



Universitatea Transilvania Brașov
Facultatea de Matematică și Informatică
Specializarea Informatică

LUCRARE DE LICENȚĂ

Monitorizarea timpului și gestionarea proiectelor
personale

Coordonator

Lect. dr. Aldea Constantin Lucian

Absolvent

Barariu Claudiu

Brașov
2023

Abstract

Această lucrare prezintă dezvoltarea unei aplicații web cu scopul de a monitoriza timpul și de a gestiona proiectele personale. Aplicația oferă o soluție eficientă și ușor de utilizat pentru organizarea timpului petrecut la proiecte și administrarea acestora. Utilizatorii beneficiază de o interfață intuitivă și funcționalități avansate, permițând înregistrarea automată a timpului lucrat și vizualizarea clară a informațiilor despre proiecte. Aplicația utilizează ASP.NET Core pentru backend și Angular pentru frontend, asigurând o experiență modernă și interactivă.

Cuprins

1. Introducere	5
1.1 Motivație	5
1.2 Scopul aplicației.....	5
1.3 Aplicații similare	6
1.4 Structura lucrării.....	8
2. Tehnologii și metodologie	9
2.1 Cadre de lucru	9
2.1.1 Angular	9
2.1.2 ASP.NET Core Web API	12
2.1.3 Microsoft SQL Server	14
2.2 Limbaje de programare	17
2.2.1 TypeScript.....	17
2.2.2 C#	18
2.3 Medii de programare.....	20
2.3.1 Visual Studio Code	20
2.3.2 Visual Studio 2022	20
2.4 Tactici și tehnici de dezvoltare	21
2.4.1 Single page application	21
2.4.2 Comunicarea HTTP	22
2.4.3 JSON Web Token	24
2.4.4 Metoda Code First	26
2.4.5 Repository Pattern	27
3. Structura și dezvoltarea aplicației	28
3.1 Arhitectura aplicației – diagrame UML	29
3.2 Arhitectura interfeței	32
3.2.1 Injectarea dependențelor	32

3.2.2	Module	33
3.2.3	Directive și componente.....	34
3.2.4	Servicii.....	36
3.3	Arhitectura serverului	37
3.3.1	Modele	38
3.3.2	Obiecte de transfer de date(DTO).....	39
3.3.3	Controlere.....	40
3.3.4	Repository	41
3.4	Arhitectura bazei de date	42
3.5	Implementare server	43
3.5.1	Interacțiunea cu baza de date prin ORM	43
3.5.2	Definirea serviciilor REST	46
3.5.3	Autentificare și autorizare cu JWT	48
3.6	Implementare interfata.....	49
3.6.1	Definirea componentelor Angular pentru interfață	49
3.6.2	Gestionarea datelor de la server	51
3.6.3	Logica de rutare și navigare între pagini.....	53
4.	Concluzii.....	55
	Bibliografie.....	56

1. Introducere

1.1 Motivație

Ideea de a dezvolta o aplicație pentru monitorizarea timpului și gestionarea proiectelor personale a luat naștere în urma unei conversații avute cu un prieten apropiat. În timpul discuției, am aflat că el întâmpină dificultăți în ceea ce privește gestionarea timpului petrecut la diverse proiecte. Metoda pe care o utiliza anterior implica pornirea unui cronometru și înregistrarea manuală a timpului lucrat pe un site web.

După o analiză atentă, am observat că această abordare se dovedește a fi ineficientă, consumă mult timp și este frustrantă. Motivul principal este că prietenul meu are de gestionat mai multe proiecte simultan și este dificil să își organizeze eficient timpul lucrat la fiecare proiect în parte. În plus, procesul de pornire a cronometrului, înregistrarea manuală a timpului lucrat și organizarea acestuia pentru fiecare proiect individual necesită efort și atenție sporită.

Astfel, am ajuns la concluzia că o soluție mai potrivită și eficientă ar fi dezvoltarea unei aplicații web personalizate care să permită monitorizarea și gestionarea proiectelor. Prin intermediul acestei aplicații, prietenul meu ar putea să își înregistreze timpul lucrat la fiecare proiect în mod automat, fără a mai fi necesară pornirea manuală a cronometrului sau înregistrarea manuală a datelor.

1.2 Scopul aplicației

Scopul aplicației este de a oferi utilizatorului o modalitate eficientă și ușoară de a-și gestiona timpul lucrat la un proiect, precum și de a-și organiza mai bine proiectele în general. Această aplicație își propune să ofere o soluție simplă și eficientă pentru persoanele care se confruntă cu o multitudine de proiecte și doresc să își monitorizeze și să își gestioneze timpul într-un mod mai eficient.

Prin intermediul unei interfețe plăcute la vedere și ușor de utilizat, utilizatorii vor avea posibilitatea de a monitoriza și organiza proiectele într-un mod mai eficient. Aplicația va oferi o vizualizare clară și concisă a timpului pontat pe fiecare proiect din portofoliul utilizatorului, facilitând astfel evidențierea timpului alocat fiecărui proiect în parte.

