațiilor academice – precum notele, orarul, cererile administrative sau comunicarea între studenți și personalul administrativ – reprezintă un aspect esențial în buna funcționare a unei universități. De obicei, aceste procese implicau un volum mare de muncă manuală, documente fizice și interacțiuni birocratice, ceea ce ducea adesea la întârzieri, erori și lipsă de transparență. Prin urmare, dezvoltarea unor soluții informatice dedicate acestui scop contribuie nu doar la optimizarea activității instituțiilor de învățământ, ci și la creșterea transparenței și a satisfacției utilizatorilor finali.

Digitalizarea acestor procese aduce o serie de beneficii semnificative:

* Acces rapid și facil la informații pentru toți actorii implicați (studenți,profesori, administratori)
* Reducerea birocrației și a timpului necesar pentru procesarea cererilor și documentelor
* Creșterea transparenței în ceea ce privește notele, situația academică și comunicarea deciziilor administrative
* Adaptabilitate și mobilitate, prin accesul la platformă de pe orice dispozitiv conectat la internet
  1. **Introducere in tematica lucrarii**

Lucrarea de față are ca scop dezvoltarea unei aplicații web destinate gestiunii informațiilor academice pentru studenți și cadre didactice. Proiectul propus se dorește a fi o soluție digitală modernă și eficientă pentru accesarea notelor, orarului și a diverselor cereri administrative, contribuind astfel la digitalizarea proceselor educaționale.

Aplicația este destinată instituțiilor de învățământ și are la bază o arhitectură dezvoltată folosind tehnologia ASP.NET Core în limbajul C#. Prin implementarea unei interfețe intuitive și a unui sistem de autentificare și autorizare, aplicația va permite studenților accesul rapid la datele personale și academice iar, cadrelor didactice, posibilitatea de a gestiona aceste informații.

Obiectivele principale ale proiectului sunt:

* Implementarea unui sistem de autentificare și gestiune a utilizatorilor (student/administrator);
* Afișarea notelor într-un catalog digital accesibil studentului;
* Vizualizarea orarului în funcție de grupă/specializare;
* Descărcarea de cereri tip (ex: cerere de bursă, adeverințe);
* Permisiuni diferențiate în funcție de rolul utilizatorului;
* O interfață prietenoasă, responsivă, adaptată dispozitivelor moderne.

Metodologia utilizată în elaborarea lucrării presupune analiza cerințelor funcționale, proiectarea bazei de date, dezvoltarea aplicației în mediu Visual Studio cu ASP.NET Core, testarea funcționalităților și evaluarea performanței aplicației.

* 1. **Problemele Rezolvate de Aplicația Propusă**

Digitalizarea proceselor educaționale și gestiunea datelor academice adresează o serie de probleme concrete întâlnite în mediul academic. Aplicația web propusă în această lucrare de licență a fost concepută pentru a rezolva următoarele probleme majore:

1. Lipsa centralizării informațiilor academice

În multe universități, date precum notele, orarul, cererile sau documentele necesare studenților sunt distribuite fragmentat, pe multiple platforme sau chiar pe suport fizic. Această lipsă de centralizare duce la dificultăți în accesarea rapidă a informațiilor și la riscul de pierdere sau neactualizare a acestora.

**Soluție**: Aplicația oferă o platformă unică, centralizată, unde studenții și cadrele didactice pot accesa, modifica și gestiona toate informațiile relevante, reducând timpul de așteptare și erorile de comunicare.

2. Ineficiența proceselor administrative

Procesele tradiționale de depunere a cererilor (bursă, adeverințe, etc.) implică de multe ori completarea manuală, deplasări la secretariat și timp îndelungat de procesare.

**Soluție**: Platforma permite generarea și descărcarea automată a cererilor tip, completate cu datele personale ale studentului. Astfel, se elimină birocrația, iar procesele devin mai rapide și mai transparente.

3. Lipsa transparenței și accesului facil la note și orar

Studenții pot întâmpina dificultăți în a-și verifica situația școlară la timp, iar comunicarea privind orarul poate fi adesea întârziată sau neclară.

**Soluție**: Catalogul digital și modulul de afișare a orarului permit accesarea în timp real a notelor și a programului, în funcție de grupă sau specializare.

4. Securitatea datelor și diferențierea permisiunilor

Accesul necontrolat la informații sensibile sau modificarea datelor de către persoane neautorizate reprezintă riscuri majore în gestionarea informațiilor academice.

**Soluție**: Sistemul de autentificare și autorizare pe bază de roluri asigură accesul diferențiat, protejând datele personale și academice. Autentificarea securizată, parolele hash-uite și gestionarea conturilor doar pe bază de invitație previn accesul neautorizat.

5. Adaptarea la dispozitive moderne și experiență prietenoasă

Multe platforme vechi nu oferă o experiență fluidă și intuitivă pentru utilizatori.

**Soluție**: Interfața aplicației este responsive, adaptată pentru orice dispozitiv, asigurând o experiență modernă și plăcută atât studenților, cât și administratorilor.

**Capitolul 2 – Tehnologii utilizate**

**2.1 ASP.NET**

ASP.NET este un framework puternic, dezvoltat de Microsoft, proiectat să faciliteze crearea de aplicații web dinamice, scalabile și sigure. Lansat în 2002, ca o parte componentă a platformei .NET Framework, ASP.NET a progresat rapid de-a lungul anilor, ajungând astăzi să fie una dintre cele mai utilizate tehnologii pentru construirea aplicațiilor web. În prezent, versiunea modernă – ASP.NET Core – este integral open-source, cross-platform și oferă performanțe ridicate, fiind compatibilă cu Windows, Linux și macOS. [1]

ASP.NET le oferă dezvoltatorilor posibilitatea de a crea aplicații web robuste, utilizând limbaje precum C#, oferind suport pentru arhitecturi moderne precum MVC (Model-View-Controller), Razor Pages, Blazor (pentru aplicații interactive cu C# în browser) și Web API (pentru servicii RESTful). Această flexibilitate ofera posibilitatea alegerii celei mai potrivite abordări în funcție de complexitatea și cerințele proiectului.

Unul dintre principalele avantaje ale ASP.NET este integrarea nativă cu ecosistemul Microsoft, ceea ce facilitează conectarea rapidă la baze de date SQL Server, utilizarea serviciilor Azure, autentificarea prin Active Directory și integrarea cu alte tehnologii precum Entity Framework Core și LINQ.

Pe partea de securitate, ASP.NET oferă un set complet de mecanisme moderne, printre care:

* Autentificare și autorizare bazate pe roluri și politici; [2]
* Protecție împotriva atacurilor comune (XSS, CSRF, SQL Injection); [3]
* Criptarea datelor sensibile și gestionarea securizata a sesiunilor;
* Integrare cu IdentityServer pentru autentificare. [4]

În ceea ce privește performanța, ASP.NET Core este recunoscut pentru viteza sa de execuție crescută, datorită motorului Kestrel și optimizărilor la nivel de runtime. De asemenea, suportă deployment modular, ceea ce înseamnă că aplicațiile pot fi livrate cu doar componentele necesare, reducând dimensiunea și timpul de încărcare.

Un alt aspect important este instrumentarea și suportul pentru dezvoltare. ASP.NET este complet integrat în Visual Studio, unul dintre cele mai puternice medii de dezvoltare, care oferă funcționalități precum IntelliSense, debugging avansat, testare automată și publicare directă în cloud.

Pe scurt, ASP.NET este o alegere solidă pentru dezvoltarea aplicațiilor web moderne, oferind: [5]

* Stabilitate și suport pe termen lung;
* Performanță ridicată și scalabilitate;
* Securitate avansată;
* Flexibilitate în arhitectură și design;
* Integrare excelentă cu alte tehnologii Microsoft și servicii cloud.

