UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU

FACULTATEA DE INGINERIE

DEPARTAMENTUL DE CALCULATOARE ŞI INGINERIE ELECTRICĂ

PROIECT DE DIPLOMĂ

Conducător ştiinţific : Prof. Dr. Ing. Florea Adrian

Îndrumător: Prof. Dr. Ing. Florea Adrian

Absolvent: Gîrjoabă Constantin

Specializarea: Calculatoare

- Sibiu, 2023 -

UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU

FACULTATEA DE INGINERIE

DEPARTAMENTUL DE CALCULATOARE ŞI INGINERIE ELECTRICĂ

SOFTWARE EDUCAȚIONAL PENTRU PROFESORI ȘI ELEVI

Conducător ştiinţific: Prof. Dr. Ing. Florea Adrian

Îndrumător: Prof. Dr. Ing. Florea Adrian

Absolvent: Gîrjoabă Constantin

Specializarea: Calculatoare

- Sibiu, 2023 -

Cuprins:

1 Introducere

1.1 Prezentarea temei

Pentru proiectul de diplomă am ales să creez un software educațional pentru profesori și studenți, care să fie utilizat, în faza inițială, doar in domeniul matematicii. Acesta oferă un set de caracteristici și instrumente pentru a îmbunătăți experiența de predare și învățare a matematicii. Folosind acest software, profesorii își pot gestiona eficient orele suplimentare petrecute în scopuri didactice, pot crea teste personalizate, pot urmări progresul elevilor și pot oferi feedback personalizat. Elevii pot susține teste online, își pot revizui notele și pot rămâne organizați cu evaluările viitoare. Aplicația este concepută ca o aplicație WEB pentru a putea fi accesată și de la distanță.

1.2 Motivul alegerii temei

Principalul motiv pentru care am ales tema a fost faptul că ambii părinți sunt profesori de matematică și petrec foarte mult timp pregătind si corectând teste pentru elevi. Prin generarea rapidă de teste personalizate și corectarea automată(sau manuală dar semnificativ mai rapidă) sper să reduc timpul necesar și efortul mental depus.

1.3 Necesitate

Astfel de aplicații, care permit susținerea testelor online, au fost foarte utile pe timpul pandemiei și cu siguranță vor fi și pe viitor. Datorită avansului tehnologic și a digitalizării în toate domeniile, nici învățământul nu trebuie să rămână în urmă. Această aplicație va oferi posibilitatea studenților de a susține testele de oriunde s-ar afla, în cazuri excepționale și cu acordul profesorului. Mai mult, anumite componente ale aplicației pot fi customizate și în procese de training și evaluare din companii.

1.4 Obiective

Principalul obiectiv este acela de a reduce timpul necesar unui profesor pentru a evalua un student. Pentru îndeplinirea acestuia, aplicația va crea următoarele posibilități:

- Generarea automată de teste personalizate
- Corectarea testelor de către aplicație (în cazul în care testul conține doar întrebări de tip „grilă”) sau corectarea asistată de către profesor
- Actualizarea notelor în urma corectării testelor
- Adăugarea notelor (altele decât cele de la teste) de către profesori pentru a menține o evidență detaliată asupra notelor studenților.
- Ținerea evidenței referitoare la prezențele studenților
- Susținerea testelor online de către studenți

2 Considerații teoretice

2.1 Mediu dezvoltare

Mediul de dezvoltare pe care l-am ales pentru realizarea aplicației este Visual Studio Code (VS Code) versiunea 1.79.1. Acesta este un mediu dezvoltat de cei de la Microsoft și folosit pe scară largă de programatori pentru dezvoltarea aplicațiilor în multe domenii și limbaje diferite. Printre principalele caracteristici ale VS Code se numără următoarele:

VS Code conține un editor de cod personalizabil care permite navigarea rapidă a codului, completarea automată, minimizarea codului, editarea mutliplă simultan. Prin adăugarea anumitor extensii pentru limbajele folosite, se introduce și IntelliSense care pe lângă cele menționate mai sus, aduce numeroase beneficii:

- Sugestii de completare a codului și informații in funcție de contextul actual.
- Sugestii pentru parametri atunci cand se apeleaza o metodă.
- Importuri automate atunci cand se detecteaza folosirea funcțiilor din clase sau module externe.
- Informații rapide despre variabile, funcții, clase, etc, atunci când punem cursorul desupra lor.
- Posibilitatea de a naviga direct la definiția sau multiplele implementări și apeluri de funcții fără a mai căuta prin toate fișierele proiectului.
- Organizarea codului in funcție de limbaj, IntelliSense va detecta limbajul folosit și se adaptează, respectând convențiile sale. Va sublinia in diferite moduri erorile de sintaxă și averitzările.
- Terminal integrat: Acesta permite programatorilor să ruleze scripturi și comenzi direct din mediul de dezvoltare eliminând necesitatea de a comuta între ferestre si eficientizând procesul de lucru.
- Debugging/Depanare integrată: VS Code conține depanatoare integrate pentru diverse limbaje de programare. Acestea permit adăugarea de breakpoint-uri, inspectarea variabilelor, rularea pas cu pas și analizarea comportamentului în timpul execuției, ajutând astfel la identificarea și rezolvarea problemelor.

- Suportă multiple limbaje de programare: Varianta inițială a mediului de programare suportă o gamă largă de limbaje și oferă posibilitatea de adăugare a extensiilor pentru a suplimenta limbajele suportate, permițând programatorilor să folosească limbajele preferate.
- Integrare Source Control: Sistemele populare de control al versiunilor, precum Git, sunt foarte ușor de integrat. Acestea permit gestionarea repository-urilor direct din mediul de dezvoltare oferind istoric al commit-urilor, gestionarea branch-urilor, evidențierea diferențelor dintre versiuni, etc.
- Suport Cross-Platform: VS Code funcționează fără probleme pe mai multe sisteme de operare, asigurând programatorilor o experiență asemănătoare pe diferite platforme.
- Extensibilitate: Programatorii pot personaliza si îmbunătăți mediul de dezvoltare prin instalarea de extensii care ulterior adaugă funcționalități suplimentare.

Extensiile instalate pentru realizarea acestui proiect sunt:

- C# for Visual Studio Code care include și un debugger de C#, permite personalizarea comportamentului editorului, precum completarea automată, importuri automate, etc., permite crearea de fișiere noi folosind template-uri de C#.
- Material Icon Theme care este doar pentru modificarea icoanelor din solution explorer.

2.2 Tehnologii și limbaje folosite

2.2.1 .NET

Microsoft .NET este o platformă de dezvoltare software robustă creată de Microsoft. Ea oferă dezvoltatorilor un cadru cuprinzător și instrumentele necesare pentru a construi o gamă largă de aplicații, inclusiv aplicații web, aplicații desktop, aplicații mobile și servicii bazate pe cloud.

În esență, .NET este un mediu de execuție gestionat care oferă un runtime numit Common Language Runtime (CLR). CLR oferă servicii precum gestionarea memoriei, securitatea și gestionarea excepțiilor, făcând dezvoltarea aplicațiilor mai eficientă și mai fiabilă. Include, de asemenea, un compilator just-in-time (JIT) care traduce codul Intermediate Language (IL) în cod mașină în timpul execuției.

.NET suportă mai multe limbaje de programare, inclusiv C#, Visual Basic .NET (VB.NET) și F#. Programatorii își pot alege limbajul preferat în funcție de familiaritatea și cerințele lor. Aceste limbaje sunt concepute pentru a funcționa perfect cu cadrul .NET, permițând dezvoltatorilor să folosească un set bogat de biblioteci și API-uri pentru a construi aplicații puternice și scalabile.

Unul dintre avantajele majore ale .NET este biblioteca sa extinsă de clase, cunoscută sub numele de .NET Framework Class Library (FCL). FCL oferă o colecție vastă de componente pre-construite, reutilizabile și API-uri pentru sarcini obișnuite, cum ar fi I/O pentru fișiere, rețelistică, acces la date și dezvoltarea interfeței cu utilizatorul. Această bibliotecă economisește timp și efort dezvoltatorilor, oferind soluții gata făcute pentru multe provocări comune de programare.

Pe lângă .NET Framework tradițional, Microsoft a introdus și .NET Core, care este o implementare cross-platform și open-source a .NET. .NET Core permite programatorilor să creeze aplicații care pot rula pe Windows, macOS și diverse variante Linux, extinzând acoperirea .NET la o gamă mai largă de platforme.

În plus, Microsoft a dezvoltat instrumente de dezvoltare puternice, cum ar fi Visual Studio și Visual Studio Code, care oferă medii de dezvoltare integrate (IDE) pentru crearea, depanarea și implementarea aplicațiilor .NET, acestea devenind astfel o alegere populara pentru construirea de aplicații scalabile în diferite domenii.

.NET CLI (Command-Line Interface) este un instrument oferit de cei de la Microsoft care permite programatorilor de aplicații si proiecte .NET să creeze, testeze, ruleze, construiască aplicații direct din linia de comandă. Acesta oferă un set de comenzi care simplifică sarcinile comune de dezvoltare și automatizează diverse aspecte ale fluxului de lucru. Printre caracteristicile esențiale ale .NET CLI se numără următoarele:

- Gestionarea proiectului, CLI oferă programatorilor posibilitatea de a crea proiecte noi folosind diverse șabloane, specificând tipul de proiect și alte opțiuni de configurare. De asemenea, facilitează modificare proiectului permițând actualizarea, adăugarea sau eliminarea dependențelor proiectului.
- Construirea si compilarea proiectului

- Managementul pachetelor, .NET CLI se integrează perfect cu managerul de pachete NuGet. Programtorii pot căuta cu ușurință pachete, le pot instala, actualiza și elimina ca dependențe de proiect. Oferă și funcționalitatea de restaurare a pachetelor, asigurându-se că proiectele au toate dependențele necesare disponibile.
- Rularea și testarea aplicațiilor: Simplifică executarea aplicațiilor .NET direct din linia de comandă. Se pot rula aplicații cu argumente și opțiuni de linie de comandă specificate, facilitând testarea și explorarea aplicațiilor.
- Integrarea ușoară în diferite editoare și IDE-uri precum VS Code sau Visual Studio

Cele mai utilizate comenzi .NET CLI sunt: dotnet new (folosită la crearea unui proiect nou bazat pe diferite template-uri), dotnet build (construiește și crează executabilul), dotnet run (rulează o aplicație .NET), dotnet test, dotnet add (adaugă un pachet într-un proiect), dotnet remove (elimină un pachet), dotnet list (afișează pachetele folosite într-un anumit proiect), dotnet sln add (adaugă un proiect într-o soluție), dotnet ef (pune la dispoziție comenzi pentru entity framework).

Proiectul .NET WebAPI este tipul de proiect pe care l-am folosit pentru API-ul aplicației. Acesta este un un tip de proiect din ecosistemul Microsoft .NET care este conceput special pentru construirea de servicii web și API-uri. El permite programatorilor să creeze API-uri RESTful care pot fi consumate de clienți, cum ar fi aplicații web, aplicații mobile și alte sisteme.

RESTful (Representational State Transfer) este un stil arhitectural pentru proiectarea aplicațiilor în rețea. Se bazează pe simplitatea, scalabilitatea și interoperabilitatea prin utilizarea metodelor standard HTTP și a URI-urilor (Uniform resource identifier) pentru a accesa și manipula resursele. Promovează comunicarea stateless și permite clienților să descopere și să interacționeze cu resurse prin link-uri hypermedia. Câteva dintre principiile de bază ale arhitecturii RESTful sunt:

- Bazat pe resurse: Resursele sunt identificate prin URI-uri unice (Uniform Resource Identifiers), iar clienții pot interacționa cu aceste resurse folosind metode HTTP standard pentru a efectua operații asupra lor.
- Stateless: Fiecare cerere de la client către server conține toate informațiile necesare, iar serverul nu stochează date specifice clientului între cereri.

- Se folosesc metodele HTTP standard (GET, POST, PUT, DELETE)
- Interfață uniformă: serviciile RESTful au o interfață uniformă care promovează simplitatea și ușurința în utilizare.
- Separarea client-server: clientul și serverul sunt entități separate care interacționează prin cereri și răspunsuri HTTP.
- Scalabilitate: Arhitecturile RESTful sunt proiectate pentru a fi scalabile, permițând gestionarea eficientă a unui număr mare de cereri concurente.
- Interoperabilitate: API-urile RESTful pot fi consumate de diverși clienți, inclusiv browsere web, aplicații mobile și alte servicii, făcându-le foarte interoperabile.

Proiectul .NET WebAPI utilizează capacitățile ASP.NET, un framework de dezvoltare web oferit de Microsoft, pentru a gestiona request-urile și response-urile HTTP, permițând o comunicare eficientă și scalabilă între clienți și servere.

Baza unui proiect .NET WebAPI este framework-ul ASP.NET Web API. Acesta simplifică procesul de construire a serviciilor bazate pe HTTP, oferind un set de caracteristici și abstracții care gestionează rutarea request-urilor, negocierea conținutului, serializarea și multe altele. Programatorii pot defini controllere care gestionează request-urile primite, procesează datele și returnează răspunsuri în diferite formate, cum ar fi JSON sau XML.

Un proiect .NET WebAPI urmează principiile arhitecturii REST și încurajează utilizarea metodelor HTTP standard (GET, POST, PUT, DELETE) pentru a efectua operații asupra resurselor și utilizează coduri de stare HTTP pentru a indica rezultatul request-urilor. Pe lângă acestea un proiect .NET WebAPI suportă și funcționalități pentru autentificare, autorizare și validare a datelor. Se integrează cu sistemul ASP.NET Identity pentru a oferi autentificarea utilizatorilor și controlul accesului bazat pe roluri.

Un proiect .NET WebAPI poate fi găzduit pe diverse platforme, cum ar fi Internet Information Services (IIS), Azure App Service sau auto-găzduit într-o aplicație personalizată.

O aplicație .NET WebAPI funcționează prin primirea request-urilor HTTP de la clienți și generând response-uri corespunzătoare. Urmează un model request-response, în care clienții fac request-uri

către anumite endpoint-uri definite de aplicația WebAPI, iar aplicația procesează acele request-uri și returnează response-urile corespunzătoare.

Cele mai importante componente ale unui astfel de proiect sunt:

- Controllere: Controllerele sunt în centrul unei aplicații .NET WebAPI. Acestea gestionează request-urile și generează response-uri. Controllerele sunt clase care derivă din clasele ApiController sau ControllerBase și conțin metode de acțiune, specifice fiecărui endpoint, adnotate cu atribute precum [HttpGet], [HttpPost], [HttpPut] sau [HttpDelete] pentru a specifica metoda HTTP la care răspund. Aceste metode de acțiune procesează request-urile primite, efectuează operațiunile necesare și returnează rezultatele ca response-uri HTTP.
- Rute: Rutele definesc tiparele URL care se mapează la anumite controllere și metode de acțiune. Configurația de rutare este de obicei definită în fișierul RouteConfig.cs sau folosind atribute precum [Route] direct în controller sau deasupra metodelor. Rutele determină modul în care request-urile primite sunt expediate către controllerul corespunzător și metoda de acțiune bazată pe URL și verbul HTTP.
- Modele: Modelele reprezintă structurile de date utilizate de aplicația WebAPI. Ele sunt de obicei definite ca clase care încapsulează datele trimise sau primite. Modelele pot include atribute de validare pentru a asigura integritatea și validitatea datelor. Aceste modele sunt utilizate ca parametri în metodele de acțiune pentru a lega și valida datele de request primite și pot fi, de asemenea, returnate ca răspunsuri.
- Serializare: Serializarea este procesul de conversie a obiectelor de date într-un format adecvat pentru transmitere, cum ar fi JSON sau XML. Framework-ul .NET WebAPI se ocupă de serializarea și deserializarea automată a obiectelor model pe baza tipului de conținut solicitat și a tipurilor de returnare specificate sau a parametrilor metodelor de acțiune.
- Filtre: Oferă o modalitate de a injecta logica de procesare suplimentară înainte sau după executarea acțiunilor din controllere. Filtrele sunt folosite pentru a implementa funcționalități precum autentificarea, autorizarea, gestionarea excepțiilor, afișare, stocarea în cache. Aplicațiile WebAPI acceptă diferite tipuri de filtre, inclusiv filtre de autorizare, filtre de excepții, filtre de acțiune. Aceste filtre sunt aplicate în diferite etape ale procesării

request-urilor și permit programatorilor să adauge o logică personalizată pentru a modifica sau valida request-urile și response-urile.

- Negocierea conținutului: Negocierea conținutului se referă la procesul de selectare a formatului de răspuns adecvat (cum ar fi JSON sau XML) pe baza formatului solicitat de client. Frameworkul .NET WebAPI realizează automat negocierea conținutului pe baza antetului „Accept” trimis de client în request. Poate gestiona o varietate de formate.
- Middleware: Componentele Middleware oferă funcționalitate suplimentară pipeline-ului aplicației. Ele pot efectua sarcini precum înregistrarea cererilor/răspunsurilor, compresiei, transformarea cererii/răspunsului, stocarea în cache și multe altele. Componentele middleware sunt configurate în fișierul Startup.cs folosind metoda app.UseMiddleware().
- Injectarea dependențelor: Aplicațiile .NET WebAPI utilizează adesea dependency injection pentru a gestiona dependențele și pentru a promova modularitatea și testabilitatea. Frameworkul oferă un container de injecție de dependență încorporat care permite programatorilor să înregistreze și să rezolve dependențele în cadrul aplicației.

Pipeline-ul de request-uri și middleware-ul sunt esențiale în procesarea request-urilor și response-urilor HTTP într-o aplicație WebAPI .NET. Mai mult componente middleware formează pipeline-ul de request-uri. Acesta reprezintă etapele pe care un request le parcurge atunci când ajunge in aplicație. Fiecare etapă are datoria de a efectua o singură sarcină sau modificare asupra request-ului. Pipeline-ul este gestionat de către runtime-ul ASP.NET care asigură execuția în ordinea potrivită. El are două părți, o parte de request-uri și una de response-uri.