Această aplicație se adresează în special studenților, programatorilor, managerilor și oricăror persoane angajate în mediul informatic, economic sau educațional, care se

confruntă cu provocări legate de monitorizarea timpului și gestionarea proiectelor. Prin utilizarea acestei aplicații, utilizatorii vor beneficia de o unealtă utilă în gestionarea proiectelor și în organizarea timpului de lucru.

Aplicația este compusă din două părți distincte: partea de backend și partea de frontend. Partea de backend a fost implementată folosind ASP.NET Core Web API, un cadru de lucru specializat în dezvoltarea serviciilor web de tip API¹. Acesta permite gestionarea și furnizarea datelor necesare aplicației. Pe de altă parte, partea de frontend a fost dezvoltată utilizând Angular, un cadru de lucru gratuit și deschis, dezvoltat de Google. Angular oferă o platformă puternică pentru dezvoltarea interfețelor de utilizator interactive și moderne. În ceea ce privește baza de date, am utilizat Microsoft SQL Server, un sistem de gestionare a bazelor de date relaționale dezvoltat de Microsoft, recunoscut pentru performanță, scalabilitate și securitatea sa.

1.3 Aplicații similare

Aplicațiile pe care le voi menționa mai jos au servit ca o sursă de inspirație în privința designului, precum și în abordarea nevoilor utilizatorului, mai exact în implementarea funcționalităților aplicației pe care o prezint.

Clockify

Clockify¹ este o aplicație de monitorizare a timpului și a productivității, fiind folosită de utilizatori atât individual cât și în echipă. Printre funcționalitățile oferite de această aplicație se numără monitorizarea timpului, gestionarea proiectelor, opțiunea de a genera și vizualiza rapoarte privind activitatea utilizatorului și posibilitatea de a colabora cu alți utilizatori.

¹ Vezi subcapitolul 2.1.2 ASP.NET Core Web API

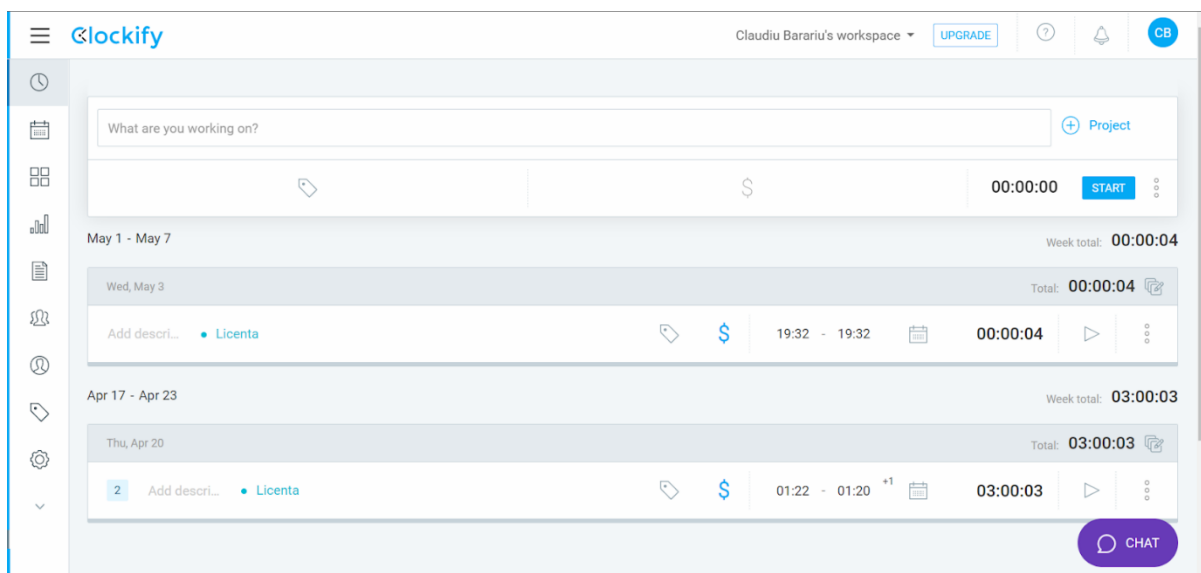


Figura 1.0: Captură de ecran din aplicația Clockify

Toogl Track

Toogl Trackⁱⁱ este o aplicație asemănătoare aplicației Clockify, fiind dezvoltată de compania Toogl, o companie din Estonia. Aplicația oferă servicii precum monitorizarea timpului și posibilitatea de a lucra cu alte persoane la același proiect.