**2.2 ASP.NET Core MVC**

ASP.NET Core MVC reprezintă una dintre cele mai importante evoluții din cadrul ecosistemului .NET, fiind o platformă modernă, open-source și cross-platform, destinată dezvoltării aplicațiilor web robuste și scalabile. Această tehnologieîmbinăputerea framework-ului ASP.NET Core cu arhitectura Model-View-Controller (MVC), oferind dezvoltatorilor un cadru clar și bine structurat pentru organizarea codului și separarea responsabilităților.

Arhitectura pe care se bazează, Model-View-Controller (MVC), separă clar responsabilitățile într-o aplicație web: [6]

Modelul – reprezintă partea de logica a aplicației. Include clasele care definesc structura datelor, regulile de validare, manipulare si accesare a datelor și logica de acces la bazele de date. Această componentă nu depinde de celelalte două, prin urmare poate funcționa în mod independent.

View-ul – este componenta care definește modul în care informația este prezentată utilizatorului. Afișează datele din Model într-un format vizibil plăcut. Fiind o componenta pasiva, nu detine logica, avand rolul doar de a prelua interactiunile utilizatorilor si afisarea datelor. De obicei, folosește fișiere Razor (.cshtml) care îmbină HTML cu C# pentru a produce pagini dinamice. Totodata, prin intermediul unui View, utilizatorii trimit date catre Controller cu scopul ca acestea sa fie procesate ulterior.

Controller-ul – funcționează ca un intermediar între utilizator și aplicație. Primește cereri din partea utilizatorului, procesează aceste cereri cu ajutorul logicii de business și returnează răspunsuri corespunzătoare. Controller-ul, așadar, are ca scop primar coordonarea fluxului de date dintre Model și View, asigurând independența dintre cele două.

O imagine care conține diagramă, text, Plan, Desen tehnic

Conținutul generat de inteligența artificială poate fi incorect.

Fig. 1 – Diagrama flux MVC

Această separare clară a responsabilităților între componentele arhitecturii MVC aduce numeroase beneficii în procesul de dezvoltare. Codul devine mai ușor de întreținut și testat, întrucât fiecare parte a aplicației are un rol bine definit și izolat. Acest lucru permite echipelor de dezvoltare să colaboreze mai eficient, lucrând simultan pe diferite module fără a interfera unele cu altele. În plus, extinderea aplicației cu noi funcționalități se poate realiza rapid și sigur, fără riscul de a afecta alte componente deja existente.

ASP.NET Core MVC vine cu o serie de caracteristici moderne care contribuie la o dezvoltare eficientă și flexibilă. Sistemul de rutare este foarte bine structurat, permițând definirea clară a adreselor URL și asocierea acestora cu acțiuni specifice din controller. Suportul pentru middleware oferă posibilitatea de a introduce funcționalități personalizate în fluxul de procesare a cererilor HTTP, ceea ce sporește controlul asupra comportamentului aplicației.

Un avantaj major al platformei este portabilitatea. Fiind construit pe .NET Core, aplicațiile dezvoltate cu ASP.NET Core MVC pot fi rulate pe mai multe sisteme de operare, inclusiv Windows, Linux și macOS. Această flexibilitate este esențială pentru echipele care lucrează în medii diverse sau care doresc să implementeze aplicații în infrastructuri cloud precum Microsoft Azure, Amazon Web Services sau containere Docker. [7]

În plus, ASP.NET Core MVC este optimizat pentru performanță și scalabilitate, fiind potrivit atât pentru aplicații de dimensiuni reduse, cât și pentru sisteme complexe, cu un număr mare de utilizatori. Platforma oferă suport nativ pentru mecanisme moderne precum dependency injection, care facilitează o testare maieficientă și un mod mai simplu de a activa/dezactiva anumite componente. [7]

**2.3 Entity Framework**

În dezvoltarea aplicațiilor moderne, gestionarea eficientă a datelor este esențială, iar în ecosistemul .NET, Entity Framework (EF) reprezintă soluția oficială oferită de Microsoft pentru maparea obiect-relatională (ORM – Object-Relational Mapping). Acest framework permite dezvoltatorilor să interacționeze cu baze de date relaționale folosind obiecte și clase din C#, fără a fi necesară scrierea manuală a interogărilor SQL. Astfel, se creează o punte intuitivă între logica aplicației și structura bazei de date, ceea ce simplifică semnificativ procesul de dezvoltare si asigura accesul la baza de date intr-un mod uniform.

Entity Framework automatizează conversia operațiunilor efectuate asupra obiectelor din cod în comenzi SQL corespunzătoare, gestionând în mod transparent conexiunile, tranzacțiile și sincronizarea datelor. Această abordare reduce considerabil riscul apariției erorilor de interogare și contribuie la o dezvoltare mai rapidă și mai sigură.

Un instrument util oferit de EF este sistemul de migrații (migrations) ,care actualizează structura bazei de date pe baza modificărilo rmodelelor aplicației. Astfel,dezvoltatorii pot dezvolta schema bazei de date fără a pierde datele existente, păstrând evidența modificărilor.

Entity Framework oferă și alte funcționalități precum validarea datelor pe baza unor reguli declarative. Aceste caracteristici contribuie la menținerea integrității datelor și la reducerea codului redundant, permițând dezvoltatorilor să se concentreze mai mult pe logica aplicației decât pe detaliile tehnice ale interacțiunii cu baza de date.

O imagine care conține text, captură de ecran, diagramă, Font

Conținutul generat de inteligența artificială poate fi incorect.În special în proiectele de dimensiuni mari sau în aplicațiile enterprise, unde gestionarea manuală a bazelor de date ar presupune un efort semnificativ, utilizarea Entity Framework aduce un plus de eficiență și organizare. Prin abstractizarea detaliilor tehnice și oferirea unui model de lucru orientat pe obiecte, EF devine un instrument esențial în arsenalul oricărui dezvoltator .NET care urmărește să construiască aplicații robuste, scalabile și ușor de întreținut.

Fig XX

**2.4 Limbajul C#**

C# (pronunțat „*Csharp*”) este limbajul de programare principal din platforma. NET, fiind un pilon în dezvoltarea aplicațiilor moderne, inclusiv a celor construite cu ASP.NET Core MVC. Creat de Microsoft, C# este un limbaj modern, lansat in anul 2000, care ruleaza pe framework-ul .NET. Este robust și orientat pe obiect, conceput pentru performanță, siguranță și ușurința de utilizare. [8]

Un atuu al limbajului C# este tipizarea puternică, cu verificări de tip la compilare și execuție, reducând riscul de erori. Sintaxa sa este clară și intuitivă,inspirată din limbajele Java și C++, facilitând învățarea pentru programatorii familiarizați cu aceste tehnologii.Totuși, un dezavantaj este necesitatea de ascrie mai mult cod,atât pentru conversia datelor (ex.string si int),cât și pentru codul "boilerplate".În același timp, C# oferă funcționalități moderne pentrua plicații complexe, modulare și ușor de întreținut.

Limbajul integrează concepte fundamentale ale programării orientate pe obiect – moștenirea, încapsularea și polimorfismul – permițând organizarea logică a codului și reutilizarea eficientă a componentelor. Aceste caracteristici sunt esențiale în proiecte de dimensiuni mari, unde claritatea și structura codului influențează direct calitatea și mentenabilitatea aplicației.

Un alt avantaj major este suportul pentru programarea sincronă, prin async și await. Acest model permite executarea operațiilor care implică resurse externe (baze de date,servicii web) fără a bloca firul principal,oferind o experiență mai bună utilizatorului. [9]

C# oferă, de asemenea, instrumente puternice pentru manipularea datelor, cum ar fi LINQ (Language Integrated Query), care interoghează colecțiile de obiecte într-un mod expresiv și concis, similar cu interogările SQL, dar mai lizibil. Această funcționalitate este utilă în combinație cu Entity Framework, facilitând accesul și procesarea datelor din baze de date relaționale. [10]

Integrarea strânsă cu mediul de dezvoltare Visual Studio și cu întregul ecosistem .NET face din C# un limbaj extrem de productiv. Visual Studio oferă suport complet pentru debugging, testare, refactorizare și publicare, precum și extensii ce facilitează dezvoltarea și reduc timpul de livrare.