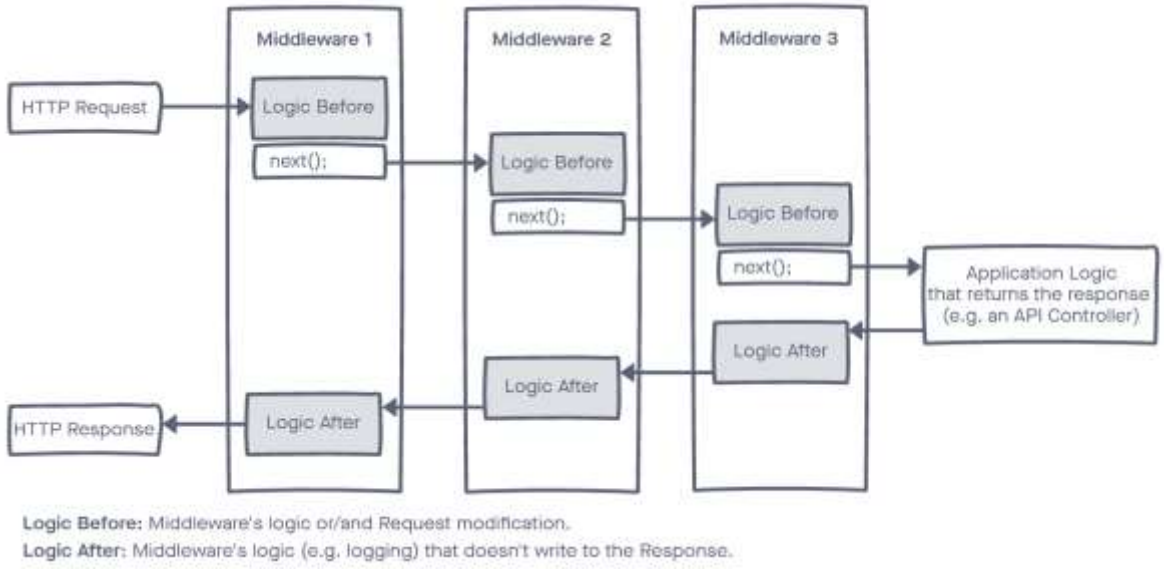

Figura 1 – Pipeline de solicitări și middleware. Preluată de la [6]

Componentele de middleware reprezintă etapele din pipeline. Metodele responsabile pentru tratarea request-ului și generarea response-ului sunt Invoke sau InokeAsync. Componentele de middleware pot fi adăugate, inlăturate sau reordonate, ordinea în care ele au fost adăugate în pipeline este ordinea în care ele se vor executa. Fiecare componentă are posibilitatea de a trimite request-ul mai departe la următoarea componentă sau de a returna un răspuns, astfel scurt-circuitând pipeline-ul. Practic un request ajunge în pipeline și trece prin componentele middleware, rând pe rând, acestea executând logica pe care o au de executat și returnând un response. Odată ce avem un response, acesta ajunge pe cealaltă parte a pipeline-ului, cea de response-uri și trece iar în ordine inversă prin toate middleware-urile, acestea având posibilitatea să analizeze sau modifice response-ul înainte de a îl trimite mai departe spre client.

Printre cele mai importante fișiere ale unui proiect de tipul WebAPI se află următoarele:

- Program.cs care este punctul de intrare în aplicație atunci când lansăm în execuție aplicația. Aici se crează o instanță de „WebApplicationBuilder” și se configură aceasta. Mai există secțiuni pentru adăugarea serviciilor(pentru autentificare, acces la baza de date sau entity framework, dependency injection, CORS(un mecanism care permite serverelor web să controleze și să activeze partajarea securizată a resurselor între origini, permițând aplicațiilor web să interacționeze în siguranță cu resurse din diferite domenii)),

configurarea pipeline-ului de request-uri, adăugarea de middleware și în final apelul funcției run care pornește aplicația.

- appsettings.json sau appsettings.Development.json este un fișier de configurare care stochează diverse setări specifice aplicației. Vă permite să definiți perechi cheie-valoare pentru a configura diferite aspecte ale aplicației, string-ul de conexiune la baza de date sau orice alte setări personalizate cerute de aplicație.
- launchSettings.json este utilizat pentru a configura setările pentru lansarea și depanarea aplicației. Oferă profiluri care specifică modul în care aplicația ar trebui să fie lansată în diferite medii (de exemplu, dezvoltare, producție). Fiecare profil conține informații precum adresa URL a aplicației, variabilele de mediu și argumentele liniei de comandă care urmează să fie utilizate în timpul depanării sau rulării aplicației.

2.2.2 Entity Framework

Entity Framework este un framework de mappare object-relational(ORM)(permite mapparea obiectelor din limbaj orientat obiect în baze de date relaționale) pentru .NET care simplifică programarea bazelor de date, oferind o abstractizare la nivel înalt asupra bazelor de date relaționale. Permite programatorilor să lucreze cu baze de date folosind un limbaj familiar orientat pe obiect și elimină nevoia de a scrie manual interogări SQL de nivel scăzut.

Entity Framework acceptă diferiți furnizori de baze de date, permițând programatorilor să lucreze cu diverse sisteme de baze de date, cum ar fi Microsoft SQL Server, Oracle, MySQL și SQLite, printre altele. Această flexibilitate îl face o alegere versatilă pentru aplicațiile care vizează diferite platforme de baze de date.

Pentru a integra Entity framework in VS Code, un pachet trebuie instalat în proiectul pe care îl dezvoltăm folosind comanda “dotnet add package Microsoft.EntityFrameworkCore”. Apoi trebuie sa avem clasele care vor reprezenta tabelele din baza de date definite și adnotate. Se crează o clasă care moștenește clasa DbContext și pentru fiecare clasă din model trebuie sa definim o proprietate de tipul DbSet. Ulterior folosim migrations pentru a crea și actualiza baza de date. Orice modificare asupra structurii bazei de date se face folosind migration. Acestea se crează, aplică, anulează din linia de comandă.

Câteva convenții folosite de Entity Framework:

- Convenția de denumire a entităților: implicit EF va presupune că numele entității va corespunde cu numele pe care îl vrem pentru tabel. În cazul în care nu respectăm această convenție, putem specifica noi numele tabelului folosind adnotația [Table(„numeTabel”)] deasupra clasei.
- Cheia primară: EF va presupune ca o proprietate numită Id sau care are sufixul Id este automat cheie primară
- Convenția de cheie străină
- Pluralizarea numelui tabelului: În cazul în care avem o clasă numită „Student”, acesta va numi tabelul(dacă nu specificăm noi un nume) „Students”
- Ștergerea cascadată: EF permite în mod convențional ștergerea în cascadă pentru relațiile în care entitățile dependente sunt șterse atunci când entitatea principală este ștearsă.

2.2.3 Angular

Angular este un framework open-source de front-end pentru aplicații web dezvoltat de Google. El permite programatorilor să creeze aplicații dinamice, receptive și bogate în funcții, cu o singură pagină (SPA) sau aplicații web progresive (PWA). Ca și limbaj de programare, Angular folosește TypeScript

Arhitectură bazată pe componente: Angular urmează o arhitectură bazată pe componente, în care aplicația este construită prin combinarea componentelor reutilizabile și modulare. Fiecare componentă reprezintă o parte specifică a interfeței cu utilizatorul, cum ar fi un antet, o bară laterală sau un formular, și își încapsulează propria logică, șabloane și stiluri.

Sintaxă bazată pe template-uri: Angular utilizează template-uri HTML declarative combinate cu directive de șablon puternice pentru a defini structura și comportamentul interfeței cu utilizatorul. Aceste șabloane permit:

- Markup asemănător HTML: șabloanele Angular folosesc o sintaxă asemănătoare HTML pentru a defini structura interfeței de utilizare. Scrieți etichete și atribute HTML familiare pentru a crea elemente precum div-uri, butoane și așa mai departe. Acest lucru face

șabloanele ușor de citit și de înțeles, în special pentru programatorii cu experiență anterioară în dezvoltarea web.

- Legarea datelor: Legarea datelor este o caracteristică fundamentală a sintaxei bazate pe șablon a Angular. Vă permite să stabiliți o conexiune între datele componentei (modelul) și elementele UI din șablon. Angular acceptă diferite tipuri de legare de date, inclusiv legarea unidirecțională (de la componentă la view sau invers) și legarea bidirecțională (sincronizarea bidirecțională a datelor între componentă și view).
- Interpolare: interpolarea este o modalitate de a insera dinamic valori în template. Folosind acolade ({{ }}), se pot include expresii, variabile sau proprietăți ale componentelor în codul template-ului. Aceste expresii sunt evaluate și înlocuite cu valorile corespunzătoare atunci când șablonul este afișat.
- Directive: Angular oferă directive încorporate care permit extinderea sintaxei HTML și adăugarea unui comportament suplimentar elementelor din șablon. Directivele sunt folosite pentru a efectua sarcini precum redarea condiționată a elementelor, bucla peste colecții, gestionarea evenimentelor, aplicarea dinamică a claselor CSS și multe altele. Exemple de directive Angular includ ngIf, ngFor, ngClass și ngModel.
- Legarea evenimentelor: legarea evenimentelor vă permite să răspundeți la interacțiunile utilizatorului (cum ar fi click-uri pe butoane, trimiteri de formulare sau mișcări ale mouse-ului) în cadrul șablonului. Folosind sintaxa de legare a evenimentelor (închisă în paranteze), puteți asocia un anumit eveniment cu o metodă sau funcție din componenta care se ocupă de eveniment.

Legare bidirecțională a datelor: Angular oferă legare bidirecțională a datelor, ceea ce înseamnă că modificările din model (date) sunt reflectate automat în vizualizare (UI) și invers. Acest lucru simplifică gestionarea datelor și reduce nevoia de actualizări manuale, îmbunătățind productivitatea dezvoltării.

Dependency injection: Angular are un sistem de dependency injection încorporat care gestionează crearea și partajarea de obiecte și servicii între diferite componente. Acest lucru promovează modularitatea codului, reutilizarea și testabilitatea.

Dezvoltare multiplatformă: Angular permite dezvoltatorilor să creeze aplicații care rulează pe mai multe platforme, inclusiv browsere web, desktopuri și dispozitive mobile.

Angular Routing: în Angular, rutarea este un mecanism care permite navigarea între diferite view-uri sau pagini dintr-o aplicație cu o singură pagină (SPA). Sistemul de rutare al Angular oferă o modalitate de a mapa URL-urile către anumite componente și de a controla fluxul de navigare al aplicației.

Router Module: rutarea Angular este gestionată prin RouterModule, care este un modul încorporat furnizat de pachetul @angular/router. RouterModule oferă funcționalitatea necesară pentru definirea rutelor, configurarea navigației și gestionarea modificărilor URL.

Configurare ruteleor: Configurarea rutelor se face în fișierul app-routing.module.ts (sau într-un fișier de modul similar). Acest fișier definește o serie de rute care mapează căile URL către clasele componente. Fiecare intrare de rută constă în mod obișnuit dintr-o cale, componentă și opțiuni de configurare opționale, cum ar fi parametri de rută, date sau guard-uri.

Router Outlet: componenta <router-outlet> este un placeholder în șablonul aplicației unde sunt redate componentele rutate. Când o rută este activată, Angular înlocuiește dinamic conținutul componentei router outlet cu șablonul componentei corespunzătoare.

Navigarea între rute: Angular oferă un serviciu de router care permite navigarea prin cod între rute. Există metode precum router.navigate() sau router.navigateByUrl() pentru a declanșa modificări de rută pe baza acțiunilor utilizatorului sau a logicii aplicației.

Route guards: Gărzile de rută oferă o modalitate de a controla accesul la rute în funcție de anumite condiții. Puteți folosi gărzi pentru a implementa autentificarea, autorizarea sau logica personalizată înainte de a permite navigarea către o anumită rută. Angular oferă mai multe tipuri de protecție, cum ar fi CanActivate, CanDeactivate, CanLoad și Resolve.

În Angular, efectuarea de request-uri HTTP pentru a prelua date de pe un server sau a trimite date către un server se face folosind modulul HttpClient încorporat. O scurtă descriere a modului în care funcționează cererile Angular:

- Importul modulului HttpClient: Pentru a utiliza modulul HttpClient, el trebuie importat din „@angular/common/http”
- Injectarea HttpClient: o instanță HttpClient trebuie injectată în componenta sau serviciul în care se vor face request-uri HTTP. Sistemul de injectare a dependenței de la Angular se ocupă de furnizarea automată a instanței HttpClient, el trebuie doar menționat ca parametru în constructor.
- Request-urile se fac folisind metodele http.get(), http.post(), etc. Care vor primi ca paremtrii adresa URL, datele pe care vrem sa le trimitem. Aceste metode returnează un Observable la care trebuie să ne „abonăm” (subscribe) pentru a accesa răspunsul.
- Gestionarea răspunsurilor: Când ne abonăm la un Observable returnat de metoda get() sau post(), putem defini funcții callback pentru a gestiona răspunsul. Aceste funcții sunt de obicei definite folosind metoda subscribe(). În cadrul funcțiilor callback, se pot accesa datele de răspuns, gestiona erorile și efectua operațiuni suplimentare pe baza răspunsului.
- Gestionarea erorilor: Pentru a gestiona erorile care pot apărea în timpul solicitării, se poate include o funcție de returnare a erorilor în cadrul metodei subscribe(). HttpClient de la Angular tratează automat cele mai comune scenarii de eroare, dar se poate personaliza gestionarea erorilor conform cerințelor fiecărei aplicației.
- Anteturi și parametri opționali: Se pot include, de asemenea, anteturi opționale sau parametri URL în request-uri oferind argumente suplimentare metodei get() sau post(). Anteturile pot fi setate folosind clasa HttpHeaders, iar parametrii URL pot fi transferați ca obiect.

Pentru aplicația client a fost nevoie și de Node.js, un JavaScript runtime environment. El este folosit în partea de build a aplicației pentru a compila automat codul TypeScript în JavaScript pentru a putea fi înteles de către browsere.

Node.js oferă si posibilitatea de a folosi Angular CLI și a executa comenzi direct din linia de comandă (ng serve, ng build, etc..). Pe lângă acestea, Node.js include și un packet manager (NPM – node packet manager) folosit de programatori pentru eficienta gestionare și instalare a altor module, librării și dependențe în proiectul Angular.

O altă funcționalitate importantă a framework-ului Angular o reprezintă formularele Angular, ele oferă o modalitate structurată și consecventă de a gestiona sarcinile legate de formulare, cum ar fi capturarea datelor utilizatorului, efectuarea validării și gestionarea trimiterii formularelor.

Există două categorii de formulare:

- Formulare bazate pe template-uri: Formularele bazate pe șabloane reprezintă o abordare mai simplă pentru construirea de formulare în Angular. Ele se bazează pe directive și pe sintaxa șablonului pentru a defini controalele de formular și pentru a gestiona logica legată de formular. Controalele de formular sunt create automat pe baza directivelor precum ngModel, care leagă câmpurile de formular cu proprietățile din clasa componente. Formularele bazate pe șabloane au o sintaxă mai simplă, deoarece cea mai mare parte a configurației formularelor este gestionată în cadrul șablonului însuși. Ele oferă o modalitate rapidă și ușoară de a configura formulare, în special pentru scenarii simple. În schimb, oferă mai puțin control și flexibilitate în comparație cu formele reactive. Accesarea și manipularea directă a controalelor de formular și a valorilor acestora din clasa de componente este mai limitată.
- Formulare reactive: Formele reactive oferă o abordare mai puternică și mai flexibilă pentru manipularea formularelor în Angular. Acestea urmează o abordare bazată pe date, în care controalele de formular sunt definite în mod explicit în clasa componentelor și legate de șablon. Formele reactive sunt create folosind clasele FormGroup și FormControl din modulul @angular/forms. Formularele reactive oferă control explicit asupra validării și trimiterii formularelor. Validatorii și logica de validare personalizată sunt definite în clasa de componente, permițând scenarii de validare mai complexe. Starea și valabilitatea formularului pot fi accesate și manipulate programatic. Ele utilizează puterea datelor de tip Observable pentru a urmări și a reacționa la modificările valorilor și stării formei. Modificările de valoare și modificările de stare ale controalelor de formular pot fi observate folosind observables, permițând caracteristici avansate, cum ar fi actualizarea dinamică a câmpurilor de formular pe baza altor valori de câmp.

2.2.4 HTML5

HTML5 este cea mai recentă versiune a Hypertext Markup Language, care este limbajul standard folosit pentru crearea și structurarea paginilor web. Este o îmbunătățire față de versiunile anterioare de HTML și introduce noi caracteristici și capabilități.

În termeni simpli, HTML5 este un set de instrucțiuni pe care browserele web le înțeleg și le folosesc pentru a afișa conținutul unei pagini web. Acesta definește structura și aspectul paginii, cum ar fi titluri, paragrafe, imagini, link-uri și multe altele.

HTML5 introduce elemente noi care oferă mai multă semnificație și structură conținutului. De exemplu, etichetele <header>, <nav>, <section>, <article> și <footer> vă ajută să descrie scopul și organizarea diferitelor părți ale unei pagini web.

Suport multimedia: HTML5 include suport încorporat pentru redarea fișierelor audio și video direct în browser, fără a fi nevoie de pluginuri suplimentare precum Flash. Oferă elemente <audio> și <video>, facilitând încorporarea conținutului multimedia în paginile web.

Formulare îmbunătățite: HTML5 îmbunătățește intrările și validarea formularelor. Introduce noi tipuri de introducere, cum ar fi e-mailul, data, numărul și intervalul, precum și atribute pentru câmpurile obligatorii și potrivirea modelelor. Acest lucru îi ajută pe dezvoltatori să creeze formulare web mai interactive și mai ușor de utilizat.

Suport offline: HTML5 oferă funcții care permit aplicațiilor web să funcționeze offline. Include un mecanism de stocare web (localStorage) care permite stocarea datelor pe partea clientului, precum și capacitatea de a stoca în cache resurse web, astfel încât utilizatorii să poată accesa și interacționa cu aplicațiile web chiar și atunci când nu sunt conectați la internet.

Compatibil cu dispozitivele mobile: HTML5 include funcții care facilitează crearea de site-uri web și aplicații compatibile cu dispozitivele mobile. Oferă capabilități de design receptiv, permițând conținutului web să se adapteze și să se afișeze în mod corespunzător pe diferite dispozitive, cum ar fi smartphone-uri și tablete.

În general, HTML5 este un standard web modern care oferă funcționalități îmbunătățite, suport multimedia mai bun și interactivitate îmbunătățită. Le permite programatorilor să creeze pagini web mai dinamice și mai bogate în funcții, care funcționează fără probleme pe diferite dispozitive și browsere.

2.2.5 Bootstrap

Bootstrap este un framework pentru front-end open-source popular, care oferă o colecție de componente și stiluri HTML, CSS și JavaScript pre-proiectate. Acesta își propune să simplifice și să accelereze dezvoltarea web, oferind un set gata de utilizare de instrumente și șabloane pentru crearea de site-uri web și aplicații web receptive și atractive din punct de vedere vizual.

Bootstrap oferă o gamă largă de componente pre-stilizate, cum ar fi butoane, formulare, bare de navigare, carduri, modale, carusele și multe altele. Aceste componente vin cu stiluri și comportamente predefinite, facilitând încorporarea lor în proiecte fără a fi nevoie de scrierea lor de la zero. Se includ pur și simplu clasele Bootstrap corespunzătoare în marcajul HTML folosind „class=’clasa-bootstrap’” pentru a utiliza aceste componente.