Prin combinarea acestor calități – claritate, siguranță, performanță și flexibilitate – C esteo alegere ideală pentruaplicațiiweb moderne. În acest proiect, C# este utilizat pentru logica de bussiness, definirea modelelor de date și interacțiunea cu baza de date prin Entity Framework, contribuind astfel la realizarea unei aplicații stabile, scalabile și ușor de extins.

**2.5 ASP.NET Identity**

O imagine care conține text, captură de ecran, linie, Font

Conținutul generat de inteligența artificială poate fi incorect.ASP.NET Identity este sistemul modern de gestionare a autentificării și autorizării în cadrul aplicațiilor dezvoltate pe platforma ASP.NET și ASP.NET Core. Acest sistem a fost conceput pentru a răspunde nevoilor tot mai complexe legate de securitatea aplicațiilor web, oferind un cadru flexibil, extensibil și sigur pentru gestionarea utilizatorilor, a parolelor, a rolurilor și a drepturilor de acces. Spre deosebire de sistemele mai vechi, precum Membership Provider, ASP.NET Identity permite o integreaza mecanisme moderne de autentificare, inclusiv autentificare pe bază de token (cum ar fi JWT), autentificare externă prin rețele sociale (Google, acebook, Microsoft, Twitter) și integrarea cu sisteme enterprise, precum Azure Active Directory.

Fig. XX Schema bazei de date generată de ASP.NET Core Identity [24]

Principiul fundamental al ASP.NET Identity se bazează pe abstractizarea notiunii de "utilizator" și a tuturor activităților asociate, fără a ține cont de sursa de date sau de modalitatea de autentificare. În acest context, dezvoltatorul are capacitatea de a crea cu ușurință modele personalizate de utilizator, adaptate necesităților specifice ale aplicației. Sistemul utilizează o arhitectură modulară, în care componentele fundamentale – gestionarea utilizatorilor, a parolelor, a rolurilor și a permisiunilor – pot fi extinse sau înlocuite conform cerințelor. De exemplu, implicit, informațiile despre utilizatori și parolele sunt conservate într-o bază de date relațională (de obicei SQL Server), însă ASP.NET Identity permite și integrarea altor surse de date, dacă este necesar.

Un aspect crucial al ASP.NET Identity este gestionarea securizată a parolelor. Parolele utilizatorilor sunt, prin default, stocate criptat, prin utilizarea algoritmilor de tip hash și salt, ceea ce asigură o protecție semnificativă împotriva atacurilor de tip brute-force sau rainbow table. În plus, sistemul oferă posibilitatea stabilirii unor politici riguroase legate de complexitatea parolelor, expirarea acestora și blocarea contului după un număr predeterminat de încercări de autentificare eșuate. Aceste măsuri contribuie la creșterea nivelului general de securitate al aplicației. [12]  
 ASP.NET Identity permite, de asemenea, gestionarea rolurilor și a permisiunilor, facilitând implementarea unui sistem detaliat de autorizare. Rolurile pot fi definite fie de dezvoltator, fie de administratorul aplicației, utilizatorii putând fi asociați unuia sau mai multor roluri, iar permisiunile de acces la diferite resurse sau funcționalități fiind controlate cu precizie, în funcție de rolurile atribuite. Acest model facilitează realizarea unor scenarii complexe, cum ar fi diferențierea între utilizatori obișnuiți, administratori sau moderatori, fiecare având drepturi distincte în cadrul aplicației. [11]  
 În ceea ce privește implementarea, sistemul se bazează pe o arhitectură modulară, în care fiecare componentă îndeplinește un rol bine definit. UserManager se ocupă de gestionarea operațiunilor legate de utilizatori, inclusiv crearea, modificarea sau ștergerea conturilor, schimbarea parolelor și atribuirea de roluri. Acesta furnizează o interfață unificată pentru toate acțiunile legate de conturile de utilizator, simplificând astfel implementarea logicii de autentificare. Simultan, RoleManager se ocupă de administrarea rolurilor din aplicație – crearea, modificarea, eliminarea acestora și asocierea lor cu utilizatorii. Prin intermediul acestor roluri, aplicația poate reglementa accesul la diverse funcționalități, având astfel un sistem de autorizare detaliat și eficient.  
 Modelul de utilizator este reprezentat prin clasa User, care încadrează toate informațiile relevante referitoare la un cont: numele de utilizator, adresa de email, parola (stocată în mod securizat sub formă de hash), precum și alte date personalizate, cum ar fi numele complet, imaginea de profil sau data înregistrării. Acest model este simplu de extins pentru a se adapta cerințelor specifice ale aplicației.  
 Un alt beneficiu semnificativ al ASP.NET Identity constă în suportul său nativ pentru autentificarea externă. Prin colaborarea cu furnizori precum Google, Facebook sau Microsoft, utilizatorii au posibilitatea de a accesa aplicația utilizând conturile deja existente pe aceste platforme, eliminând astfel necesitatea de a crea un cont separat. Aceasta facilitează procesul de integrare și promovează adoptarea aplicației.  
 În plus, ASP.NET Identity a fost conceput pentru a permite extensibilitatea și personalizarea. Dezvoltatorii au capacitatea de a adăuga cu ușurință câmpuri adiționale la modelul de utilizator (de exemplu, nume, prenume, adresă, fotografie de profil etc.), pot stabili politici personalizate de autentificare sau pot să dezvolte mecanisme de autorizare care să răspundă cerințelor specifice ale proiectului.

Prin intermediul acestor caracteristici, ASP.NET Identity oferă un cadru modern, sigur și adaptabil pentru gestionarea identității utilizatorilor într-o aplicație web, facilitând astfel implementarea rapidă a funcționalităților de autentificare, autorizare și administrare a utilizatorilor, fără a pune în pericol securitatea sau scalabilitatea aplicației.

**2.6 ASP.NET REPOSITORY**

În arhitectura aplicațiilor moderne dezvoltate cu ASP.NET Core MVC, modelul Repository are un rol esențial în organizarea logicii de acces la date. Acest model arhitectural vizează separarea responsabilităților între logica de afaceri și cea de persistență a datelor, contribuind astfel la o structură mai clară, flexibilă și ușor de întreținut.  
 Repository-ul funcționează ca un intermediar între aplicație și sursa de date, oferind o interfață abstractă pentru operațiile CRUD (Create, Read, Update, Delete). În loc ca aceste operații să fie implementate direct în controlere sau alte componente ale aplicației, ele sunt centralizate într-o clasă dedicată – repository-ul – responsabilă cu gestionarea interacțiunii cu baza de date. Astfel, componentele aplicației care necesită accesul sau modificarea datelor nu trebuie să fie familiarizate cu detalii despre tehnologia de stocare, ci doar să utilizeze metodele disponibile prin intermediul repository-ului. [13]   
 În cadrul unei aplicații ASP.NET Core, repository-ul este de regulă implementat ca o interfață generică, precum IRepository sau IGenericRepository, care definește metodele esențiale pentru manipularea entităților: obținerea tuturor înregistrărilor, căutarea după criterii, adăugarea, actualizarea și ștergerea acestora. Această interfață este implementată ulterior de una sau mai multe clase concrete care folosesc contextul bazei de date furnizat de Entity Framework pentru a realiza operațiile necesare. [14]  
 Unul dintre principalele avantaje ale acestui model este testabilitatea. Deoarece repository-ul oferă o interfață abstractă, implementarea reală poate fi substituită cu o versiune simulată în timpul testelor unitare. Aceasta permite testarea logicii de afaceri independent de baza de date, conducând la un cod mai robust și ușor de întreținut. În plus, dacă se dorește o schimbare a tehnologiei de stocare în viitor, modificările necesare se vor limita la repository, fără a impacta restul aplicației.  
 Modelul Repository sprijină, de asemenea, scalabilitatea aplicației. Pe măsură ce proiectul se extinde, adăugarea de noi entități sau extinderea funcționalităților devine mai simplă datorită structurării bine definite și izolate a logicii de acces la date. Repository-urile pot fi specializate pentru fiecare entitate sau implementate generic, utilizând tipuri generice pentru a gestiona mai multe tipuri de date cu același cod reutilizabil.  
 Astfel, utilizarea modelului Repository în aplicațiile ASP.NET Core MVC aduce un avans semnificativ în organizare, claritate și flexibilitate. Separarea logicii de afaceri de cea de acces la date facilitează o dezvoltare mai curată, testabilă și extensibilă, contribuind la menținerea unui cod bine structurat și sustenabil pe termen lung.

**2.7 MODELUL UNIT OF WORK**

O imagine care conține text, captură de ecran, Font

Conținutul generat de inteligența artificială poate fi incorect.În cadrul proiectului, modelul Unit of Work interacționează strâns cu modelul repository, având rolul de a gestiona aplicarea tranzacțiilor la nivel de SQL. Atunci când se dorește inserția de date în multiple tabele, modelul Unit of Work facilitează acțiunile de Rollback în cazul apariției unor probleme, asigurându-se astfel că informațiile sunt salvate împreună, având relevanță doar în contextul integrității lor. Dacă toate inserțiile sau modificările sunt finalizate cu succes, acestea sunt aplicate într-o singură tranzacție, garantând consistența datelor. [14]

Fig XX Utilizarea Unit Of Work [25]

**Capitolul 3 – Proiectarea aplicatiei**

**3.1 Cerinte**

Este necesar un dispozitiv cu acces la internet, compatibil cu browsere moderne. Un aspect esențial al soluției dezvoltate este securitatea, astfel încât un sistem de înregistrare și autentificare securizat trebuie să fie proiectat. În cadrul aplicației există trei tipuri de utilizatori: studenți, profesori și administratori de sistem. În funcție de rolurile menționate anterior, soluția oferă diverse funcționalități pentru utilizatorii autentificați. Funcționalitatea principală constă în gestionarea activităților academice, necesitând implementarea unor procese specializate în acest sens. Gestionarea utilizatorilor se face prin intermediul unui sistem bazat pe roluri, fiecare tip de utilizator având permisiuni și funcționalități dedicate.  
 Administratorul este utilizatorul cu cele mai extinse drepturi și responsabilități în aplicație. Acesta are rolul de a gestiona platforma la nivel global, asigurând securitatea și buna funcționare a sistemului. Printre atribuțiile sale principale se numără invitarea și crearea conturilor pentru studenți și profesori, gestionarea și modificarea rolurilor și permisiunilor tuturor utilizatorilor, precum și monitorizarea activităților din platformă. Administratorul are acces la toate datele academice, având capacitatea de a adăuga, edita sau șterge informații despre utilizatori, note, orare, grupe sau discipline și poate interveni oricând pentru corectarea sau actualizarea acestora.  
 Profesorul este cadru didactic, membru al personalului universitar, care utilizează aplicația pentru a gestiona activitățile educaționale și evaluarea studenților. Profesorul are acces la propriul orar, în care sunt incluse cursurile și seminariile pe care le susține, având posibilitatea de a vizualiza lista studenților corespunzători fiecărei grupe sau discipline pe care o coordonează. Un aspect crucial al rolului său este capacitatea de a introduce, modifica sau valida notele la materiile predate, asigurând o evidență corectă și actualizată a parcursului educațional al studenților. Totuși, profesorul nu are acces la datele personale ale altor profesori sau studenți, în afara celor pe care îi predă, și nu poate invita sau gestiona alți utilizatori, această responsabilitate revenind exclusiv administratorului.  
 Studentul este utilizatorul final al aplicației, reprezentând o persoană înscrisă la cursurile universitare, care folosește platforma pentru a accesa informații personale și academice. Acesta poate vizualiza notele obținute la diferite discipline prin catalogul digital, consulta orarul personalizat în funcție de grupă sau specializare și are acces la funcția de generare și descărcare a cererilor tip (precum cererile de bursă sau adeverințele). În ceea ce privește editarea datelor, studentul poate modifica doar anumite aspecte ale profilului personal, cum ar fi datele de contact, fără a interveni asupra informațiilor critice sau asupra datelor altor utilizatori. De asemenea, accesul studentului la sistem este strict limitat la propriile date, acesta neavând capacitatea de a adăuga, modifica sau șterge informații academice sau de a gestiona alți utilizatori.

**Diferențe principale între Student, Profesor și Administrator**

|  |  |  |  |
| --- | --- | --- | --- |
| **Funcționalitate** | **Student** | **Profesor** | **Administrator** |
| Vizualizare note | Doar propriile note | Notele tuturor studenților la disciplina predată | Toate notele, pentru orice student sau disciplină |
| Vizualizare orar | Orar personalizat pe grupă/specializare | Orar cu disciplinele predate | Orarul tuturor grupelor, specializărilor și profesorilor |
| Adăugare/modificare note | Nu | Da, pentru disciplinele proprii | Da, pentru orice disciplină sau student |
| Descărcare cereri tip | Da | Nu | Poate administra modelele de cereri |
| Acces la lista de studenți | Nu | Da, pentru grupele/disciplinele predate | Da, pentru toți studenții și toate grupele |
| Generare rapoarte | Nu | Da (opțional) | Da, la nivel global și detaliat |
| Editare profil | Doar date personale | Doar date personale | Poate edita datele tuturor utilizatorilor |
| Administrare utilizatori | Nu | Nu | Da, poate adăuga, modifica, șterge sau dezactiva orice utilizator |
| Invitare utilizatori | Nu | Nu | Da, pentru studenți și profesori |
| Gestionare structură academică | Nu | Nu | Da, poate crea/edita grupe, discipline, orare |
| Resetare parole/conturi | Nu | Nu | Da, pentru orice utilizator |
| Acces la rapoarte și statistici | Nu | Limitat, doar pentru studenții/disciplinele proprii | Da, la nivel de platformă |

Fix. XX Diferentele dintre roluri [16]

O imagine care conține text, diagramă, desen, cerc

Conținutul generat de inteligența artificială poate fi incorect.

Fig. XX Diagrama utilizatori

**3.2 Securizarea datelor în ASP.NET - Register si Login**

În cadrul aplicației, procesele de înregistrare și autentificare sunt implementate astfel încât să respecte cele mai bune practici de securitate oferite de platforma ASP.NET Core.

1. Înregistrarea unui utilizator (Register)

Procesul de înregistrare începe, de regulă, doar pe baza unei invitații trimise de administrator, pentru a preveni accesul neautorizat. Link-ul de inregistrare este generat de catre sistem si are forma de UUID. Odată ce utilizatorul accesează link-ul de invitație, el completează un formular cu datele personale și parola dorită. Parola introdusă NU este stocată niciodată în clar în baza de date. În schimb, ASP.NET Identity folosește o abordare modernă și sigură pentru a proteja datele de autentificare: [17]

* La fiecare înregistrare, sistemul generează un "salt" unic (o secvență aleatoare de biți) pentru fiecare utilizator.
* Parola este combinată cu acest salt și apoi este trecută printr-o funcție de hash criptografică rezistentă la atacuri (de exemplu PBKDF2, implicit în ASP.NET Core Identity).
* O imagine care conține text, captură de ecran, Font, linie

  Conținutul generat de inteligența artificială poate fi incorect.Rezultatul (hash-ul + salt-ul) este stocat în baza de date, nu parola originală. Astfel, chiar dacă baza de date ar fi compromisă, parolele reale nu pot fi recuperate ușor prin atacuri de tip dicționar.

Fig. XX Parolele identice au hash identic

O imagine care conține text, Font, captură de ecran, linie

Conținutul generat de inteligența artificială poate fi incorect.