Personalizare: În timp ce Bootstrap oferă stiluri și componente gata făcute, permite, de asemenea, personalizarea pentru a se potrivi cerințelor specifice ale proiectului. Se pot suprascrie stilurile implicite, modifica aspectul grilei și crea propriile clase CSS personalizate. Bootstrap oferă, de asemenea, un instrument de personalizare pe site-ul său oficial, unde se selectează componentele și stilurile de care e nevoie și se generează o versiune personalizată a framework-ului.

Responsive grid: Bootstrap utilizează un sistem de grid responsive, care permite crearea unui aspect flexibil și receptiv pentru paginile web. Sistemul de grid se bazează pe un aspect cu 12 coloane care poate fi personalizat pentru a se adapta la diferite dimensiuni de ecran. Permite aranjarea si alinierea cu ușurință a conținutului în rânduri și coloane, adaptându-se la diferite dispozitive și rezoluții de ecran.

Design responsive: Bootstrap prioritizează designul și funcționalitatea pentru dispozitivele mobile în mod implicit. Se asigură că site-ul sau aplicația arată și funcționează bine pe smartphone-uri,

tablete și ecrane desktop. Caracteristicile responsive ale Bootstrap ajută la crearea designuri fluide și adaptive care se ajustează automat în funcție de dispozitivul utilizatorului.

Plugin-uri JavaScript: Bootstrap include un set de plugin-uri JavaScript care adaugă interactivitate și funcționalitate îmbunătățită paginilor web. Aceste plugin-uri se ocupă de sarcini obișnuite, cum ar fi meniurile drop-down, modale, slidere și multe altele. Se pot încorpora și personaliza cu ușurință aceste pluginuri incluzând fișierele JavaScript Bootstrap și inițializandu-le în cod.

Folosind Bootstrap, programatorii pot economisi timp și efort în proiectarea și construirea front-end-ului site-urilor sau aplicațiilor web. El oferă o bază solidă cu componente pre-stilizate și o serie de opțiuni personalizabile, permițând crearea de interfețe cu aspect profesional și ușor de utilizat.

2.2.6 CSS

CSS (Cascading Style Sheets) este un limbaj de programare care funcționează împreună cu HTML pentru a determina cum ar trebui să arate și să fie prezentate paginile web. Vă permite să stilați și să formatați diferite elemente de pe un site web, cum ar fi schimbarea culorilor, fonturilor și machetelor. Folosind selectoare și reguli CSS, puteți controla aspectul anumitor elemente și puteți crea design-uri atrăgătoare și consistente pentru paginile dvs. web.

2.2.7 TypeScript

TypeScript este un limbaj de programare care extinde JavaScript prin adăugarea de tastare statică. Oferă o experiență de dezvoltare mai structurată și mai robustă, permițând dezvoltatorilor să detecteze erori și erori la începutul procesului de dezvoltare.

Siguranța tipului: TypeScript introduce tastarea statică, care vă permite să definiți tipurile de variabile, parametrii funcției și valorile returnate. Acest lucru ajută la identificarea erorilor și erorilor în timpul dezvoltării, deoarece compilatorul poate detecta nepotrivirile de tip și poate furniza mesaje de eroare utile. Permite detectarea timpurie a problemelor potențiale, reduce erorile de rulare și îmbunătățește calitatea codului.

Suport îmbunătățit pentru instrumente și IDE: TypeScript oferă suport bogat pentru instrumente și o integrare excelentă cu editoarele de cod și mediile de dezvoltare populare. Oferă funcții precum navigarea codului, completarea automată, instrumente de refactorizare și documentație îmbunătățită. Aceste instrumente sporesc productivitatea și fac programarea mai eficientă.

TypeScript este un superset de JavaScript, ceea ce înseamnă că orice cod JavaScript valid este, de asemenea, cod TypeScript valid. Această compatibilitate permite dezvoltatorilor să adopte treptat TypeScript în proiectele lor fără a rescrie bazele de cod existente.

Caracteristici ale limbajului: TypeScript extinde JavaScript cu caracteristici suplimentare ale limbajului, cum ar fi clase, interfețe, module, generice și decoratori. Aceste caracteristici le permit dezvoltatorilor să scrie cod mai structurat, modular și mai ușor de întreținut.

2.2.8 C#

C# este un limbaj de programare modern, orientat pe obiecte, dezvoltat de Microsoft. Este utilizat pe scară largă pentru dezvoltarea unei varietăți de aplicații, inclusiv software desktop, aplicații web, aplicații mobile, jocuri și multe altele.

Programare orientată pe obiecte (OOP): C# urmează principiile OOP, permițându-vă să definiți clase, obiecte și relațiile lor. Acceptă concepte precum încapsularea, moștenirea și polimorfismul, care promovează organizarea codului, reutilizarea și modularitatea.

Limbaj strongly typed: C# este un limbaj strongly typed, ceea ce înseamnă că variabilele trebuie declarate cu tipurile lor specifice. Implementează siguranța tipului și captează erorile legate de tip în timpul compilării, reducând riscul erorilor de rulare.

Gestionarea automată a memoriei: C# utilizează un colector de gunoi pentru a gestiona automat alocarea și dealocarea memoriei. Această caracteristică scutește dezvoltatorii de sarcinile manuale de gestionare a memoriei și ajută la prevenirea scurgerilor de memorie și a altor probleme legate de memorie.

Sintaxă simplificată: C# oferă o sintaxă curată și lizibilă, facilitând scrierea și înțelegerea codului. Include caracteristici precum proprietăți, evenimente, delegați și expresii lambda, care sporesc expresivitatea și reduc codul standard.

Independența platformei: Odată cu introducerea .NET Core, C# a devenit un limbaj multiplatformă, permițând programatorilor să creeze aplicații care rulează pe diferite sisteme de operare, inclusiv Windows, macOS și Linux.

Programare asincronă: C# are suport încorporat pentru programarea asincronă folosind cuvintele cheie async and await. Acest lucru le permite dezvoltatorilor să scrie aplicații adaptabile și scalabile care pot gestiona eficient sarcini precum operațiunile de rețea și I/O fără a bloca firul de execuție principal.

2.2.9 Github

GitHub este o platformă web și un sistem de control al versiunilor care le permite programatorilor să colaboreze la proiecte, să găzduiască depozite de coduri și să gestioneze fluxurile de lucru de dezvoltare software. Oferă o gamă largă de caracteristici și instrumente care facilitează partajarea codului, urmărirea problemelor, managementul proiectelor și colaborarea în echipă.

Controlul versiunilor: GitHub utilizează Git, un sistem distribuit de control al versiunilor, care permite dezvoltatorilor să urmărească modificările, să colaboreze la cod și să revină cu ușurință la versiunile anterioare dacă este necesar. Oferă funcții precum ramificarea, fuzionarea și rezolvarea conflictelor.

Colaborare: GitHub oferă un mediu de colaborare pentru dezvoltatori pentru a lucra împreună la proiecte. Oferă instrumente pentru revizuirea codului și urmărirea problemelor, permițând colaborarea eficientă și feedbackul între membrii echipei.

Integrarea într-un proiect din cadrul VS Code este facilă, trebuie urmați următorii pași:

- Instalarea sau asigurarea că avem Git instalat.
- Inițializarea unui repository git din terminalul integrat al mediului de programare folosind comanda „git init”

- Conectarea la github și crearea unui nou repository pentru proiect.
- Conectarea celor doua repository-uri din terminalul integrat folosind comanda „git remote add origin ‚url-repo-github' ”.
- Commit la modificările din proiect
- Push, transferăm repository-ul local pe Github.

3 Dezvoltarea aplicației

3.1 Arhitectura aplicației

Software-ul Educațional creat urmează arhitectura client-server, cu backend-ul implementat ca și proiect .NET WebAPI și frontend-ul dezvoltat folosind Angular.

La un nivel înalt, arhitectura aplicației constă din trei componente principale:

- Frontend-ul (client-side)
- Backend-ul (server-side)
- Stratul de comunicare care le conectează.

Frontend (Angular): Angular oferă o abordare modulară și structurată pentru dezvoltarea interfețelor utilizator. Avesta cuprinde diverse module, componente, servicii și directive care lucrează împreună pentru a crea o experiență de utilizator fără întreruperi. Componenta frontend a aplicației interacționează cu backend printr-un set de API-uri expuse de proiectul WebApi. Ea se ocupă de interacțiunile utilizatorului, prezintă date si randează interfața utilizatorului bazat pe răspunsurile primite. Este responsabil pentru furnizarea unei interfețe intuitive, ușor de utilizat.

Backend (.NET WebAPI): WebAPI este un framework din ecosistemul .NET care permite creare de servicii bazate pe HTTP, pemițând clienților să interacționeze cu serverul folosind stilul arhitectural RESTful. Această componentă se ocupă în primul rând de stocarea datelor, procesarea si logica de business. Aici se definesc funcționalitățile ți operațiunile disponibile pentru aplicația client. Aceste API-uri sunt responsabile pentru primirea request-urilor de la frontend, procesarea lor, interacțiunea cu sursele de date (în cazul nostru, baza de date) și returnarea răspunsurilor adecvate către frontend. Backend-ul poate include diverse module cum ar fi autentificarea și autorizarea, gestionarea utilizatorilor, gestionarea conținutului și a funcționalităților.

Stratul de comunicare acționează ca o punte între componentele frontend și backend, facilitând schimbul de date prin request-uri și response-uri. Acest nivel se bazează pe protocoalele HTTP sau HTTPS pentru comunicare. Frontend-ul interacționează cu backend-ul făcând request-uri HTTP către anumite endpoint-uri, metode de acțiune, expuse de către proiectul WebAPI. Aceste

request-uri includ acțiuni precum preluarea datelor utilizatorului, trimiterea de formulare, preluarea conținutului necesar. Backend-ul procesează cererile, efectuează opreațiunile necesare și returnează răspunsuri, de regulă în format JSON sau XML care conțin datele solicitate de către frontend. Acest strat poate include și alte elemente precum mecanisme de securitate, limitarea ratei de request-uri și mecanisme de caching pentru a îmbunătăți performanța.

Această configurație permite o aplicație scalabilă, modulară și sigură.

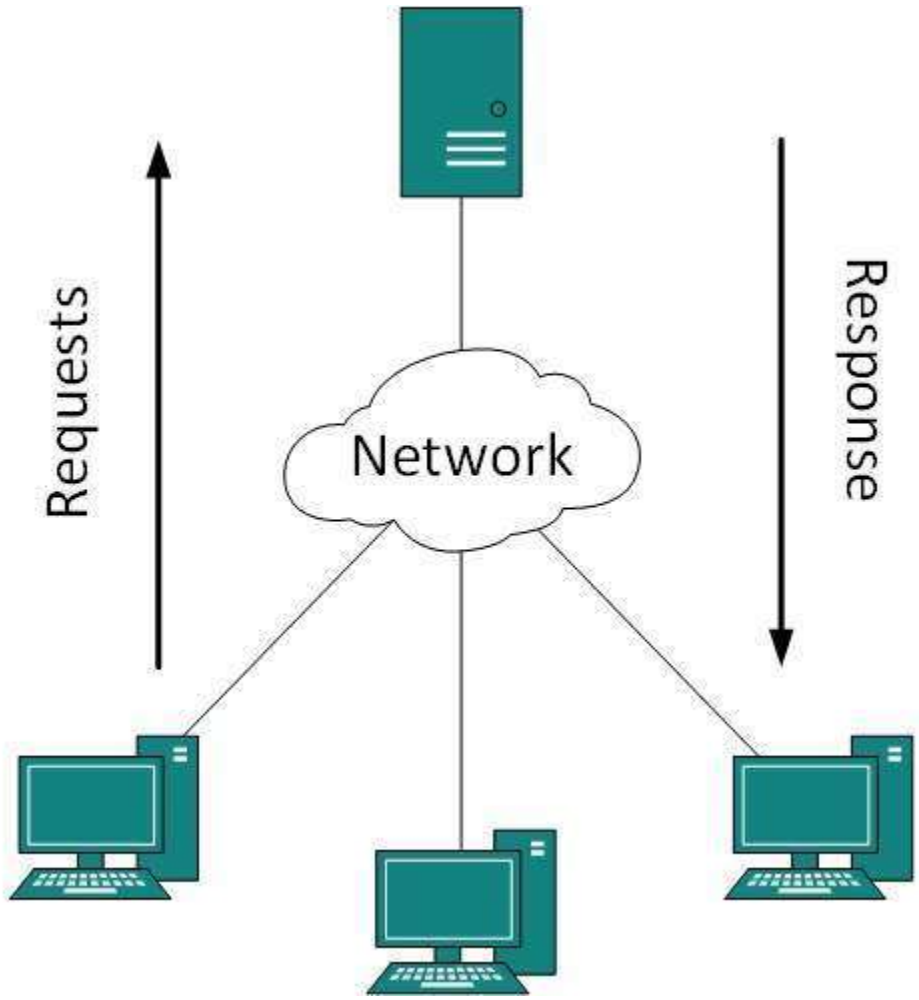

Figura 2 – Arhitectura client server. Preluată din [11]

În această diagramă, unitatea de sus reprezintă serverul și calculatoarele din partea de jos reprezintă multiplii clienți care fac solicitări și primesc răspunsuri.

3.2 Interfața și experiența utilizatorului

3.2.1 Descriere

La pornirea aplicației utilizatorul va ajunge pe pagina de start a aplicației. Aceasta cuprinde

- Numele aplicației
- O scurtă descriere
- Un buton pentru informații de suplimentare
- Un buton pentru a adăuga formularul de înregistrare
- Formularul de autentificare

Figura 3 – Pagina de start

Pentru vizualizarea celorlalte funcționalități, utilizatorul trebuie fie să se autentifice (în cazul în care are deja cont creat) sau să își creeze un cont (caz în care va fi autentificat automat dacă crearea contului este cu succes).

Odată autentificat, utilizatorul cu rol de profesor este dus pe pagina principală a profesorului, cea unde poate vizualiza cele mai importante informații.

Figura 4 – Pagină după autentificare

De aici, profesorul are posibilitatea de a rămâne pe această pagină și să vizualizeze detalii referitoare la studenți, să adauge note, să verifice prezența și să creeze teste. Celelalte opțiuni sunt:

- Categories: pentru vizualizarea sau adăugarea categoriilor de întrebări
- Questions pentru vizualizarea, sortarea, filtrarea și adăugarea întrebărilor
- Tests pentru vizualizarea și corectarea testelor
- Log out (apăsând pe butonul din dreapta sus unde apare mesajul de "Welcome 'username'") și selectând opțiunea de log out.

3.2.2 User stories

Ca orice utilizator vreau să mă pot autentifica

Ca și profesor vreau să:

- adaug probleme pentru teste in baza de date
- creez noi categorii de probleme
- pot vizualiza situatia tuturor claselor si elevilor proprii
- pot genera teste personalizate fiecărui elev (variabile: grad dificultate, număr probleme, categorie probleme, tip răspuns)
- introduc răspunsurile elevilor iar aplicația să returneze nota
- pot introduce note elevilor
- pot face prezența elevilor
- pot vizualiza, filtra și ordona toată lista de probleme.

Ca și student vreau să:

- pot susține testele care mi-au fost atribuite mie
- îmi pot vizualiza notele

3.2.3 Use case-uri

Use case diagram:

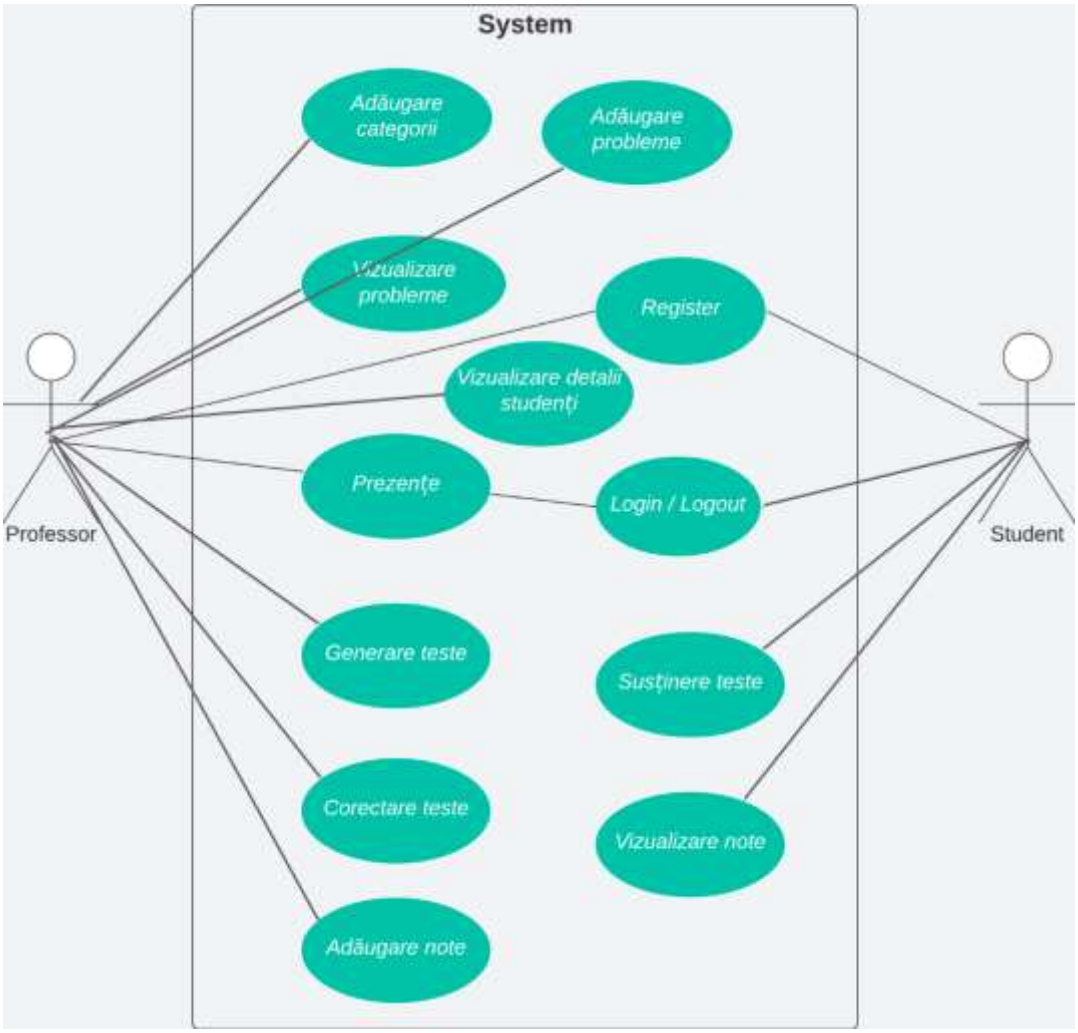

Figura 5 – diagramă use case

Use case-uri principale:

Use case:

Vizualizare detalii studenți.

Descriere:

Profesorul are opțiunea de a vizualiza informații despre studenți, notele acestora și prezențele lor.