Fig XX Parolele identice dar cu salt-ul aplicat vor avea hash diferit

Pentru trimiterea mailurilor folosim serviciul SMTP de la Google, iar pentru securizarea datelor de conectare am folosit conceptul de secrete, pentru a evita “*hardcodarea*” datelor sensibile. Secretele sunt un mod securizat de stocare la nivel local a datelor. Activarea secretelor se face folosind comanda “*dotnet user-secrets init*”. Adaugarea de noi secrete se face folosing comanda “*dotnet user-secrets set "Service:Key" "Value"”.* [18]. Acestea sunt stocate sub forma de fisier “.*json*” si au fost folosite in felul urmator:

public MailManager(IConfiguration configuration)

{

\_smtpServer = "smtp.gmail.com";

\_smtpPort = 587;

\_smtpUser = configuration["Smtp:Email"];

\_smtpPass = configuration["Smtp:Password"];

}

O imagine care conține text, Font, captură de ecran, algebră

Conținutul generat de inteligența artificială poate fi incorect.Fig XX Email Invitatie

O imagine care conține text, captură de ecran, număr, Font

Conținutul generat de inteligența artificială poate fi incorect.

Fig XX Formular Register

**2. Autentificarea utilizatorului (Login)**

La logare, utilizatorul introduce adresa de email și parola. Parola introdusă este trecută prin același proces de hash, folosind salt-ul asociat utilizatorului stocat în baza de date. Dacă rezultatul hash-ului coincide cu cel stocat, autentificarea este considerată validă, iar utilizatorul primește accesul permis de rolul său.

O imagine care conține text, captură de ecran, Font, număr

Conținutul generat de inteligența artificială poate fi incorect.ASP.NET Core Identity folosește sesiuni sau token-uri de autentificare (de exemplu cookie-uri securizate sau JWT), care nu expun niciodată date sensibile și sunt protejate împotriva atacurilor de tip CSRF, XSS sau man-in-the-middle. [19]

Fig XX – Formular Login

3. Certificatul digital generat de ASP.NET și autentificarea sigură

Pentru a asigura o comunicare sigură între client și server, aplicația folosește un certificat digital (SSL/TLS), generat și configurat fie automat de ASP.NET în mediul de dezvoltare (self-signed), fie emis de o autoritate de certificare pentru producție. Acest certificat asigură criptarea tuturor datelor transmise între client și server, astfel încât parolele, token-urile de autentificare și orice alte date sensibile să nu poată fi interceptate de terți. [20]

În mediul ASP.NET, certificatul digital este folosit în principal la nivelul serverului web (IIS) pentru configurarea HTTPS, dar poate fi folosit și pentru semnarea token-urilor JWT (de exemplu, folosind certificate X.509 pentru autentificarea și validarea semnăturii). Acest lucru adaugă un nivel suplimentar de securitate, garantând că datele transmise și identitatea serverului nu pot fi falsificate sau compromise.

O imagine care conține text, captură de ecran, Font

Conținutul generat de inteligența artificială poate fi incorect.

Fig. XX Certificatul SSL asociat aplicației [20]

**3.3 Utilizarea pattern-ului Repository**

În cadrul aplicației, patternul Repository are un rol crucial în gestionarea accesului la datele persistente, separând logica de acces la baza de date de logica de business. Acest pattern este recomandat în arhitectura aplicațiilor ASP.NET Core, oferind o interfață abstractă între nivelul de business și nivelul de stocare a datelor, ceea ce facilitează întreținerea, testarea și extinderea sistemului. [14]

1. Ce este un repository?

Un repository este o clasă sau componentă care oferă metode pentru operații CRUD (Create, Read, Update, Delete) asupra entităților din sistem, ascunzând detaliile specifice ale accesului la baza de date (de exemplu, SQL, Entity Framework, API-uri externe). Acesta funcționează ca un „intermediar” între aplicație și sursa de date, permițând dezvoltatorului să interacționeze cu datele prin metode clare și reutilizabile, fără a depinde de implementarea specifică a stocării.

1. La ce sunt folosite repository-urile în proiect?

În acest proiect, repository-urile sunt utilizate pentru a centraliza logica de acces la date pentru fiecare tip de entitate principală, cum ar fi utilizatorii, notele, disciplinele sau cererile. De exemplu, există un IStudentRepository pentru operațiuni legate de studenți și un IProfesorRepository pentru profesori. Aceste repository-uri definesc interfețe și implementări specifice prin care aplicația poate adăuga, edita, șterge sau citi datele necesare, fără a depinde direct de infrastructura de stocare (precum Entity Framework sau SQL).

În practică, pattern-ul Repository este implementat astfel:

* Definirea interfețelor repository-ului pentru fiecare entitate

Pentru fiecare entitate principală, există o interfață specifică (de exemplu, IStudentRepository, ITeacherRepository, IsubjectRepository etc.). Aceste interfețe definesc metodele esențiale pentru operațiile CRUD, precum și metode suplimentare, adaptate nevoilor aplicației. De exemplu, IStudentRepository poate include metode precum GetStudentById, GetStudentsByGrupa sau GetStudentsByName, iar INotaRepository poate avea metode pentru calcularea mediei unui student sau pentru extragerea notelor aferente unei discipline.

* Implementarea repository-urilor concrete

Fiecare interfață are o implementare concretă (de exemplu, StudentRepository, ProfesorRepository etc.), care conține codul de interacțiune cu baza de date, utilizând Entity Framework Core. În aceste clase, operațiile definite în interfețe sunt traduse în interogări LINQ sau apeluri directe la contextul bazei de date (DbContext). De exemplu, metoda GetStudentsByGrupa din StudentRepository va efectua o interogare pentru a returna toți studenții dintr-o anumită grupă.

* Injectarea repository-urilor prin Dependency Injection [21]

Repository-urile sunt înregistrate în containerul de servicii al aplicației ASP.NET Core (în fișierul Program. cs) folosind metoda services. AddScoped() și sunt injectate automat în controlere sau servicii atunci când este necesar.

AddScoped:

* Durată de viață: O instanță este creată o singură dată per request HTTP.
* Utilizare tipică: Servicii care trebuie să păstreze date pe durata unei cereri, cum ar fi un context de bază de date (DbContext).

AddTransient:

* Durată de viață: O instanță nouă este creată de fiecare dată când serviciul este solicitat.
* Utilizare tipică: Servicii ușoare, fără stare (stateless), cum ar fi validatori, helperi etc.

 AddSingleton:

* Durată de viață: O singură instanță este creată pentru întreaga durată de viață a aplicației.
* Utilizare tipică: Servicii care nu depind de contextul cererii și care pot fi partajate în siguranță, cum ar fi cache-uri, configurări, loggere.

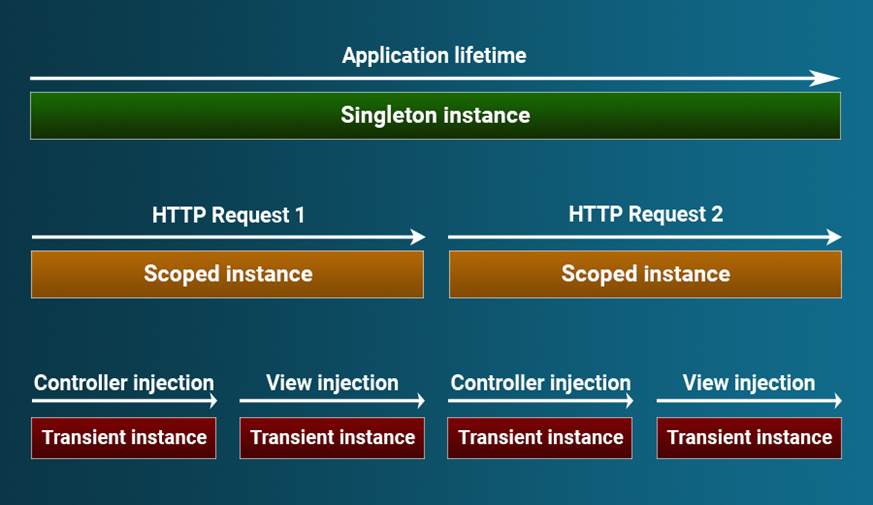


Fig XX Diagrama Dependency Inejction

Astfel, în exemplul StudentController, constructorul primește ca parametru un IStudentRepository, iar operațiile asupra studenților se desfășoară exclusiv prin intermediul acestui repository, fără acces direct la baza de date.

* Utilizarea repository-urilor în logica de business și în controlere

În controllere sau servicii de business, repository-urile sunt utilizate pentru a efectua operații asupra datelor. De exemplu, atunci când un administrator dorește să adauge un student nou, metoda din controller va apela IStudentRepository. Add(student). Pentru afișarea situației școlare a unui student, controllerul va utiliza metode din INotaRepository pentru a extrage și calcula notele relevante. Această abordare asigură consistență, reutilizare și o logică clară de acces la date.

* Avantajul testării și mentenanței

Prin utilizarea interfețelor, repository-urile pot fi ușor înlocuite cu implementări false (mock-uri) în testele unitare, permițând simularea diverselor scenarii fără a depinde de o bază de date reală. De asemenea, orice modificare a logicii de acces la date (de exemplu, schimbarea ORM-ului sau a sursei de date) se va realiza doar la nivelul repository-ului concret, fără a afecta restul aplicației.

**3.3.1 Modul de functionare a pattern-ului Repository**

În dezvoltarea aplicațiilor cu ASP.NET Core, accesul la date este esențial pentru orice sistem informatizat. O provocare semnificativă este separarea logicii de business de detaliile persistării datelor, pentru a asigura extensibilitatea, întreținerea și testarea ușoară a aplicației. În acest context, pattern-ul Repository oferă o soluție structurată și elegantă, abstractizând interacțiunea cu baza de date prin interfețe și clase dedicate fiecărei entități principale. Vom ilustra modul de funcționare pentru utilizatorul Student, având o funcționalitate similară cu celelalte componente.

1. Interfața IStudentRepository

public interface ISubjectRepository

{

Task<List<SubjectViewModel>> GetAllViewModelsAsync();

Task<Subject?> GetByCodeAsync(string subjectCode);

Task<bool> ExistsAsync(string subjectCode);

Task AddAsync(Subject subject);

Task SaveChangesAsync();

}

Interfața definește contractul obligatoriu pentru orice implementare de repository pentru Student:

* Metode pentru citirea tuturor studenților sau a unui student după id.
* Metodă pentru adăugarea unui nou student (cu returnarea succesului operației).
* Metodă pentru salvarea modificărilor.

2. Clasa concretă StudentRepository

namespace StudentPortal.Repositories

{

public class StudentRepository : IStudentRepository

{

private readonly ApplicationDbContext \_context;

public StudentRepository(ApplicationDbContext context) => \_context = context;

public async Task<Student?> GetByIdAsync(int studentId)

=> await \_context.Students.FirstOrDefaultAsync(s => s.StudentId == studentId);

public async Task<List<Student>> GetAllAsync()

=> await \_context.Students.ToListAsync();

public async Task<bool> AddAsync(Student student)

{

await \_context.Students.AddAsync(student);

return await \_context.SaveChangesAsync() > 0;

}

public async Task SaveChangesAsync()

{

await \_context.SaveChangesAsync();

}

public async Task<IEnumerable<Student>> GetAllStudentsAsync()

=> await \_context.Students.ToListAsync();

public async Task<Student?> GetStudentByIdAsync(int id)

=> await \_context.Students.FirstOrDefaultAsync(s => s.StudentId == id);

}

}

* Clasa implementează interfața IStudentRepository și folosește ApplicationDbContext pentru a accesa baza de date.
* GetAllStudentsAsync / GetAllAsync: Returnează toți studenții
* GetStudentByIdAsync / GetByIdAsync: Caută un student după ID
* AddAsync: Adaugă un student nou și salvează modificarea imediat, returnând dacă operația a fost cu succes.
* SaveChangesAsync: Permite salvarea modificărilor la un moment dat
  1. **Modul de funcționare a pattern-ului UnitOfWork**

În arhitectura unei aplicații bine structurate, Unit of Work reprezintă un model de proiectare destinat coordonării tuturor modificărilor aduse entităților într-o sesiune de lucru, tratându-le ca parte a unei singure tranzacții.

Fiecare operațiune de business care implică interacțiuni cu multiple repository-uri – de exemplu, salvarea simultană a unui student și a cererilor sale – se desfășoară în cadrul unei sesiuni Unit of Work. Toate repository-urile implicate în această operațiune utilizează același context de date, de obicei un DbContext în cazul utilizării Entity Framework. Acest context funcționează ca un container care monitorizează toate entitățile modificate, adăugate sau șterse pe parcursul execuției. [15]

Un aspect semnificativ este că modificările asupra entităților nu sunt aplicate imediat în baza de date. Acestea sunt înregistrate în contextul de date, care le monitorizează și le pregătește pentru persistență. Această abordare permite realizarea mai multor operațiuni asupra datelor într-un mod controlat, evitând interacțiunea directă cu baza de date la fiecare pas.

La finalul operațiunii, se invocă metoda SaveChanges() a Unit of Work-ului, care trimite toate modificările acumulate către baza de date, într-o singură tranzacție. Astfel, toate acțiunile sunt aplicate simultan, asigurând consistența datelor. În cazul apariției unei erori în timpul acestei operațiuni – fie de validare, fie din cauza unei probleme de conectivitate sau a unei constrângeri de integritate – întreaga tranzacție poate fi anulată (rollback). Acest mecanism previne situațiile în care doar o parte din modificări sunt salvate, evitând astfel inconsistențele sau corupția datelor.

O imagine care conține text, captură de ecran, Font, număr

Conținutul generat de inteligența artificială poate fi incorect.

Fig XX Diagrama functionalitatii pattern-ului Unit of Work [22]

**2.4.1 Implementarea UnitOfWork in proiect**

În cadrul acestui proiect, pattern-ul UnitOfWork a fost implementat pentru a gestiona eficient operațiile complexe care implică mai multe entități, cum ar fi înscrierea unui student la mai multe discipline, actualizarea simultană a notelor și a statutului academic, sau procesarea cererilor administrative.

• Definirea interfeței IUnitOfWork

A fost creată o interfață IUnitOfWork care expune toate repository-urile relevante (ex: IStudentRepository, IProfesorRepository, INotaRepository etc.) și o metodă Task SaveAsync() pentru salvarea tranzacțiilor în baza de date.

• Implementarea clasei UnitOfWork

Clasa UnitOfWork implementează interfața și primește ca parametru un context de date (ApplicationDbContext). Toate repository-urile sunt instantiate cu acest context, asigurându-se astfel că modificările asupra entităților sunt urmărite global. Metoda SaveAsync() apelează context. SaveChangesAsync(), persistând toate schimbările efectuate de repository-uri într-o singură tranzacție.

public async Task<IDbContextTransaction> BeginTransactionAsync()

=> await \_dbContext.Database.BeginTransactionAsync();

public async Task<int> SaveChangesAsync()

=> await \_dbContext.SaveChangesAsync();

public async Task CommitAsync()

{

var transaction = \_dbContext.Database.CurrentTransaction;

if (transaction != null)

{

await transaction.CommitAsync();

}

}

public async Task RollbackAsync()

{

var transaction = \_dbContext.Database.CurrentTransaction;

if (transaction != null)

{

await transaction.RollbackAsync();

}

}

}

• Injectarea UnitOfWork-ului în servicii/controlere

UnitOfWork este înregistrat în containerul de servicii al aplicației și este utilizat în serviciile sau controlerele care necesită coordonarea operațiunilor asupra mai multor tipuri de date. De exemplu, atunci când o cerere administrativă implică modificări la studenți, note și statut academic, toate aceste operații sunt efectuate prin intermediul UnitOfWork-ului, iar la final se realizează o singură salvare a datelor.