Actori:

- Profesor
- Studenți

Precondiții:

- Profesorul trebuie să fie autentificat

- Profesorul trebuie să predea la cel puțin o clasă
- Clasele la care predă profesorul trebuie să conțină minim un student

Scenariul principal:
Profesorul intră pe pagina cu clasele sale și alege una dintre ele. În mod implicit se vor încărca informațiile studenților din acea clasă. Profesorul are posibilitatea de a selecta ce anume dorește să vadă: informații, note sau prezențe.
Postcondiții: -

3.2.4 User requirements

User Management:
Fiecare utilizator își va putea crea propriul cont folosind un formular de înregistrare. Inițial am dorit introducerea unui user de tip administrator pentru gestionarea conturilor.

Login:
Toți utilizatorii trebuie să se autentificie pentru a putea folosi aplicația.

Logout:
Utilizatorii au posibilitatea de a se deconecta în orice moment din aplicație.

Vizualizare alți useri:
Doar utilizatorii cu rol de profesor vor putea vizualiza detalii referitoare la alți utilizatori. Singurii ultilizatori pe care îi vor putea vizualiza sunt studenții din propriile clase. Aceștia vor avea acces la detaliile studenților(nume, prenume, email, număr de telefon), notele acestora(note normale și notele pe categorii) și prezențele la ore.

Vizualizare probleme și categorii:
Doar utilizatorii cu rol de profesor vor putea vizualiza/sorta/filtra lista de probleme și cea de categorii.

Adăugare probleme și categorii:
Utilizatorii autentificați cu rol de profesor sunt singurii care pot să adauge noi categorii pentru probleme. Pentru adăugarea problemelor ei vor trebui să introducă textul, categoria, dificultatea și tipul răspunsului.

Generarea testelor:

Utilizatorul autentificat cu rol de profesor poate să genereze teste individuale selectând pentru fiecare numărul total de probleme, numărul de probleme dintr-o anumită categorie, gradul de dificultate pentru fiecare problemă.

Corectarea asistată a testelor:

Testele se vor corecta pe rând doar de către profesor. Identificarea testului curent se va face cu ajutorul unui id. Răspunsurile la întrebările de tip grilă sunt introduse de profesor de pe foile de răspuns ale elevilor si evaluate automat de către aplicație. Răspunsurile la întrebările cu răspuns deschis sunt corectate de profesor, care introduce apoi doar nota la obiectivul respectiv.

La finalul corectării, notele elevului vor fi actualizate automat.

Adăugarea notelor:

Utilizatorii autentificați cu rol de profesor pot adăuga note(altele decât cele rezultate din urma testelor) doar propriilor studenți.

Susținerea testelor:

Utilizatorii cu rol de student vor putea să susțină testele online la ora la care acestea au fost stabilite.

3.3 Structura bazei de date

Pentru această aplicație am creat baza de date relațională reprezentată în următoarea diagramă.

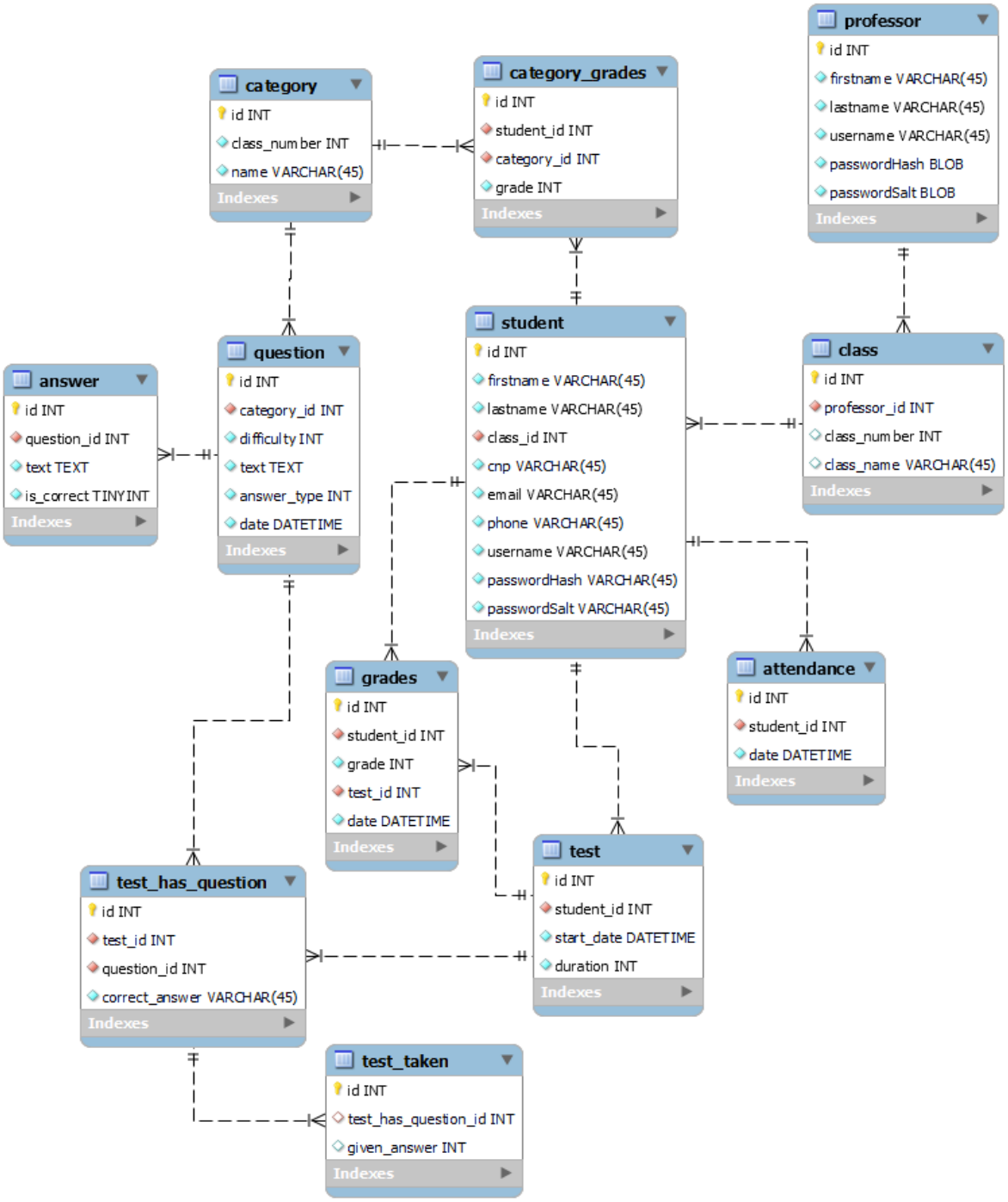

Figura 6 – Diagramă bază de date

Am încercat să simplific pe cât posibil structura bazei de date și să o aduc cât mai aproape de formele normale ale bazelor de date, păstrând totuși posibilitatea de a lucra ușor cu ea în cadrul aplicației.

Utilizatorii fiind componenta centrală a aplicației, am creat câte o tabelă pentru fiecare tip de utilizator al aplicației. Fiecare utilizator va avea nume, prenume, username, 2 câmpuri pentru parolă și detalii suplimentare aferente fiecărui tip de utilizator.

Aplicația a fost gândită pentru utilizarea în învățământul preuniversitar din România. Aici elevii sunt împărțiți în clase și de regulă au un singur profesor la o anumită materie. Astfel tabela „class" devine o tabelă de legătură între tabelele „professor" și „student" definind o relație de tipul „one to many" de la profesor la student (un profesor poate avea mai mulți studenți, un student are un singur profesor, în acest caz, de matematică). Această tabelă mai are 2 câmpuri, unul care reprezintă numele (de ex. 12-D) iar celălalt reprezintă numărul clasei (de ex. 12). Numărul va fi folosit in aplicație pentru sortări/căutări în timp ce numele va fi folosit strict pentru afișare si diferențiere între clase de aceeași generație.

O altă componentă importantă a aplicației o reprezintă împărțirea pe categorii a întrebărilor și notelor elevilor. Astfel am creat tabela „category" care conține numărul clasei si numele categoriei (de ex: o categorie pentru toate clasele de a 9-a este funcția de gradul doi). Id-ul categoriei va fi cheie străină în tabela cu întrebări respectiv tabela notelor la fiecare categorie.

Fiecare întrebare va avea o categorie de care aparține, un grad de dificultate (de la 1 la 3), textul întrebării, data la care a fost adăugată și tipul răspunsului care poate să fie de 2 feluri: Răspuns deschis (doar pentru cazul testelor printate și punctate direct de către profesor în timpul corectării asistate) sau răspuns grilă.

În cazul întrebărilor cu răspuns grilă, pentru a respecta formele normale ale bazelor de date si a nu păstra o listă de răspunsuri în tabela de întrebări, am creat o tabelă de răspunsuri care va conține toate variantele de răspuns. Această tabelă va avea ca si cheie străină id-ul întrebării de care aparține, textul răspunsului și un flag care să marcheze dacă răspunsul este cel corect pentru această întrebare, sau nu.

Scopul principal al aplicației fiind acela de a genera teste, este necesară o tabelă și pentru acestea. Fiecare test va fi asignat unui student, va avea o dată și oră de susținere (în cazul susținerii online) și o durată (tot în cazul susținerii online). Pentru testele care urmează să fie exportate și printate nu sunt relevante data și durata, ele fiind susținute fizic la școală sub supravegherea profesorului.

Legătura dintre teste si întrebări se face prin tabela de legătură „test_has_question" care definește o relație „many to many" între cele două tabele. Orice întrebare poate să apară pe mai multe teste și orice test poate să conțină mai multe întrebări. Pe lângă aceste câmpuri, tabela va reține și răspunsul corect la întrebare.

Pentru reținerea răspunsurilor date de studenți la testele online există tabela „test_taken" care are ca si cheie străină un „test_has_question" id prin care putem identifica testul, întrebarea și răspunsul corect. Ea mai conține un câmp pentru răspunsul dat de student. Această tabelă va fi folosită pentru corectarea automată a testelor online.

Tabela „grades" va conține notele studenților așa cum sunt în sistemul actual de învățământ. Această tabelă va avea două chei străine, una pentru a identifica studentul și cealaltă pentru a identifica testul la care a fost acordată nota. Ea va mai avea un câmp pentru notă și unul pentru data la care a fost corectat testul.

Pentru ca profesorul să aibă o evidență detaliată asupra notelor studenților, a fost necesară introducerea unor note pe categorii. Acestea vor fi stocate în tabela „category_grades" care va conține nota, id-ul studentului care primește nota și id-ul categoriei la care a primit nota.

Ultima tabelă din baza de date este cea folosită pentru notarea prezențelor studenților. Ea conține doar id-ul studentului ca si cheie străină și data la care acesta a fost prezent.

Utilizarea Entity Framework în cadrul proiectului:
Pentru a introduce Entity Framework în proiect trebuie să instalăm extensia NuGet Gallery, care este un packet manager cu ajutorul căruia vom instala pachetele necesare pentru EF (Microsoft.EntityFrameworkCore.Design, Microsoft.EntityFrameworkCore.Sqlite).

Pentru faza de development a aplicației am folosit Sqlite datorită ușurinței de utilizare. Astfel nu este necesară configurarea unui server de baze de date, securitate, etc. Sqlite folosește doar un fișier pentru memorarea datelor.

Pentru utilizarea funcționalităților aduse de EF trebuie creată o clasă care să moștenească clasa „DbContext", aceasta reprezintă o sesiune cu baza de date și este folosită pentru accesul la aceasta. Constructorul acestei clase va primi ca și parametru string-ul de conectare la baza de date. Pentru fiecare tabel din baza de date vom avea o proprietate de tipul „DbSet<type> numeTabel". Această clasă va fi folosită ca un serviciu în aplicația noastră, ea va fi injectată in clasele unde avem nevoie de acces la baza de date. Pentru a indica aplicației faptul că este un serviciu, trebuie să specificăm asta în fișierul Program.cs la secțiunea de servicii. Realizăm asta folosind următoarea secvență de cod:

services.AddDbContext<DataContext>(opt => {opt.useSqlite('connectionString');});

Connection string-ul se poate introduce manual, sau îl luăm din unul dintre fișierele „appsettings.json" sau „apsettings.Development.json" cu ajutorul funcției „builder.Configuration.GetConnectionString(‚DefaultConnection')", unde DefaultConnection reprezintă cheia valorii connection string-ului din fișierul „appsettings.json".

Crearea unui connection string este foarte simplă, în fișierul „appsettings.json" adăugăm o nouă configurație cu cheia „ConnectionStrings" (obligatoriu la plural, chiar dacă oferim un singur string) și valoarea „Data source=numeBazaDate.db".

Odată ce avem connection string-ul creat, clasa de tipul DbContext cu cel puțin o proprietate DbSet, va trebui creată o „migrare" pentru ca EF să ne creeze baza de date. Migrarea este procesul prin care se sincronizează schema bazei de date cu modelul definit de noi în codul sursă. Această migrare se va uit în clasa DbContext și se va uita ce proprietăți avem, pentru fiecare proprietate de tip DbSet va crea un tabel așa cum se specifică prin codul aplicației. De exemplu pentru linia *public DbSet<AppUser> Users* EF va crea un tabel numit „Users" mapând atributele din clasa „AppUser" la câte un câmp din tabelă. Migrarea se crează din lina de comandă folosind comanda *dotnet ef migrations add „numeMigrare" -o „Path/Migrare"*. Pentru a aplica migrarea trebuie să actualizăm baza de date folosind comanda *dotnet ef database update*.

3.4 Detalii tehnice de implementare

3.4.1 „Walking skeleton”

Primul pas făcut în implementarea proiectului a fost crearea unui „Walking skeleton” atât pentru aplicația de back-end, cât și pentru cea de front-end. Termenul „skeleton” se referă la conținutul aplicației care la început va conține doar componentele arhitecturale fundamentale iar termenul „walking” implică faptul că scheletul este capabil de funcționalitatea minimă, permițându-i interacțiunea de bază client - server.

Pentru realizarea scheletului de back-end, am creat proiectul, am instalat pachetele și extensiile necesare, am definit câteva entități, am încorporat entity framework, am adăugat câteva date în baza de date, am adăugat un controller unde am tratat asincron un request și am configurat rutarea spre controller și metoda care tratează request-ul. În ultima parte am mai creat un repository de Git pentru a gestiona versiunile.

Pașii urmați pe partea de front-end au fost următorii: Crearea proiectului, familiarizarea cu fișierele importante din cadrul proiectului, adăugarea extensiilor, importarea modulului HttpClient în fișierul app.module.ts pentru a putea injecta o instață a acestuia in componente sau oriunde avem nevoie să facem request-uri, crearea unei componente Angular de unde facem un request către metoda implementată în back-end pentru a putea verifica funcționalitatea aplicației, adăugarea bootstrap și în cele din urmă configurarea aplicației astfel încât aceasta să poată rula și pe protocolul HTTPS.

3.4.2 Repository pattern

Repository pattern este un design pattern folosit de regulă pentru separarea logicii de acces la date de logica de business a unei aplicații. El introduce un nivel suplimentar de abstractizare între aplicație și baza de date. Într-un repository, de regulă, se implementează doar metode de tip CRUD, ascunzând astfel complexitatea și detaliile accesului la date. Oferă o oarecare flexibilitate suplimentară pentru a modifica structura bazei de date sau chiar și provider-ul de baze de date, fiind necesară doar modificarea metodelor din repository fără a modifica restul aplicației. În anumite cazuri, pattern-ul poate introduce complexitate inutilă aplicației dar, dat fiind faptul că acest proiect este unul din care am încercat să învăț cât mai multe, am considerat că merită introdus.

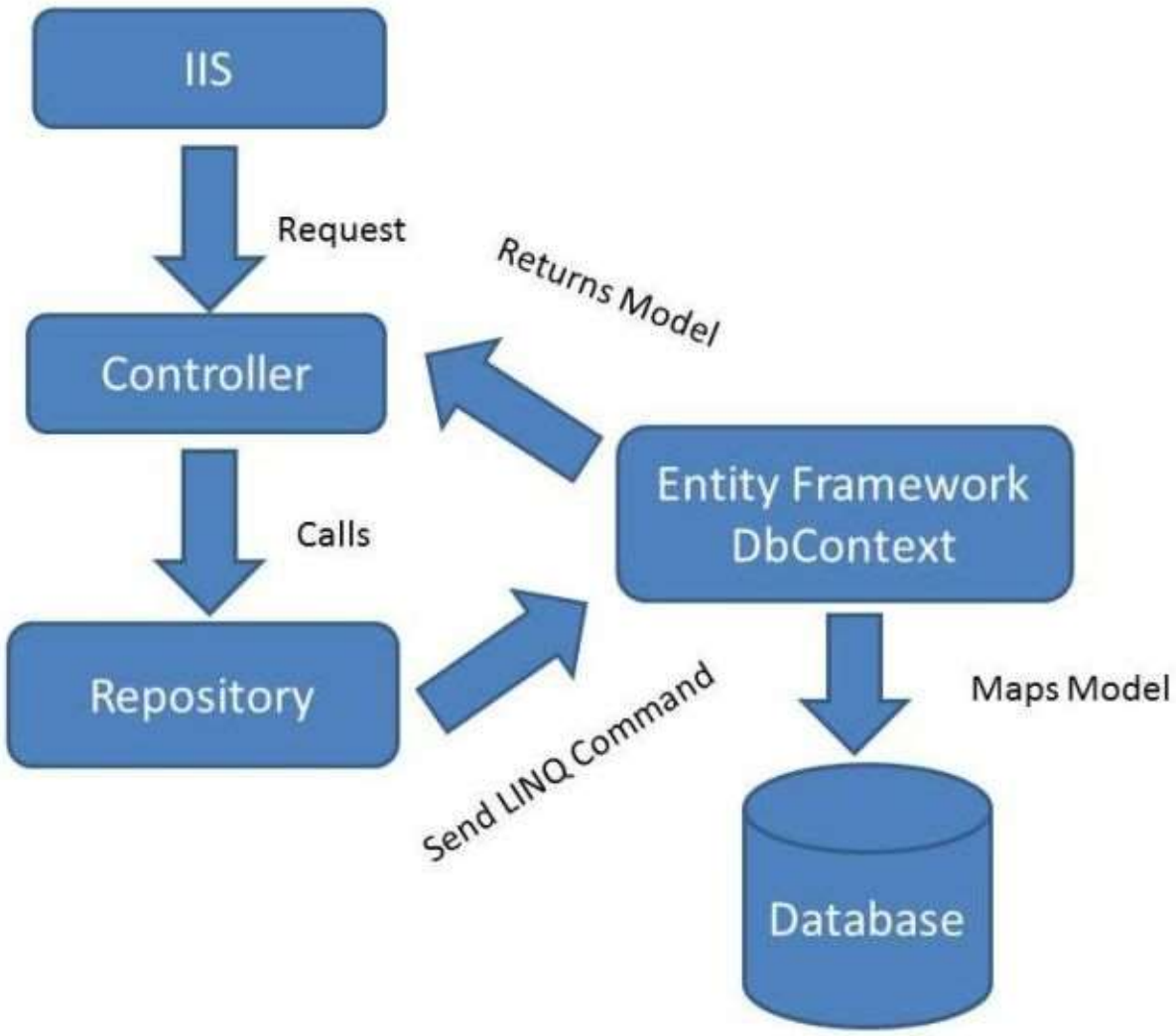

Figura 7 – Repository Pattern. Preluată din [9]

3.4.3 Testare folosind Postman

Postman este un instrument software folosit pentru testarea API-urilor web. Oferă o interfață grafică ușor de utilizat, care permite trimiterea de request-uri HTTP către API-uri și vizualizarea răspunsurilor. Oferă funcții precum editarea antetelor request-urilor, includerea de parametri query, editarea corpului request-urilor și opțiunile de autentificare pentru a simula diferite scenarii și a testa funcționalitatea și comportamentul API-ului.

El permite crearea de workspace-uri, salvarea și organizarea request-urilor în foldere pentru reutilizare.

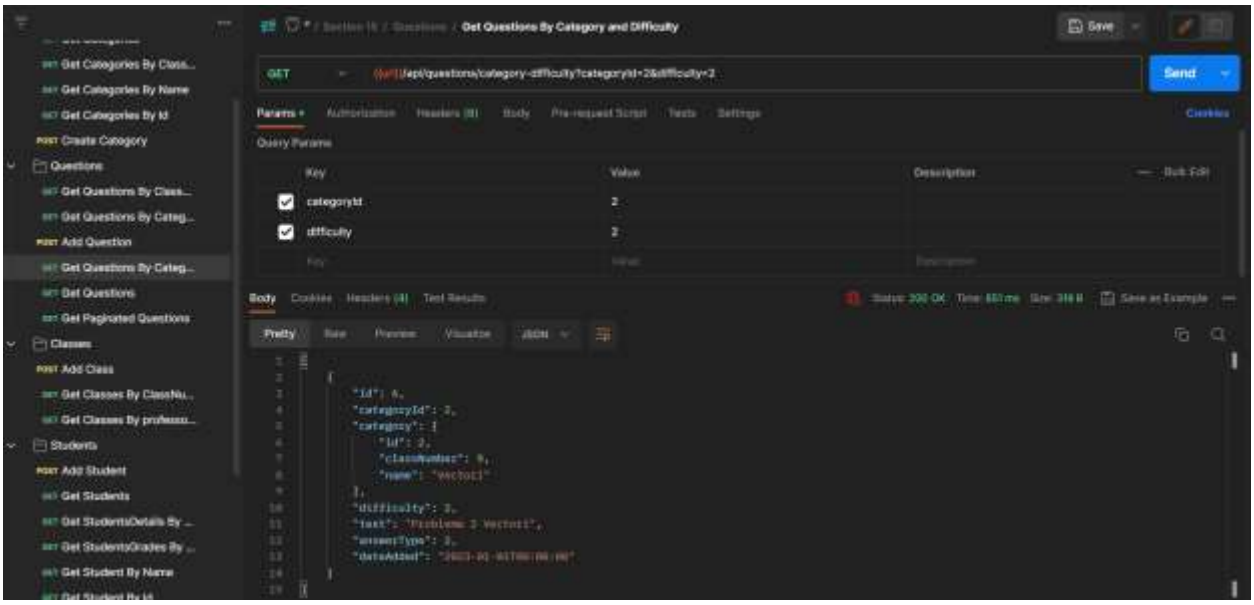

Figura 8 – Exemplu complet request Postman

În figura de mai sus se poate vedea o parte din interfața Postman. În partea stângă este workspace-ul în care am creat mai multe request-uri pentru a testa fiecare endpoint al API-ului. În zona superioară se crează sau modifică un request existent și în partea de jos sunt opțiunile pentru vizualizarea răspunsului primit în urma request-ului.

Figura 9 – Alt mod de vizualizare răspuns

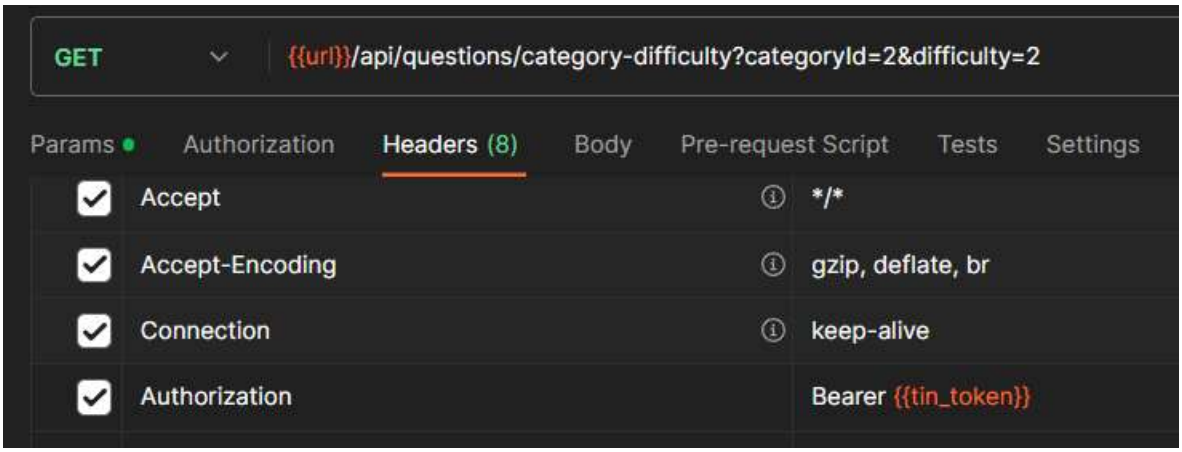

Figura 10 – Opțiuni creare request Postman(secțiunea pentru antete)

3.4.4 Adăugarea unei noi funcționalități aplicației

Pentru adăugarea fiecărei noi funcționalități aplicației am respectat următorii pași:

Pe partea de server:

- Crearea unei noi entități dacă noua funcționalitate implică un nou tip de date.
- Completarea clasei DataContext adăugând o nouă proprietate de tipul DbSet pentru ca Entity Framework să creeze un nou tabel pentru entitate.
- Crearea unei interfețe unde se declară metodele folosite în repository-ul pentru această entitate
- Crearea repository-ului care implementează interfața menționată mai sus. Aici injectăm o instanță a clasei DataContext pentru accesul la baza de date. Metodele din repository care returnează ceva se declară și implementează ca fiind asincrone, chiar dacă în interfața pe care o implementăm ele nu sunt declarate asincrone.
- În fișierul "ApplicationServiceExtensions" se adaugă un nou service pentru repository-ul creat anterior pentru ca acesta să poată fii injectat in controllere.
  services.AddScoped<IQuestionRepository, QuestionRepository>();
- Crearea unui nou controller care implementează clasa BaseApiController în care injectăm repository-ul creat anterior. Aici se definesc toate endpointurile și metodele de acțiune care tratează request-urile pe care le va folosi noua funcționalitate.
- În cazul în care din controller nu vrem să returnăm entitatea completă către client, ci doar anumite proprietăți, creăm o clasă tip DTO (Data Transfer Object) care va conține doar proprietățile pe care vrem să le transmitem și configurăm un profil de mappare de la entitate la DTO dacă este nevoie.
- În cele din urmă testăm cele realizate simulând niște request-uri folosind Postman.

Pe partea de client:

- Crearea unei clase care să corespundă cu ceea ce trimitem și primim de la server.
- Crearea unui nou service pentru această clasă. În service se injectează o instanță de HttpClient pentru a putea face request-urile către server.

- Crearea unei noi componente în care injectăm service-ul pentru a apela metodele care fac request-uri către server.
- Crearea șablonului componentei pentru a afișa datele primite, în cazul în care este nevoie.

3.4.5 Modificarea codului pentru a deveni asincron

Codul asincron din .NET WebAPI permite executarea paralelă a operațiunilor. Acesta permite serverului să gestioneze mai multe request-uri eficient, fără a bloca firul de execuție.
Cuvintele cheie folosite pentru a scrie cod asincron sunt **„async”** și **„await”**. Primul se folosește în semnătura metodelor pentru a indica faptul că metoda poate efectua operații asincrone. Acest lucru permite metodei să se execute concomitent cu alte operațiuni.

*public **async** Task<ActionResult> AddCategoryGrade(CategoryGrade categoryGrade)*

„await” este folosit în cadrul unei metode asincrone, înainte de o operație după care se poate aștepta cum ar fi o solicitare de rețea, o interogare la baza de date sau o operație I/O. Cuvântul cheie await indică faptul că metoda ar trebui să întrerupă execuția până la finalizarea operației așteptate.
*var student = **await** _studentRepository.GetStudentById(id);*

Gestionarea răspunsului: Când se finalizează o operație asincronă, se poate continua procesarea rezultatului în metoda asincronă. Aceasta poate implica transformarea datelor, actualizarea răspunsului sau efectuarea de operații suplimentare.

3.4.6 Paginare, sortare, filtrare și caching

Pentru a crea teste cât mai diferite între ele este nevoie de o colecție mare de întrebări. Profesorii trebuie să aibă opțiunea de a vizualiza întrebările pentru a se asigura, ocazional, de corectitudinea acestora, pentru a vedea dacă există întrebări prea vechi care nu mai corespund cetințelor actuale, pentru a introduce noi probleme fără ca acestea să existe deja în baza de date.

Acest aspect poate cauza probleme, atunci când sunt foarte multe întrebări. Soluția la aceste probleme, în cazul acestei aplicații, este introducerea opțiunilor de sortare, filtrare și paginare. Practic, când se face un request către server pentru a aduce întrebările, de pe partea clientului vom trimite un obiect de tip QuestionParams care va conține următoarele proprietăți:

- categoryId – reprezintă categoria din care face parte întrebarea, default are valoarea 0, caz în care la filtrare nu se ține cont si de categorie.
- difficulty – poate lua valorile 1, 2, 3 sau 4 și reprezintă dificultatea întrebărilor pe care vrem să le vizualizăm. Dificultățile fiind de la 1 la 3, atunci când difficulty = 4 nu se va ține cont de dificultate în procesul de filtrare.
- answerType – reprezintă tipul răspunsului. Ia valorile 1 sau 2 și 0 atunci când nu ne interesează tipul răspunsului.
- pageNumber – numărul paginii pe care vrem să o vizualizăm
- pageSize – numărul de întrebări de pe o pagină
- orderBy – felul în care vrem să fie ordonată lista. Poate lua valorile “dateAdded”, “difAscending” sau “difDescending”

Implementare paginare, filtrare, sortare pe client

În aplicația client, am creat două funcții ajutătoare care folosesc tipuri generice pentru a putea fi folosite cu ușurință și pentru alte tipuri de date, nu doar pentru întrebări.

Funcția care face request-ul este următoarea:

```
export function getPaginatedResult<T>(url: string, params: HttpParams, http: HttpClient) {
  const paginatedResult: PaginatedResult<T> = new PaginatedResult<T>;
  return http.get<T>(url, { observe: 'response', params }).pipe(
    map(response => {
      if (response.body) {
        paginatedResult.result = response.body;
      }
      const pagination = response.headers.get('Pagination');
      if (pagination) {
        paginatedResult.pagination = JSON.parse(pagination);
      }
      return paginatedResult;
    })
  );
}
```

Ea va returna un obiect de tip paginatedResult<T> care are 2 proprietăți.

- Result de tip T
- Un obiect de tip pagination care conține informații referitoare la paginare (numărul paginii, numărul de elemente pe pagină, numărul total de elemente, numărul total de pagini)

A doua funcție returnează un obiect de tip HttpParams prin care trimitem obiectul de tip QuestionParams spre server:

```
export function getPaginationHeaders(pageNumber: number, pageSize: number) {
  let params = new HttpParams();
  params = params.append('pageNumber', pageNumber);
  params = params.append('pageSize', pageSize);
  return params;
 }
```

Ordinea apelării funcțiilor pe partea de client este simplă și începe în componenta "questions" care conține un obiect de tip questionParams și are un "questionService" injectat. Atunci când se construiește componenta, obiectul questionParams ia valoarea unui obiect de același tip din "questionService".

```
constructor(private questionService: QuestionService, private categoryService: CategoryService) {
  this.questionParams = this.questionService.getQuestionParams();
 }
```

Acesta este formularul pe care îl au la dispoziție utilizatorii pentru a selecta parametrii de filtrare:

Figura 11 - Opțiuni filtrare întrebări

Oricare buton se apasă se va apela metoda getPaginatedQuestions din componentă care verifică existența obiectului questionParams și la rândul ei apelează metoda getPaginatedQuestions din questionService.

```
this.questionService.getPaginatedQuestions(this.questionParams).subscribe({
      next: response => {
        if (response.result && response.pagination) {
          this.questions = response.result;
          this.pagination = response.pagination;
        }
      }
    })
```

Dacă primim un răspuns actualizăm lista de întrebări cu noile întrebări primate și detaliile legate de paginare.

Metoda din questionService primește ca și parametru obiectul de tip QuestionParams pe care îl transformă într-un obiect de tip HttpParams pentru a putea fi transmis mai departe serverului prin request.

```
let params = getPaginationHeaders(questionParams.pageNumber, questionParams.pageSize);
  params = params.append('categoryId', questionParams.categoryId);
  params = params.append('answerType', questionParams.answerType);
  params = params.append('difficulty', questionParams.difficulty);
  params = params.append('orderBy', questionParams.orderBy);
```

După aceea se apelează funcția getPaginatedResult() descrisă mai sus care va returna în acest caz un array de obiecte de tip Question și primeste ca și parametri URL-ul către care se face request-ul, obiectul params de tip HttpParams și instanța de HttpClient din questionService.

```
return getPaginatedResult<Question []>(this.baseUrl + 'questions/paginated', params, this.http).pipe(
    map(response => {
      this.questionsCache.set(Object.values(questionParams).join('-'), response);
      return response;
    })
  )
```

Răspunsul primit de questionService este adăugat într-un “questionsCache” care este descris mai jos, după descriere implementării pe server.

Implementare paginare, filtrare, sortare pe server

Pe server pentru implementare am creat o clasă PagedList<T> care moștenește clasa List<T> și conține următoarele atribute:

- CurrentPage
- TotalPages
- PageSize
- TotalCount
- Lista propriu-zisă de obiecte de tip <T>

Clasa mai conține metoda care crează lista paginată:

```
public static async Task<PagedList<T>> CreateAsync(IQueryable<T> source,
    int pageNumber, int pageSize)
{
    Console.WriteLine("PagedList CreateAsync");
    var count = await source.CountAsync();
    var items = await source.Skip((pageNumber - 1)*pageSize)
                            .Take(pageSize)
                            .ToListAsync();
    return new PagedList<T>(items, count, pageNumber, pageSize);
}
```

Ca și pe aplicația client, această clasă permite reutilizarea ei pentru diferite tipuri de date.

Request-ul ajunge în controller-ul specific întrebărilor, QuestionsController, și se apelează metoda GetPaginatedQuestions din questionRepository care primește ca și parametru (din query) un obiect de tip QuestionsParams.

```
var questions = await _questionRepository.GetPaginatedQuestions(questionsParams);
```

În questionRepository se adaugă unui query filtrele și se returnează un PagedList<Question>

```
var query = _context.Questions.AsQueryable();

    if (questionsParams.Difficulty != 4)
    {
    query = query.Where(question => question.CategoryId == questionsParams.CategoryId);
    query = query.Where(q => q.difficulty == questionsParams.Difficulty);
```

```
        query = query.Where(q => q.AnswerType == questionsParams.AnswerType);
        }

        query = questionsParams.OrderBy switch
        {
            "difDescending" => query.OrderByDescending(q => q.difficulty),
            "difAscending" => query.OrderBy(q => q.difficulty),
            _ => query.OrderBy(q => q.DateAdded)
        };

        return await PagedList<Question>.CreateAsync(
            query.AsNoTracking(),
            questionsParams.PageNumber,
            questionsParams.PageSize);
```

Înapoi în controller, folosind o metodă de extensie "AddPaginationHeader()" care extinde un HttpResponse, adăugăm în header-ul response-ului detaliile legate de paginare.

Metoda de extensie din HttpExtensions.cs

```
public static void AddPaginationHeader(this HttpResponse response, PaginationHeader header)
    {
        var jsonOptions = new JsonSerializerOptions{PropertyNamingPolicy =
JsonNamingPolicy.CamelCase};
        response.Headers.Add("Pagination", JsonSerializer.Serialize(header,jsonOptions));
        response.Headers.Add("Access-Control-Expose-Headers", "Pagination");
    }
```

Partea finală a metodei de acțiune din controller-ul de întrebări.

```
Response.AddPaginationHeader(new PaginationHeader(
            questions.CurrentPage,
            questions.PageSize,
            questions.TotalCount,
            questions.TotalPages));

 return Ok(questions);
```

Caching pe aplicația client

Presupunând că un utilizator vrea să facă aceeași filtrare de mai multe ori, pentru a nu face un request către server de fiecare dată când se repetă un set de filtre am introdus în questionService un obiect care să rețină listele de de întrebări aparținând unui anumit set de filtre.

questionsCache = new Map();

Tipul Map va reține perechi cheie-valoare unde în cazul nostru cheia o va reprezenta setul de filtre și valoarea va fi lista de întrebări.

A fost necesară adăugarea cache-ului în questionService deoarece durata de viață a unui service într-o aplicație Angular este de la pornirea aplicației până la închiderea acesteia. Componentele se distrug de fiecare dată când trecem de la o componentă la alta și din acest motiv nu sunt un loc recomandat pentru stocarea informațiilor de care avem nevoie un timp mai îndelungat.

De fiecare dată când se apelează metoda getPaginatedQuestions din service primul lucru care se efectuează este verificare dacă nu cumva avem deja în cache pagina solicitată. Dacă ea există deja, se returnează sub forma unui Observable.

```
const response = this.questionsCache.get(Object.values(questionParams).join('-'));
  if(response) return of(response);
```

Cheia este create din toate atributele obiectului questionParams despărțite de o cratimă.