Exemplu de utilizare

* Se adaugă studentul prin unitOfWork.Students.Add(student);
* Se adaugă notele prin unitOfWork.Notes.Add(note1); ...
* La final, se apelează await unitOfWork.SaveAsync(); Astfel, dacă intervine o eroare, niciuna dintre operații nu este salvată parțial, iar datele rămân consistente.
  + 1. **Avantajele pattern-ului UnitOfWork**

Pattern-ul Unit of Work oferă importante avantaje în dezvoltarea aplicațiilor ce implică operațiuni complexe asupra datelor. În primul rând, contribuie la menținerea consistenței și integrității datelor, având în vedere că toate modificările sunt gestionate ca parte a unei singure tranzacții. Astfel, în cazul unei erori în timpul procesului de salvare, întregul set de operațiuni poate fi anulat (rollback), prevenind salvarea parțială a datelor sau inconsistențele. Acest model simplifică de asemenea logica serviciilor și a controlerelor, centralizând procesul de salvare a modificărilor și reducând redundanța codului. Prin separarea clară a responsabilităților și abstractizarea accesului la date, Unit of Work facilitează testarea și extinderea aplicației, contribuind la întreținerea și evoluția acesteia pe termen lung.

**3. Fluxuri de lucru (importul “*bulk*” de date)**

În cadrul aplicației, importul „bulk” (în masă) permite popularea rapidă a bazei de date cu entități precum studenți, departamente, orare sau subiecte, utilizând un fișier Excel. Acest flux de lucru este deosebit de util la începutul anului universitar sau când se preiau date din sisteme externe. Procesul incepe cu pregătirea fișierului Excel, conform unui șablon standardizat furnizat de aplicație. Administratorul completează fișierul cu informațiile necesare, cum ar fi datele studenților (nume, email, grupă), structura departamentelor, orarele sau lista de subiecte.

O imagine care conține text, captură de ecran, număr, Font

Conținutul generat de inteligența artificială poate fi incorect.Fișierul este încărcat printr-un formular din aplicație și ajunge la backend, unde este procesat cu ajutorul unei biblioteci specializate (de exemplu, EPPlus). Datele sunt extrase și validate pentru fiecare rând din fișier (de exemplu, verificarea formatului email-ului sau evitarea duplicatelor). Dacă datele respectă criteriile de validare, acestea sunt mapate pe entitățile interne și persistate în baza de date, de obicei prin intermediul pattern-urilor Repository și UnitOfWork pentru a asigura tranzacționalitatea. La final, administratorul primește un raport privind rezultatul importului: numărul de rânduri importate cu succes și eventualele erori întâmpinate.

Fig XX Importul de orar

La importul de orar, sunt necesare urmatoarele campuri: *DepartmentCode, SubjectCode, IsLab (1 = True, 0 = False), Teacher, Weekday, StartTime, EndTime, Specialization, Year, Semester.*

O imagine care conține text, captură de ecran, software, Font

Conținutul generat de inteligența artificială poate fi incorect.Fig XX Rezultat import orar

In cazul in care informatiile sunt introduse eronat sau exista deja (fie in fisierul de import, fie in baza de date) vor fi returnate erori corespunzatoare. De asemenea se verifica si existenta codului de departament, a codului de materie si a profesorului.

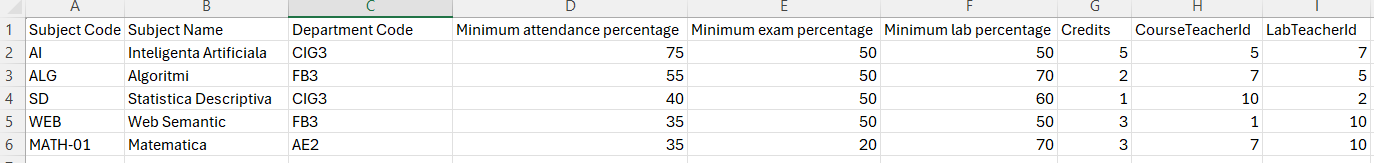


Fig XX Importul de subiecte (materii)

La importul de subiecte, sunt necesare urmatoarele campuri: *SubjectCode, SubjectName, DepartmentCode, Minimun Attendace Percentage, Minimum Exam Percentage, Minimum Lab Percentage, Credits, CourseTeacherId si LabTeacherId.*

O imagine care conține text, captură de ecran, număr, software

Conținutul generat de inteligența artificială poate fi incorect.

Fig XX Rezultat import subiecte

O imagine care conține text, captură de ecran, Font, număr

Conținutul generat de inteligența artificială poate fi incorect. Daca informatiile introduse in fisierul excel sunt valide, se vor introduce cu succes. In cazul in care acestea se repeta sau nu sunt valide (fie in fisierul de import, fie in baza de date) vor fi returnate erori si se va trece la urmatoarea linie.

Fig XX Importul de departamente

La importul de subiecte, sunt necesare urmatoarele campuri: *Department Code, Department Name, Department Head, Department Phone Number.*

O imagine care conține text, captură de ecran, număr, software

Conținutul generat de inteligența artificială poate fi incorect.

O imagine care conține text, captură de ecran, Font, număr

Conținutul generat de inteligența artificială poate fi incorect.Fig XX Rezultat import departmente

Fig XX Importul de utilizatori

O imagine care conține text, captură de ecran, software, număr

Conținutul generat de inteligența artificială poate fi incorect.

Fig XX Rezultat import utilizatori

(numele utilizatorilor invitati va aparea ca si “*temp*” pana la completarea procesului de inregistrare)

**4.1 Adăugarea și modificarea manuală a datelor**

Pe lângă funcționalitatea de import „bulk” din fișiere Excel, aplicația permite și gestionarea manuală a datelor pentru fiecare entitate principală. Acest lucru este util atunci când trebuie adăugat rapid un singur subiect (disciplină), când se corectează o eroare sau când se actualizează anumite informații fără a recurge la importuri automate.

Ca un exemplu se va folosi entitatea “*Subject*”. Interfața aplicației oferă un formular dedicat pentru introducerea manuală a unui nou subiect. Formularul cuprinde câmpuri precum denumirea subiectului, codul acestuia, numărul de credite, departamentul asociat și alte informații relevante. După completare, datele sunt validate atât la nivel de interfață, cât și la backend. Dacă totul este corect, subiectul este salvat în baza de date cu ajutorul repository-ului dedicat (ISubjectRepository).

O imagine care conține text, captură de ecran, număr, Font

Conținutul generat de inteligența artificială poate fi incorect.

Fig. XX Adaugarea manuala in aplicatie

Adaugarea manuala poate fi realizata cu succes doar in cazul in care informatiile introduse (in cazul de fata cele legate de “*DepartmentCode*” , “*CourseTeacherId*” si “*LabTeacherId*”) sunt valide si exista in baza de date. In caz contrar, la actionarea butonului “*AddSubject*” aplicatia va returna eroare si inserarea nu va fi realizata.

O imagine care conține text, Font, linie, număr

Conținutul generat de inteligența artificială poate fi incorect.

Fig. XX Rezultatul inserarii manuale

1. **Proiectarea bazei de date**

Pentru realizarea schemei bazei de date, a fost convertita structura bazei de date din proiect, din format C#, in format DBML (Database Markup Language). DBML (este un limbaj simplu, bazat pe text, folosit pentru a descrie structura bazelor de date într-un mod ușor de citit și scris. Este adesea utilizat pentru a genera diagrame ER (Entity-Relationship). Unul dintre principalele avantaje ale DBML este lizibilitatea sa. Sintaxa intuitivă permite modelarea tabelelor, specificarea coloanelor (tipurile de date, constrângerile ca și cheile primare și cele străine) și definirea relațiilor într-un mod accesibil, atât pentru dezvoltatori, cât și pentru echipele de proiectare. În plus, există instrumente moderne precum “*dbdiagram.io*” care transformă aceste definiții într-o diagramă vizuală, facilitând astfel colaborarea și comunicarea în cadrul proiectelor. [26]

Caracteristici principale ale DBML:

* Sintaxă simplă și intuitivă – similară cu limbajele de programare moderne.
* Flexibilitate – fiind independent de platformele specifice de baze de date, această caracteristică îl face o alegere potrivită pentru proiecte cu diverse cerințe tehnice, asigurând o documentație clară și ușor de adaptat
* Portabil – poate fi convertit în SQL sau în alte formate de modelare a bazelor de date.
* Suport pentru relații – permite definirea relațiilor între tabele (ex: has many, belongs to).