Dacă nu avem în cache pagina solicitată, atunci se face request-ul către server, se aduce pagina și se introduce în cache.

```
this.questionsCache.set(Object.values(questionParams).join('-'), response);
```

3.4.7 Comunicarea între componente

În Angular, comunicarea între componente poate fi realizată folosind diverse tehnici bazate pe relația dintre componente și direcția fluxului de date. Tehnicile pe care eu le-am folosit în această aplicație sunt:

- Comunicare părinte-copil (input binding): Aceasta este cea mai simplă formă de comunicare. O componentă de tip părinte poate transmite date către o componentă de tip copil folosind proprietățile de intrare. Componenta copil primește datele ca intrare și le poate folosi în cadrul șablonului sau logicii sale. Componenta părinte controlează datele care sunt transmise componentei copil. În componenta copil definim variabilele pe care le primim de la componenta părinte cu decoratorul "@Input".
  Un exemplu din aplicația mea pentru acest tip de comunicare se regăsește în componentele "classes" și "students" (classes fiind componenta părinte):

  <app-students [schoolClass]="selectedClass" [mode]="viewMode"></app-students>

  @Input() schoolClass: SchoolClass | undefined;
  @Input() mode: string | undefined;

  Prima linie reprezintă modul de transmitere din șablonul componentei părinte. Între paranteze drepte se specifică numele proprietății din componenta copil și între ghilimele obiectul din componenta părinte pe care vrem să îl transmitem. Următoarele două linii reprezintă declararea atributelor în componenta copil.
  Dacă în componenta copil implementăm și interfața "OnChanges" avem acces la metoda ngOnChanges care se apelează de fiecare dată când o proprietate de tip @Input se modifică în componenta părinte. De exemplu:

```
ngOnChanges(changes: SimpleChanges): void {
   if (changes['schoolClass']) {
     this.loadStudentsByClass();
     if(this.schoolClass)
       this.loadCategoriesByClassNumber(this.schoolClass?.classNumber);
   }
   if (changes['mode']) {
     console.log("view mode changed");
   }
 }
```

- Comunicare copil-părinte (event binding): Componentele copil pot emite evenimente pentru a notifica componentele părinte despre anumite acțiuni sau modificări. Componenta copil definește evenimente personalizate folosind decoratorii @Output și EventEmitter, iar componenta părinte se leagă de aceste evenimente folosind sintaxa de legare a evenimentelor. Când evenimentul dorit are loc în componenta copil, acesta emite evenimentul, iar componenta părinte îl poate gestiona cu o metodă corespunzătoare.
  Astfel de comunicare există între componentele "add-grade" și "students", students fiind de această dată componenta părinte. În componenta add-grade am definit următoarele evenimente:

  @Output() removeForm = new EventEmitter<void>();
  @Output() reloadStudentGrades = new EventEmitter<void>();
  this.reloadStudentGrades.emit();
  this.removeForm.emit();

  removeForm este emis atunci când utilizatorul apasă pe un buton de cancel, renunțând la adăugarea unei note, fiind astfel necesară îndepărtarea formularului din componenta părinte.

  *<app-add-grade *ngIf="showAddGradeForm" (reloadStudentGrades)="gradeAdded()" (removeForm)="hideComponent()"....>*

  Acesta este felul în care se leagă evenimentele din componenta copil la o metodă din componenta părinte.
- Comunicarea componentelor care nu au o relație de tip părinte-copil (serviciu partajat): Când două sau mai multe astfel de componente trebuie să comunice între ele direct, un serviciu partajat poate fi folosit ca mediator. Componentele pot injecta serviciul partajat și pot folosi metodele sau proprietățile acestuia pentru a face schimb de date și a-și coordona comportamentul.
  Exemple pentru acest tip de comunicare sunt componentele în care se injectează mai multe servicii.

3.5 Înregistrare și autentificare

3.5.1 Tehnica de stocare a parolei

Pentru autentificarea utilizatorilor, în faza inițială, am folosit tehnica de hashing and salting de securizare a parolelor.

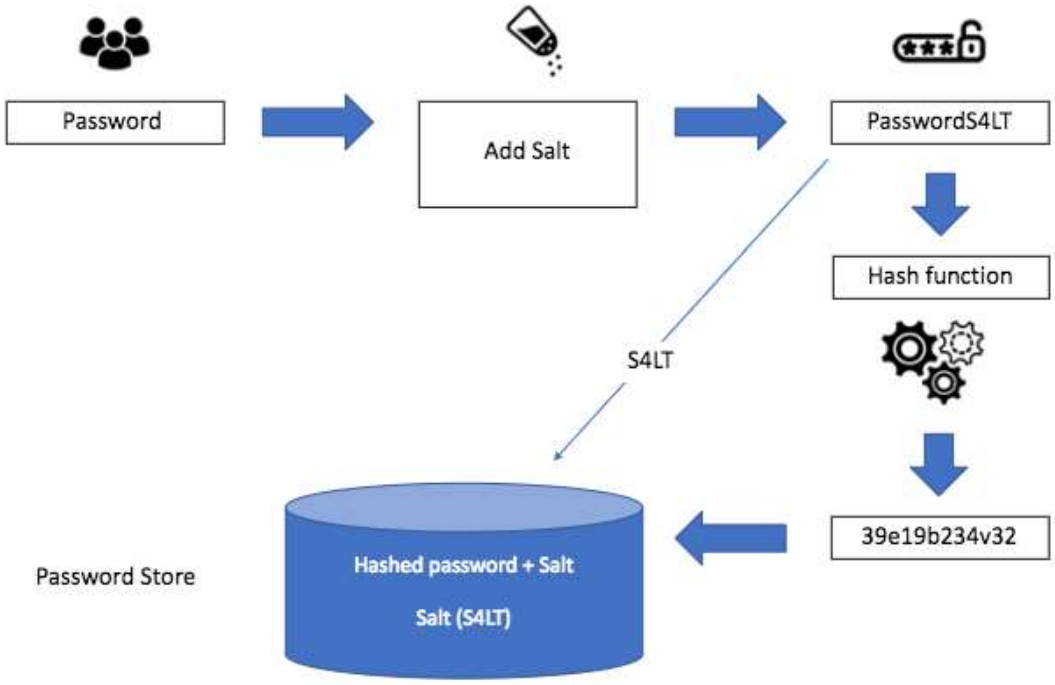

Figura 12 – Procesarea parolei. Preluată din [7]

Hashing: În loc să stochez parolele utilizatorului ca și text, acestea sunt hashuite folosind o funcție hash criptografică. O funcție hash primește o intrare (parolă) și produce un șir de caractere de dimensiune fixă, care este valoarea hashului. Hashul rezultat este un proces unidirecțional, ceea ce înseamnă că nu poate fi inversat pentru a obține parola originală.

Salt: Sarea este un șir aleator de caractere care este generat pentru fiecare utilizator. Este atașată la parolă înainte de hashing. Scopul sării este de a adăuga unicitate și aleatorism parolei codificate. Salting-ul ajută la apărarea împotriva atacurilor, cum ar fi atacurile “rainbow table”, în care o listă de hash-uri precalculate sunt comparate cu hash-urile stocate.

Verificarea parolei: Când un utilizator încearcă să se autentifice, parola introdusă este combinată cu sarea stocată pentru acel utilizator. Același algoritm de hashing este aplicat acestei combinații, producând un nou hash. Acest nou hash este apoi comparat cu parola hash stocată pentru acel utilizator. Dacă hashurile se potrivesc, parola introdusă este considerată validă.

Important de reținut este faptul că validările se fac doar pe server, sarea nu trebuie să părăsească serverul sub nicio formă.

Entitățile utilizatorilor vor conține 3 câmpuri necesare autentificării:

- Username
- PasswordHash
- PasswordSalt

3.5.2 JSON Web Tokens

JWT pot fi folosite pentru autentificarea la un API, nefiind necesară menținerea unei conexiuni permanente între client și API. Clientul face un request, primește un răspuns iar relația se termină până când clientul vrea să facă un nou request.
Tokenurile sunt utile pentru că sunt suficient de mici ca dimensiune încât pot fi transmise cu fiecare request.
Un JWT este practic un string lung compus din 3 părți separate de , . ’:

- Header: conține de obicei două proprietăți și anume: informații referitoare la tipul token-ului și algoritmul folosit pentru criptarea semnăturii.
- Payload: conține afirmațiile(claims) sau declarațiile despre utilizator. Claimurile pot să fie standard(„iss”-issued by, „exp”- expiration time...) sau specifice aplicației(username, rol,nume...)
- Signature: este encriptată de către server folosind o cheie securizată care nu părăsește niciodată serverul. Aceasta este singura parte a token-ului care este encriptată

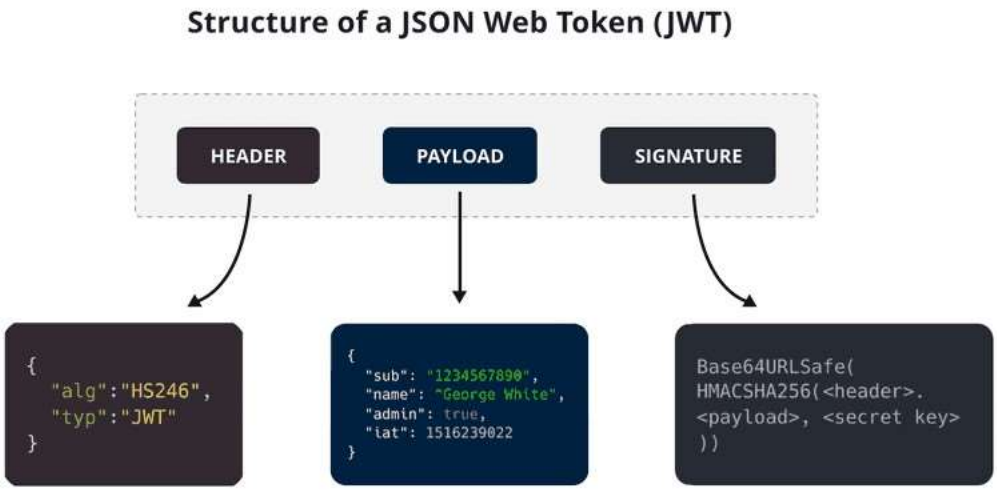

Figura 13 – JSON Web Token. Preluată din [8]

Verificarea token-ului: pe partea de server, serverul validează JWT efectuând următoarele verificări:

- Verificarea semnăturii folosind cheia secretă pentru a se asigura că nu a fost modificată.
- Verificarea timpului de expirare (dacă există o cerere de expirare) pentru a se asigura că token-ul este încă valabil.
- Verificarea oricăror alte revendicări relevante pentru a se asigura că îndeplinesc criteriile cerute.

Funcționarea autentificării folosind Token:

- Utilizatorul vrea să se autentifice și trimite un request către server care conține username-ul și parola
- Server-ul validează datele utilizatorului, dacă acestea sunt valide, crează un token și îl returnează clientului.
- Clientul stochează token-ul primit, de obicei în memoria browserului, pentru a îl putea trimite cu următoarele sale request-uri (de obicei este transmis într-un header al request-ului)
- Server-ul verifică token-ul și trimite răspunsul la request-uri dacă token-ul este valid.

Funcția de creare a token-ului folosită:

```
public string CreateToken(AppUser user)
    {
       var claims = new List<Claim>
       {
          new Claim(JwtRegisteredClaimNames.NameId, user.Id.ToString()),
          new Claim(JwtRegisteredClaimNames.UniqueName, user.UserName)
       };

       var creds = new SigningCredentials(_key,SecurityAlgorithms.HmacSha512Signature);

       var tokenDescriptor = new SecurityTokenDescriptor
       {
          Subject = new ClaimsIdentity(claims),
          Expires = DateTime.Now.AddDays(7),
          SigningCredentials = creds
       };

       var tokenHandler = new JwtSecurityTokenHandler();

       var token = tokenHandler.CreateToken(tokenDescriptor);
       return tokenHandler.WriteToken(token);
    }
```

Trebuie instalat pachetul System.IdentityModel.Tokens.Jwt pentru SymmetricSecurityKey (aceeași cheie este folosită atât pentru encriptare, cât și pentru decriptare) și JwtSecurityTokenHandler()

Cheia este salvată în configurația aplicației, pentru a avea acces la ea trebuie sa injectăm, ca și parametru în constructor, un obiect de tip IConfiguration și să o prelucrăm de fiecare dată cand se instanțiază clasa TokenService de unde se apelează metoda CreateToken.

O parte din funcția de înregistrare a unui nou user:

```
using var hmac = new HMACSHA512();
user.UserName = registerDto.Username.ToLower();
user.PasswordHash = hmac.ComputeHash(Encoding.UTF8.GetBytes(registerDto.Password));
user.PasswordSalt = hmac.Key;

 _context.Users.Add(user);
```

```
 await _context.SaveChangesAsync();

return new UserDto{
        Username = user.UserName,
        Token = _tokenService.CreateToken(user)
};
```

3.6 Funcționalitățile profesorilor

3.6.1 Selectarea claselor si vizualizarea detaliată a studenților

Una dintre funcționalitățile esențiale ale aplicației este cea de vizualizare a informațiilor referitoare la studenți. Pentru aceasta utilizatorul cu rol de profesor are la dispoziție pagina de clase. Când ajunge pe această pagină el are 2 variabile de selectat:

- Clasa a căror studenți vrea să îi vadă
- Care dintre informații referitoare la studenți vrea să le vadă

În momentul inițializării componentei “classes” prin intermediul serviciului “classService” se face un request către server care va returna doar clasele la care profesorul autentificat predă.

```
getProfessorClasses() {
   return this.http.get<SchoolClass[]>(this.baseUrl + 'schoolClass/professorId');
  }
```

Odată încărcate clasele, acestea sunt afișate sub forma unor butoane pe care profesorul poate să le apese pentru selectarea clasei dorite.

Pentru selectarea opțiunilor de vizualizare, profesorul are la dispoziție un grup de 3 butoane:

- Student Details pentru vizualizarea detaliilor personale ale studenților
- Student Grades pentru vizualizarea notelor și a notelor pe categorii
- Attendance pentru vizualizarea absențelor studenților

După ce profesorul selectează o clasă, pe pagină se inserează o nouă componentă numită “students” care primește ca și date de intrare clasa selectată și modul de vizualizare (ce anume

vrea profesorul să vadă, default este ‘details’). Această componentă va afișa studenții clasei selectate sub forma unui tabel în funcție de opțiunea selectată.

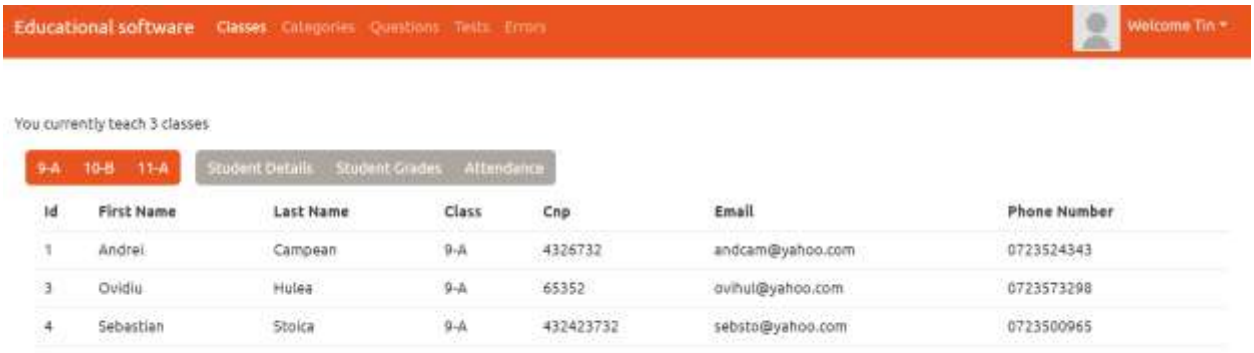

Figura 14 – Detalii personale studenți

Acesta este un exemplu de reprezentare a unei clase cu opțiunea de vizualizare “Student Details”.

Figura 15 – Afișare note studenți

Odată cu selectarea opțiunii “Student Grades”, pe lângă tabelul cu notele studenților, apar încă 3 butoane:

- Add Grade pentru adăugarea unei note în câmpul “Grades“
- Add Category Grade pentru adăugarea unei note la o anumită categorie
- Create Test pentru crearea unor teste

Acestea vor fi descrise mai detaliat în următoarele capitole.

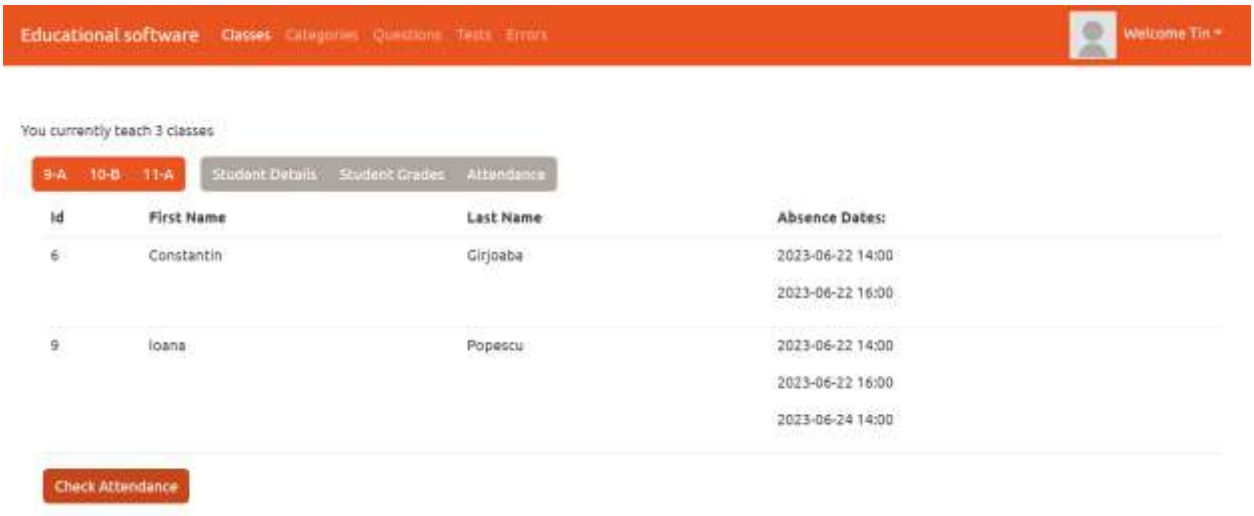

Figura 16 – Afișare absențe studenți

La fel ca în cazul afișării notelor, opțiunea de absențe adaugă un buton suplimentar pentru verificare prezenței.

Orice modificare efectuată în componenta “classes” (clasă selectată sau opțiune de vizualizare) va declanșa automat evenimentul “ngOnChanges” din componenta “students” care va reîncărca studenții dacă a fost schimbată clasa selectată sau va schimba doar metoda de vizualizare.

```
ngOnChanges(changes: SimpleChanges): void {
  console.log("ngOnChanges");
  if (changes['schoolClass']) {
    this.loadStudentsByClass();
    this.selectedIds = [];
    if(this.schoolClass)
      this.loadCategoriesByClassNumber(this.schoolClass?.classNumber);
  }
  if (changes['mode']) {
    console.log("view mode changed");
  }
 }
```

Pe partea de backend pentru această funcționalitate am creat mai multe metode și Dto-uri. Pentru a nu face multiple request-uri către server. Am creat câte o metodă pentru returnarea detaliilor studenților, returnarea notelor și returnarea absențelor unei clase întregi cu toate câmpurile necesare. Trimiterea notelor unei clase către client se face sub forma unei liste de obiecte de tip "StudentGradesDto" care conține următoarele câmpuri:

- Id-ul studentului
- Prenume
- Nume
- Listă cu notele
- Un dicționar care va conține ca și chei categoriile la care studentul a primit note și ca valori o listă cu respectivele note

Metoda din repository care crează și returnează această listă este următoarea:

```
public async Task<List<StudentsGradesDto>> GetStudentsGradesByClassId(int id)
    {
        var students = await _context.Students.Where(s => s.SchoolClassId ==
id).ToListAsync();
        List<StudentsGradesDto> studentsGrades = new List<StudentsGradesDto>();
        foreach (var s in students)
        {
            var sGrades = new StudentsGradesDto();
            sGrades.Id = s.Id;
            sGrades.FirstName = s.FirstName;
            sGrades.LastName = s.LastName;
           sGrades.Grades = await _context.Grades.Where(g => g.StudentId == s.Id).Select(g
=> g.TestGrade).ToListAsync();
            sGrades.CategoryGrades = new Dictionary<int, List<int>>();
            var categoryGrades = await _context.CategoryGrades.Where(g => g.StudentId ==
s.Id).ToListAsync();

            if (categoryGrades != null)
            {
                foreach (var catGrade in categoryGrades)
                {
                    if (sGrades.CategoryGrades.ContainsKey(catGrade.CategoryId))
```

```
                {
                    sGrades.CategoryGrades[catGrade.CategoryId].Add(catGrade.Grade);

                }
                else
                {
                    sGrades.CategoryGrades.Add(catGrade.CategoryId, new List<int> {
catGrade.Grade });
                }
            }
            studentsGrades.Add(sGrades);
        }
    }
    return studentsGrades;
}
```

3.6.2 Vizualizarea, filtrarea si adăugarea categoriilor

Am creat o pagină pentru a oferi profesorilor posibilitatea de a vizualiza categoriile de întrebări deja existente sau de a adăuga una nouă. Pentru vizualizare au și opțiunea de a selecta doar categoriile corespunzătoare unei anumite clase.

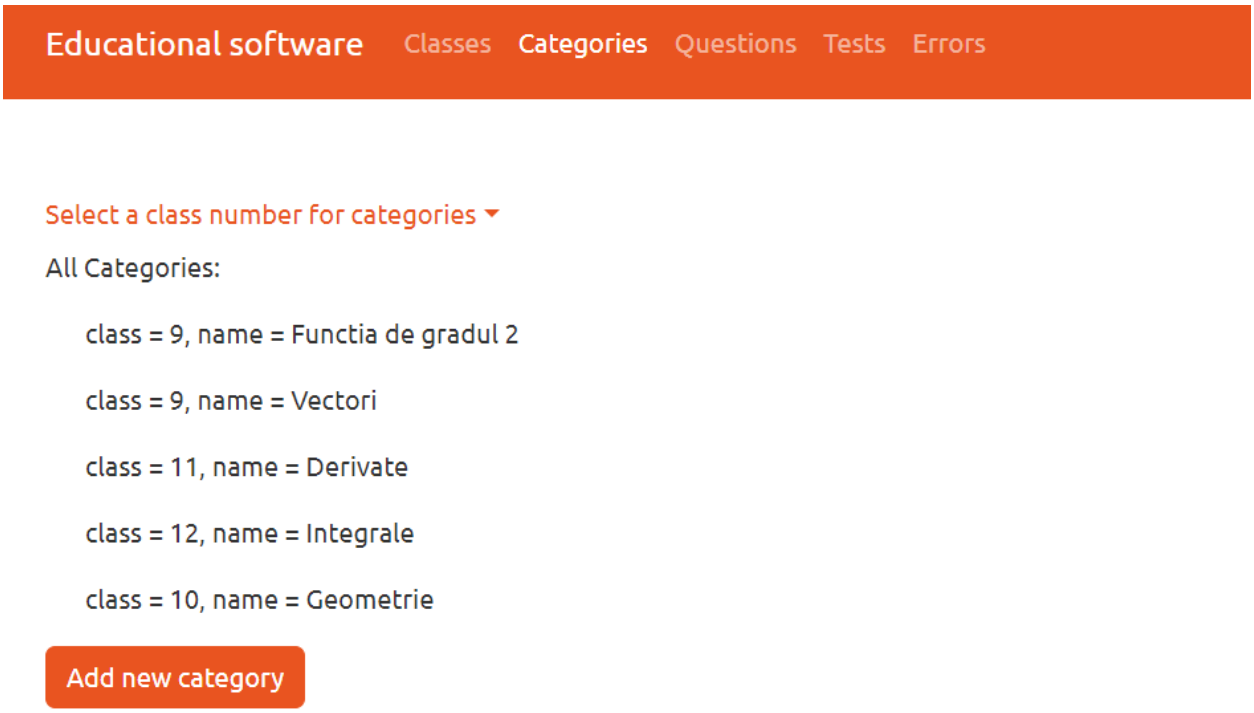

Figura 17 – Pagină categorii

Pentru selectarea clasei se face click pe butonul de select și se alege una dintre clase.
Dacă profesorul dorește adăugarea unei noi categorii, trebuie doar să apese butonul “Add new category”, moment în care va apărea un formular pentru adăugare. Formularul va conține doar două câmpuri, unul pentru numele categoriei și celălalt pentru numărul corespunzător clasei căreia vrea să îi adauge categoria.

Pentru această funcționalitate am folosit niște request-uri simple, nefiind nevoie de prea multe operații nici pe partea de frontend nici pe server.

```
[HttpPost]
    public async Task<ActionResult> AddCategory(CreateCategoryDto categoryDto)
    {
        var category = new Category
        {
            ClassNumber = categoryDto.ClassNumber,
            Name = categoryDto.CategoryName
        };
        _categoryRepository.AddCategory(category);

        if(await _categoryRepository.SaveAllAsync())
            return Ok("Category added successfully");
        return BadRequest("Failed to add category");
    }
```

Aceasta este metoda din controller-ul de categorii responsabilă de crearea unei noi categorii.

3.6.3 Vizualizarea, filtrarea, sortarea si adăugarea întrebărilor

Aspectul teoretic al acestei funcționalități a fost descris în mare parte în capitolul 3.4.6. Pagina de vizualizare, și adăugare a întrebărilor este necesară profesorilor atât pentru a putea verifica actualitatea și corectitudinea întrebărilor din baza de date, cât și pentru a nu introduce aceeași întrebare de mai multe ori.

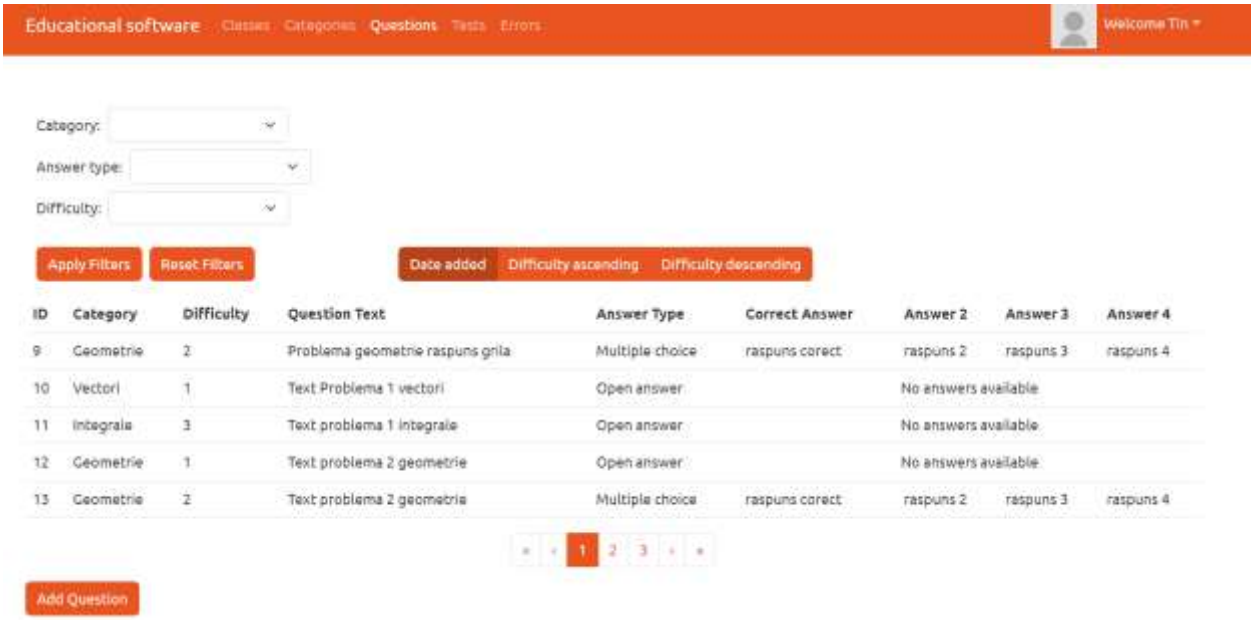

Figura 18 – Pagină întrebări

Pe lângă opțiunile de filtrare și ordonare, mai există în partea de jos niște butoane responsabile de schimbarea paginii.

În momentul în care un profesor vrea să introducă o nouă întrebare, apasă pe butonul “Add Question” și se inserează o nouă componentă numită “questions-form” în pagină.

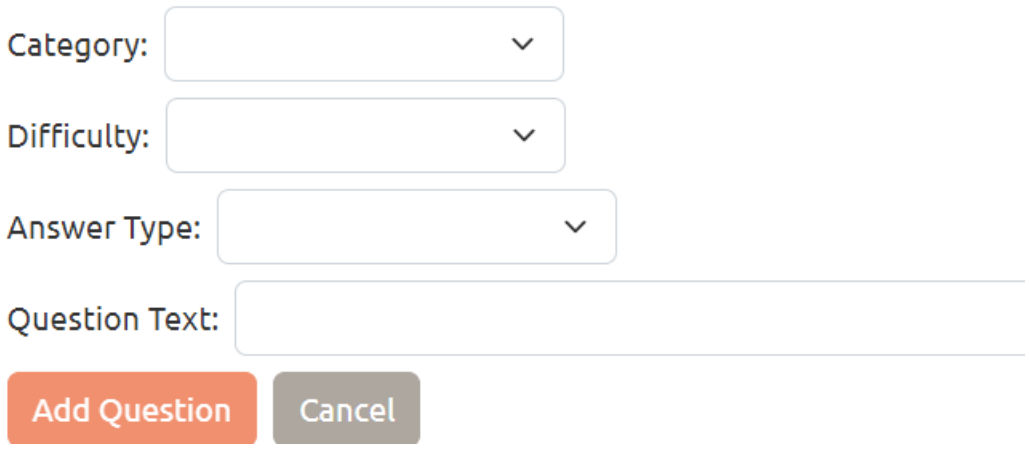

Figura 19 – Formular adăugare întrebare nouă

Aceasta primește ca și date de intrare lista de categorii și poate să emită 2 evenimente către componenta părinte. Un eveniment pentru eliminarea formularului din pagină și unul să semnalizeze componentei părinte că o întrebare a fost aduăugată și este nevoie de actualizarea listei de întrebări.

Primele trei câmpuri sunt de tipul select pentru a evita transmiterea unor date eronate către server. Dacă se selectează ca și tipul răspunsului “Multiple choice” mai apar patru câmpuri suplimentare care reprezintă răspunsul corect și respectiv celelalte variante de răspuns.

```
if(this.questionForm.get('answerType')?.value === '1')
  {
   const question = {
    categoryId: this.questionForm.get('categoryId')?.value,
    difficulty: this.questionForm.get('difficulty')?.value,
    answerType: this.questionForm.get('answerType')?.value,
    text: this.questionForm.get('text')?.value,
    correctAnswer: this.answersForm.get('correctAnswer')?.value,
    answer2: this.answersForm.get('answer2')?.value,
    answer3: this.answersForm.get('answer3')?.value,
    answer4: this.answersForm.get('answer4')?.value
   }
   this.questionService.addQuestion(question).subscribe({
    next: _ => this.toastr.success('Question added successfully')
   });
  }
  else
  {
   const question = {
    categoryId: this.questionForm.get('categoryId')?.value,
    difficulty: this.questionForm.get('difficulty')?.value,
    answerType: this.questionForm.get('answerType')?.value,
    text: this.questionForm.get('text')?.value
   }
   this.questionService.addQuestion(question).subscribe({
    next: _ => this.toastr.success('Question added successfully')
   });
  }
  this.questionAdded.emit();
```

}

Aceasta este funcția care crează un obiect ce urmează să fie trimis către server pentru adăugarea unei întrebări în baza de date.

Pentru a afișa în același tabel și răspunsurile întrebărilor, în cazul celor cu răspuns multiplu, atunci când returnez întrebările de pe server trebuie să atașez și răspunsurile aferente.

```
List <Answer> answers = new List<Answer>();
foreach(var question in questions){
   if(question.AnswerType == 1 && question.Id>=9)
    answers.AddRange(await _answerRepository.GetAnswersByQuestionId(question.Id));
}
QuestionsWithAnswersDTO response = new QuestionsWithAnswersDTO
{
   Questions = questions,
   Answers = answers
};
```

Varianta curentă a aplicației conține doar întrebări cu răspuns deschis sau întrebări cu răspuns multiplu dar cu o singură variantă corectă.

3.6.4 Adăugare note

O altă funcționalitate esențială pentru profesori este adăugarea notelor care se face din pagina de "Student Grades" apăsând pe butonul de "Add Grade" sau "Add Category Grade". Din componenta "students" se inserează o componentă numită "add-grade".

```
<app-add-grade *ngIf="showAddGradeForm" (reloadStudentGrades)="gradeAdded()" (removeForm)="hideComponent()" [mode]="addGradeMode" [categories]="categories" [students]="students"></app-add-grade>
```

Ea primește ca și date de intrare de la componenta părinte lista de categorii, modul de adăugare (adăugare notă sau adăugare notă la o categorie) și lista de studenți. Componenta crează următorul formular pentru adăugarea notei:

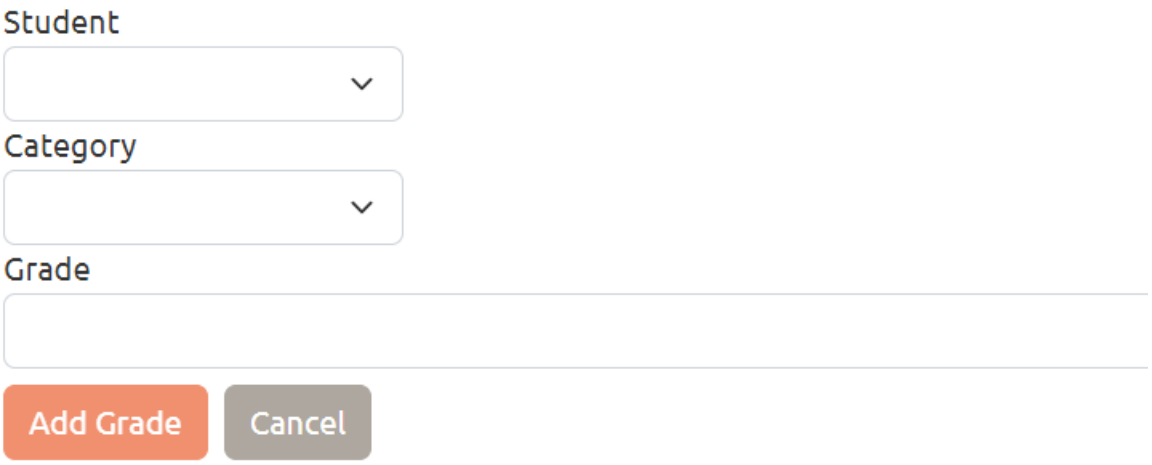

Figura 20 – Formular adăugare notă la o categorie

Formularul conține două câmpuri de tip select pentru a selecta studentul căruia acordăm nota și categoria acesteia și un câmp pentru notă. Opțiunile câmpurilor de tip select sunt preluate din datele de intrare a componentei.

Formularul pentru adăugarea unei note normale este asemănător, singura diferență fiind lipsa câmpului “Category”.

Pentru fiecare tip de formular am făcut câte un request către controller-ul aferent notelor. În controller, se verifică dacă nota introdusă de profesor este validă (între 0 și 10) și se adaugă obiectului de tip notă ora la care a fost acordată.

```
[HttpPost]
    public async Task<ActionResult> AddGrade(GradeParams gradeParams)
    {
        if (gradeParams.Grade > 0 && gradeParams.Grade <= 10)
        {
            var fullGrade = new Grade
            {
                StudentId = gradeParams.StudentId,
                TestGrade = gradeParams.Grade,
                DateReceived = DateTime.Now
            };
```

```
            _gradeRepository.AddGrade(fullGrade);
        }
        else return BadRequest("Not a valid grade!");

        if (await _gradeRepository.SaveAllAsync())
            return Ok("Grade added succesfully");
        return BadRequest("Failed to add grade");
    }
```

3.6.5 Verificare prezență

Verificarea prezenței studenților se face din componenta de clase, selectând opțiunea de vizualizare aferentă prezenței. În partea de jos a paginii apare butonul “Check attendance” iar în urma apăsării acestuia se adaugă o coloană nouă în tabelul studenților care conține câte un checkbox pentru fiecare student în parte. Se bifează toți studenții care sunt absenți și se apasă pe butonul “Add absences”.

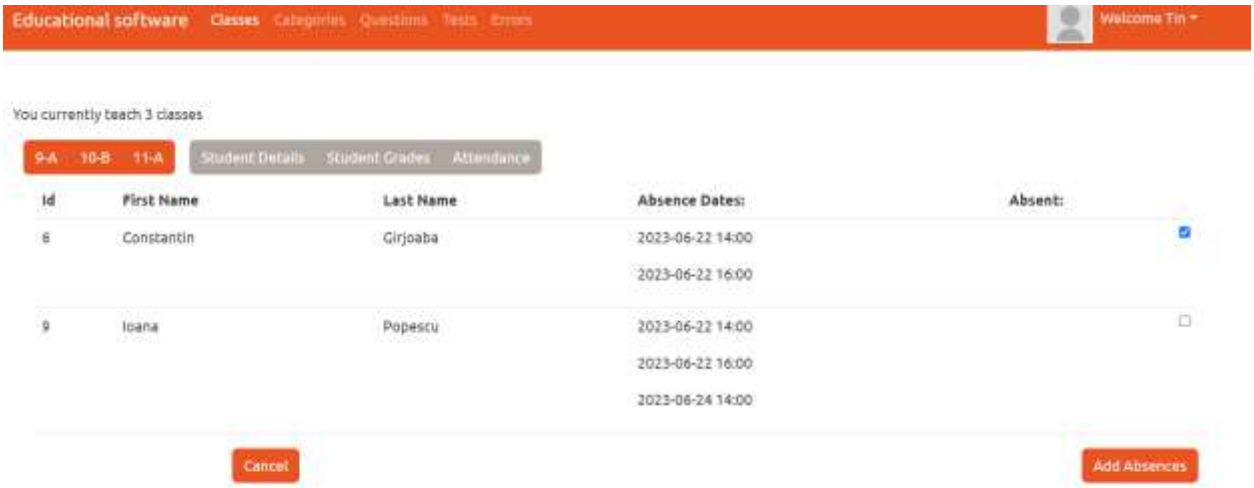

Figura 21 – Adăugare absențe

În urma bifării unui checkbox, se adaugă id-ul respectivului student într-un array care conține id-urile studenților absenți. Acest array este trimis ca parametru în momentul efectuării request-ului către server. După ce am adăugat absențele, tabelul se actualizează și se golește array-ul de id-uri selectate. Acesta se mai golește și dacă apăsăm pe butonul de “Cancel” sau la schimbarea clasei selectate.