O imagine care conține text, captură de ecran, număr

Conținutul generat de inteligența artificială poate fi incorect.

Fix XX Diagrama bazei de date [23]

| **Relație** | **Tip** | **Explicație** |
| --- | --- | --- |
| User ↔ Student | 1:1 | Fiecare student este și un utilizator |
| User ↔ Teacher | 0..1:1 | Un profesor este și un utilizator |
| Subject ↔ Teacher | M:1 | Un profesor poate preda mai multe materii |
| Student ↔ Subject (prin Situation) | M:N | Un student are situații pentru mai multe materii |
| Subject ↔ Department | M:1 | O materie aparține unui singur departament |
| TimeTable ↔ Subject | M:1 | Orarul este făcut pentru o materie |
| TimeTable ↔ Teacher | M:1 | Cursul/laboratorul este susținut de un profesor |
| TimeTable ↔ Department | M:1 | Orarul este asociat cu un departament |

Fig XX Explicarea relatiilor diagramei relationale

Table TimeTable {

DepartmentCode varchar(50) [not null, ref: > Department.DepartmentCode]

Year int [not null]

Semester int [not null]

SubjectCode varchar(50) [not null, ref: > Subject.SubjectCode]

Weekday varchar(100) [not null]

StartTime time [not null]

EndTime time [not null]

IsLab boolean [not null]

LabTeacherId int [ref: > Teacher.TeacherId]

CourseTeacherId int [ref: > Teacher.TeacherId]

Specialization varchar [not null]

Indexes {

(DepartmentCode, Year, Semester, SubjectCode) [pk]

}

}

* 1. Definitii

[not null, ref: > Department.DepartmentCode]

* “*not* *null*” reprezinta faptul ca valoarea este obligatorie
* “*ref: > Department.DepartmentCode*” campul se refera la *DepartmentCode* din tabela *Department*

Indexes {

(DepartmentCode, Year, Semester, SubjectCode) [pk]

}

* Un index cu proprietatea [pk] reprezinta cheia primara compusa din tabela respectiva
  1. Utilizatorul – structura utilizatorilor aplicatiei

Cel mai important element al sistemului este tabelul *User*, care conține informații fundamentale despre toți utilizatorii aplicației: nume, prenume, adresă de email, adresă, oraș, județ și CNP. Acest tabel servește drept "super-entitate", din care derivă cele două roluri: student și profesor. Astfel, un student sau un profesor trebuie să fie precedat de existența sa ca *User*.

Tabelul Student este corelativ *1-to-1* cu *User* prin cheia *UserId*. Fiecare student este un utilizator înregistrat în sistem, având atribute suplimentare precum *RegistrationNumber*, Specialization, *Year*, *Semester* și *Group*. De asemenea, este inclus un câmp *RegisteredOn*, care indică data înregistrării studentului.

Similar, tabelul *Teacher* se leagă de *User* tot printr-o relație *1-to-1*, cu o mențiune: nu toți utilizatorii sunt profesori, astfel că relația este specificată ca *0. . 1:1*. Așadar, un utilizator poate să nu fie profesor, dar un profesor trebuie să fie definit ca User. Profesorii beneficiază, de asemenea, de un câmp *RegisteredOn*.

* 1. Departamentele și materia – structura academică

Fiecare materie (*Subject*) aparține unui departament, iar acest lucru este reprezentat prin câmpul *DepartmentCode* din tabelul *Subject*. Departamentele sunt entități independente, definite în tabelul *Department*, care conține un cod, un nume, un număr de telefon și o referință către șeful de departament (*DepartmentHeadId*), care este un *Teacher*. Așadar, fiecare departament este condus de un profesor.

Materia (*Subject*) este definită printr-un cod, un nume, numărul de credite, dar și câteva praguri minime (pentru prezență, examen și proiect). În plus, fiecare materie are atribuit un profesor de curs (*CourseTeacherId*) și, dacă este cazul, un profesor de laborator (*LabTeacherId*).

* 1. Situația studentului – legătura dintre student și materie

Relația dintre studenți și materii este gestionată printr-un tabel intermediar numit *Situation*. Aici vedem cum se construiește o relație many-to-many: un student poate urma mai multe materii, iar o materie este parcursă de mai mulți studenți. În tabelul *Situation* sunt înregistrate și informații legate de performanța studentului: procentul de prezență (*AttendancePercentage*), punctajul la examen și proiect, și nota finală (*FinalGrade*). De asemenea, este stocat și ID-ul profesorului care a evaluat studentul (*TeacherId*), ceea ce înseamnă că sistemul poate urmări cine a notat pe cine.

* 1. Orarul – când se țin materiile

Planificarea activităților didactice este gestionată prin tabelul *TimeTable*. Acesta conține informații despre ce materii se țin, când, în ce zi a săptămânii (*Weekday*), în ce interval (*StartTime* – *EndTime*), dar și dacă este vorba despre un laborator (*IsLab*). Tot aici se specifică și specializarea, anul și semestrul pentru care orarul este valabil.

Pentru fiecare înregistrare în orar, este precizat:

* codul departamentului (*DepartmentCode*)
* codul materiei (*SubjectCode*)
* profesorul responsabil de curs (*CourseTeacherId*)
* profesorul responsabil de laborator (*LabTeacherId*)

Bibliografie

[1] - <https://en.wikipedia.org/wiki/ASP.NET>

[2] - <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity-custom-storage-providers?view=aspnetcore-8.0>

[3] - <https://learn.microsoft.com/en-us/archive/msdn-magazine/2012/january/asp-net-security-securing-your-asp-net-applications>

[4] - <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-9.0&tabs=visual-studio>

[5] - <https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022>

[6] - <https://www.simplilearn.com/tutorials/asp-dot-net-tutorial/what-is-asp-dot-net-mvc>

[7] - <https://medium.com/@ravipatel.it/exploring-net-8-and-asp-net-core-features-advantages-and-comparison-with-asp-net-framework-a6b5743fc7be>

[8] - <https://www.geeksforgeeks.org/c-sharp/introduction-to-c-sharp/>

[9] - <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios>

[10] - <https://learn.microsoft.com/en-us/dotnet/csharp/linq/>

[11] - <https://www.yogihosting.com/aspnet-core-identity-roles/>

[12] - <https://code-maze.com/csharp-hashing-salting-passwords-best-practices/>

[13] - <https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff649690(v=pandp.10)?redirectedfrom=MSDN>

[14] - <https://deviq.com/design-patterns/repository-pattern>

[15] - <https://dotnettutorials.net/lesson/unit-of-work-csharp-mvc/>

[16] - <https://chatgpt.com/>

[17] - <https://www.nexusgroup.com/what-is-a-salt-hash/>

[18] - <https://learn.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-8.0&tabs=windows>

[19] - <https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-9.0>

[20] - <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/certauth?view=aspnetcore-9.0>

[21] - <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-8.0>

[22] - <https://www.c-sharpcorner.com/UploadFile/b1df45/unit-of-work-in-repository-pattern/>

[23] - <https://dbml.dbdiagram.io/home>

[24] - <https://logcorner.com/how-to-customize-asp-net-identity-core-with-external-database-storage-step-by-step/>

[25] - <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-implementation-entity-framework-core>

[26] - <https://foliant-docs.github.io/docs/tutorials/db/dbml/>