Metoda care efectuează efectiv adăugarea absențelor este următoarea:

```
public void AddStudentsAttendance(List<int> studentIds)
    {
       foreach(int id in studentIds) {
          Attendance a = new Attendance{
             StudentId = id,
             PresenceDate = DateTime.Now
          };
          _context.Attendance.Add(a);
       }
    }
```

3.6.6 Generare, corectare si vizualizare teste

Generarea testelor pornește din componenta “students”, modul de vizualizare “Student Grades”. Aici pe lângă opțiunile de adăugare a notelor, mai este și butonul “Create Test” care va adăuga, ca și în cazul absențelor, o nouă coloană tabelului pentru a putea selecta studenții cărora vrem sa atribuim câte un test.

Odată ce s-au selectat studenții, se apasă pe butonul “Generate Tests” care ne va duce la o nouă componentă numită „create-test”. Aceasta primește ca și date de intrare array-ul de id-uri selectate și lista de categorii aferente clasei selectate.

Navigarea spre componenta unde se configurează testul se face în următoarea metodă:

```
createTests() {
   this.testMode = false;
   const categoriesArray = JSON.stringify(this.categories);
   const params = {selectedIds: this.selectedIds.join(','), categories: categoriesArray}
   this.router.navigate(['/create-test'], {queryParams: params});
   console.log(this.selectedIds);
   this.selectedIds = [];
 }
```

Pentru a putea trimite lista de categorii, trebuie să o convertim la un format posibil de transmis, în cazul acesta JSON string. Transformăm și id-urile în string, despărțite de câte o virgulă și le trimitem către componenta “create-test”.

În momentul inițializării componentei, trebuie să preluăm datele primite și să le convertim înapoi la formatul cu care putem lucra.

```
this.route.queryParams.subscribe(
    params => {
      const idsArray = params['selectedIds']
      if(idsArray) {
        this.selectedIds = idsArray.split(',').map(Number);
      }
      const categoriesArray = params['categories']
      if(categoriesArray) {
        this.categories = JSON.parse(categoriesArray)
      }
    }
  )
```

După ce am obținut datele, inițializăm două formulare necesare generării testelor.

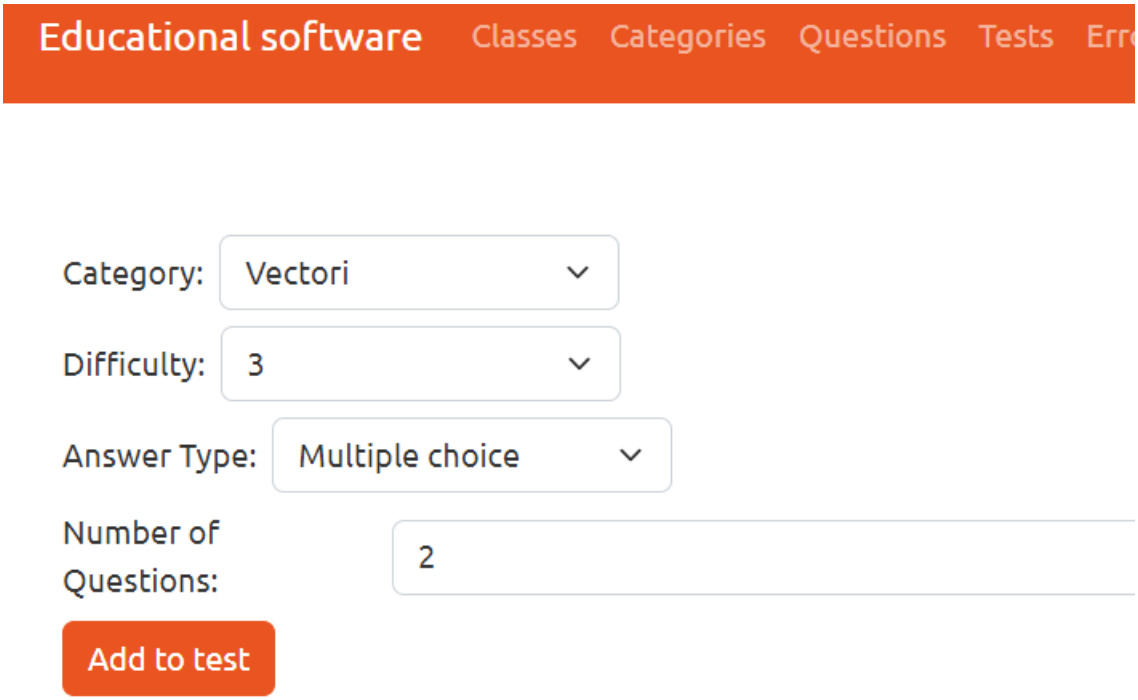

Figura 22 – Formular adăugare configurație test

În partea din stânga a paginii avem un formular pentru adăugarea unei configurații pentru test. O configurație conține:

- Categoria întrebării
- Dificultatea întrabării
- Tipul răspunsului
- Numărul de probleme

Unui test putem adăuga mai multe configurații. Una pentru fiecare tip de probleme pe care dorim să le introducem.

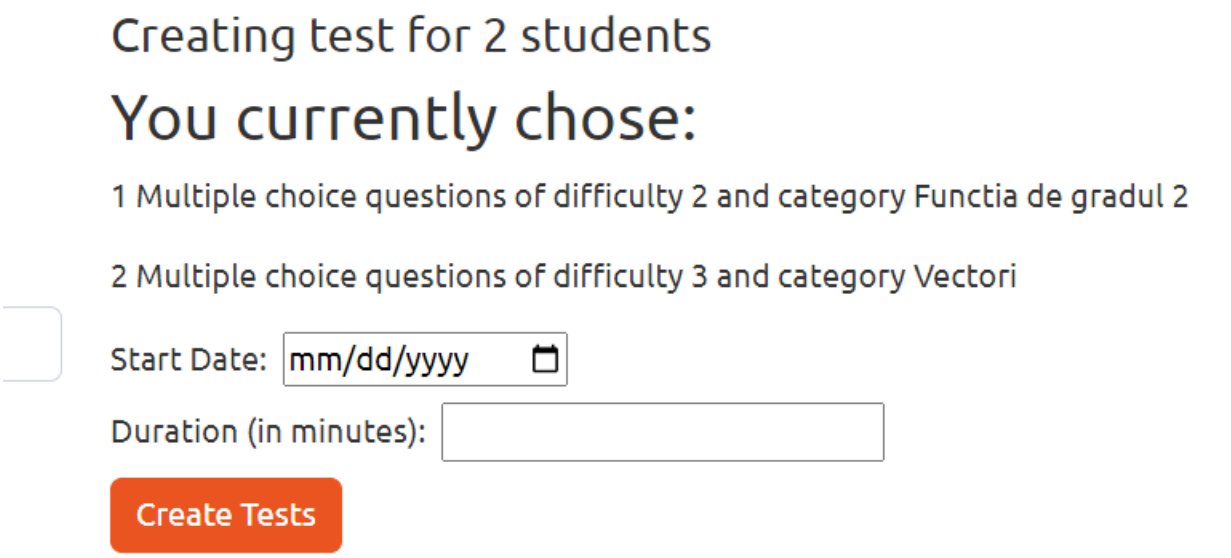

Figura 23 – Informații și formular final creare test

În partea din dreapta a paginii avem informații referitoare la numărul de teste pe care urmează să le generăm, configurațiile adăugate testului deja și un formular care va adăuga testului ora de începere și durata.

Apăsarea butonului “Create Tests” va trimite către server lista de studenți cărora atribuim câte un test, lista de configurații, ora de început și durata testelor. Serverul va crea câte un test pentru fiecare student respectând cerințele impuse.

Vizualizarea și corectarea testelor se va face din componenta “tests”. Vizualizarea unui test se face pe baza id-ului. Se afișează câte un rând pentru fiecare întrebare a testului.

Corectarea se va face tot pe baza id-ului testului. În cazul testelor tipărite și susținute fizic, profesorul introduce testul id-ului apoi răspunsurile la întrebări pentru cele cu răspuns multiplu sau un punctaj pentru întrebările cu răspuns deschis. După corectarea testelor, se calculează si actualizează notele studenților.

3.7 Tratarea erorilor

Crearea unui controller în care implementăm metode de acțiune care vor returna erori.
Dacă nu suntem în development mode, nu vom primi text ajutător atunci când apare o eroare.
Cel mai bine e să tratezi excepțiile la cel mai înalt nivel pentru că excepțiile vor fi transmise în sus în lanțul de middleware până când sunt tratate.

Crearea unei clase ApiException cu proprietățile StatusCode, Message, Details

Crearea unui middleware pentru tratarea excepțiilor:

- Crearea unei clase ExceptionMiddleware în care avem un RequestDelegate (reprezintă următorul middleware la care trebuie să meargă request-ul), un Ilogger (pentru a vedea output-ul în consolă) și un IHostEnvironment pentru a vedea dacă suntem în production sau development mode. Singurul esențial este RequestDelegate-ul.

  ```
  public ExceptionMiddleware(RequestDelegate next, ILogger<ExceptionMiddleware> logger, IHostEnvironment env)
      {
         _env = env;
         _logger = logger;
         _next = next;
      }
  ```

- În această clasă implementăm o metodă numită obligatoriu „InvokeAsync” (framework-ul are nevoie de această metodă pentru a decide se întâmplă în continuare în lanțul de

middleware). Ea primește ca și parametru un HttpContext pentru a avea acces la request-ul Http care trece momentan prin lanțul de middleware.

```
public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, ex.Message);
            context.Response.ContentType = "application/json";
            context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;
            var response = _env.IsDevelopment()
                ? new ApiException(context.Response.StatusCode, ex.Message, ex.StackTrace?.ToString())
                : new ApiException(context.Response.StatusCode, ex.Message, "Internal Server Error");
            var options = new JsonSerializerOptions{PropertyNamingPolicy = JsonNamingPolicy.CamelCase};
            var json = JsonSerializer.Serialize(response,options);
            await context.Response.WriteAsync(json);
        }}
```

Excepțiile vor trece prin toate middleware-urile și dacă nu sunt tratate ele ajung in blocul catch și le tratăm noi la cel mai înalt nivel. Prima linie din blocul catch afișează eroarea în terminalul VS Code. Următoarele 2 linii de cod specifică formatul răspunsului și setează StatusCode-ul răspunsului. Apoi se verifică modul în care suntem și dacă suntem în modul de development se crează un nou ApiException care va conține și stackTrace-ul excepției pentru detalii suplimentare, dacă suntem în modul de producție se crează un ApiException dar fără detalii suplimentare referitoare la excepție. Ultimele linii modifică formatul răspunsului pentru a putea fi transmis către client, ele sunt necesare deoarece nu suntem într-un controller (acolo acestea sunt făcute în mod implicit).

- Pentru a folosi middleware-ul el trebuie inclus în vârful pipeline-ului de request-uri ca și middleware. Acest lucru se face în fișierul “Program.cs”

```
app.UseMiddleware<ExceptionMiddleware>();
```

Pe partea de client:

Doar cu scopul de a putea testa și vizualiza cu ușurință funcționarea tratării excepțiilor, am introdus o pagină de erori care nu necesită autentificare.

Figura 24 – Testare erori

Pentru utilizarea funționalității am creat un interceptor care permite interceptarea unui request fie merge spre server sau când revine de la server.

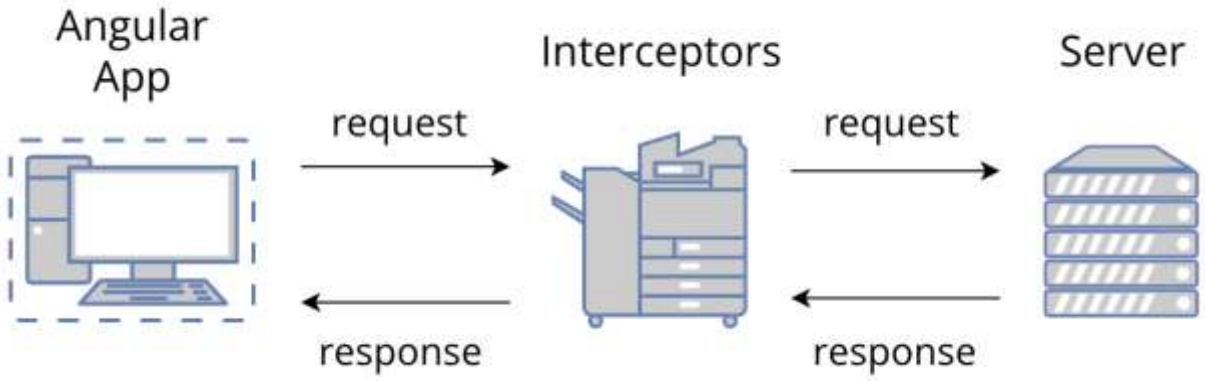

Figura 25 – Interceptor request-uri. Preluată din [10]

Interceptor-ul se crează din lina de comandă folosind comanda *"ng g interceptor nume_interceptor"*. El implementează automat interfața HttpInterceptor și metoda "intercept". În constructor injectăm un Router pentru a putea să redirecționăm utilizatorul către altă pagină dacă este cazul și un ToastrService pentru a putea afișa utilizatorului, sub forma unui pop-up, eroarea.

Parametrii metodei intercept sunt un HttpRequest și un HttpHandler

```
intercept(request:HttpRequest<unknown>,next:HttpHandler)
:Observable<HttpEvent<unknown>> {
  return next.handle(request).pipe(
    catchError((error: HttpErrorResponse) => {
      if (error) {
        switch (error.status) {
          case 401:
```

```
        this.toastr.error('Unauthorized', error.status.toString())
        break;
      case 404:
        this.router.navigateByUrl('/not-found');
        break;
      ....
    }
    throw error;
```

Funcția pipe() permite modificarea sau adăugarea unor filtre sau operații în lanț înainte de afișarea lor. Una dintre operații este catchError iar în interiorul ei specificăm ce vrem să facem în funcție de eroarea primită.

4 Concluzii si dezvoltări ulterioare

4.1 Concluzii

Software-ul educațional creat reprezintă o unealtă utilă profesorilor. El eficientizează modul de creare și corectare a testelor, oferă posibilitatea de a urmări și contoriza absențele studenților, prezintă date suplimentare referitoare la niveul de instruire al studenților. Aplicația nu este în forma finală încă, dar până la începutul următorului an școlar va fi pregătită de testare în condiții reale de învățământ. Domeniul de aplicabilitate este deocamdată de o singură materie, putând fi extins cu ușurință la multiple materii.

4.2 Dezvoltări ulterioare

Pentru ca aplicația să poată fii folosită la capacitatea ei maximă, în primă fază doar de către părinții mei pentru testare, mi-am propus ca până la începerea următorului an școlar să mai adaug următoarele funcționalități:

- Adăugarea utilizatorului de tip student care să susțină testele online și să își vizualizeze numărul de absențe, notele și istoricul testelor.
- Adăugarea unui utilizator de tip administrator care să verifice și să aprobe autentificarea celorlalți utilizatori pentru a restricționa utilizarea aplicației de către persoane neautorizate.
- Posibilitatea de a adăuga poze întrebărilor atunci când sunt necesare (Grafice de funcții, diagrame, tabele, etc)
- Adăugarea mai multor răspunsuri corecte la o singură întrebare și posibilitatea de a oferi punctaje parțiale în cazul în care se bifează doar o parte din răspunsul corect.
- Extinderea colecției de probleme pentru a avea teste cât mai diversificate

Bibliografie

[1] Florea, Adrian, et al. "Enhanced Learning and Educational Management through Online Collaborative Technologies." *J. Digit. Inf. Manag.* 9.1 (2011): 33-42.

[2] Anghel, Traian, et al. "WEB-BASED TECHNOLOGIES FOR ONLINE E-LEARNING ENVIRONMENTS." *eLearning & Software for Education* (2011).

[3] Florea, Adrian, et al. "Online collaborative education management tool." *Proceedings of the 5th International Conference on Virtual Learning (2010-Towards a Learning and Knowledge Society-2030), Targu Mures, Romania*. 2010.

[4] Anghel, Traian, Adrian Florea, and Delilah Florea. "IMPROVING COURSE INTERACTION AND MANAGEMENT WITH TESTING ASSISTANT." *eLearning & Software for Education* (2010).

[5] Florea, Adrian, Traian Anghel, and Delilah Florea. "Testing Assistant–An Interactive Training Tool for Evaluating Students Knowledge." *International Conference on Embedding Innovation in Teaching and Management of Higher Education (ICITM 2009), Shah Alam, Selangor, Malaysia*. 2009.

[6] Create ASP.NET Core Middlewares for Reusable and Modular Code, 2020. Disponibil online https://www.dotnetnakama.com/blog/asp-dotnetcore-create-middleware/

[7] Jason Jung, „What are Salted Passwords and Password Hashing?", 2021. Disponibil online https://www.okta.com/au/blog/2019/03/what-are-salted-passwords-and-password-hashing/

[8] Rishabh Poddar, „What is a JWT? Understanding JSON Web Tokens", 2020. Disponibil online https://supertokens.com/blog/what-is-jwt

[9] Sandeep Singh Shekhawat, „CRUD using the Repository Pattern in MVC". Disponibil online https://www.c-sharpcorner.com/UploadFile/3d39b4/crud-using-the-repository-pattern-in-mvc/

[10] Hugo Noro, „Angular Interceptors to Manage HTTP Requests", 2021. Disponibil online https://dev-academy.com/how-to-use-angular-interceptors-to-manage-http-requests/

[11] „Client server model”. Disponibil online client-server-model

[12] .NET Documentation. Disponibil online https://learn.microsoft.com/en-us/dotnet/

[13] Angular Documentation. Disponibil online https://angular.io/docs

[14] Neil Cummings, „Build an app with ASPNET Core and Angular” 2023. Disponibil online: https://www.udemy.com/course/build-an-app-with-aspnet-core-and-angular-from-scratch/