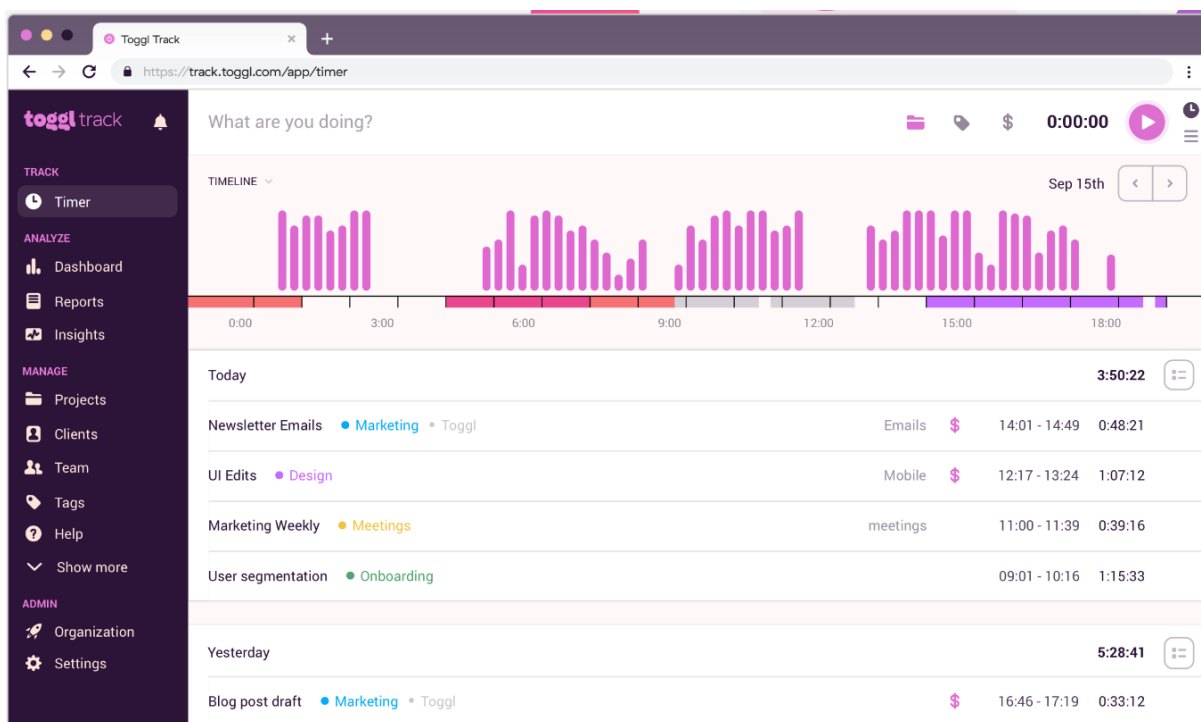


Figura 1.1: Captură de ecran cu aplicația Toogl Tracker

1.4 Structura lucrării

Lucrarea de documentație va conține patru capitole în care voi încerca să acopăr atât partea teoretică, cât și partea practică a aplicației, și la final este și o scurtă bibliografie.

În primul capitol, Introducere, voi prezenta motivația din spatele dezvoltării aplicației, evidențiind necesitatea unei soluții eficiente și ușor de utilizat pentru monitorizarea timpului și gestionarea proiectelor personale. Scopul aplicației este clar definit, iar aplicațiile similare arată diferențele dintre aplicația pe care o prezint și cele existente, ele servind în unele aspecte drept surse de inspirație. De asemenea, voi oferi și o structură a lucrării, pentru ca cititorii să poată avea o privire de ansamblu asupra conținutului acesteia.

În capitolul următor, Tehnologii și metodologie, voi menționa și explica tehnologiile și limbajele de programare pe care le-am utilizat în dezvoltarea aplicației. Ca tehnologii, voi discuta despre Angular, ASP.NET Core Web API, și Microsoft SQL Server. Limbajele de programare de care m-am folosit în această aplicație sunt, pe partea de web, HTML, CSS, TypeScript, iar pentru server am folosit C#. De asemenea, voi prezenta și mediile de programare în care am dezvoltat aplicația, și anume Visual Studio Code și Visual Studio 2022. Pe lângă toate cele menționate voi spune câteva lucruri și despre metodologia de dezvoltare, cum ar fi conceptul de single page application, comunicarea HTTP, JSON Web Token, metoda Code First și Repository Pattern.

În al treilea capitol, Structura și dezvoltarea aplicației, mă voi concentra asupra arhitecturii aplicației și a modului de implementare. Voi descrie arhitectura aplicației, punând în evidență interfața utilizatorului, serverul și baza de date. Voi aborda modul de implementare a aplicației atât în privința interfeței cât și a serverului, menționând câteva noțiuni și despre baza de date. La finalul capitolului voi explica într-un manual cum se utilizează corespunzător aplicația.

În ultimul capitol, Concluzii, voi realiza o sinteză a rezultatelor obținute în cadrul lucrării și voi evidenția importanța aplicației, precum și contribuțiile acesteia.

La final, voi prezenta bibliografia utilizată pentru realizarea lucrării, unde sunt enumerate sursele de informații.

2. Tehnologii și metodologie

2.1 Cadre de lucru

2.1.1 Angular

Angular este un cadru de lucru deschis utilizării gratuite, bazat pe TypeScript, fiind dezvoltat de echipa Angular de la gigantul Google. Angular este o tehnologie relativ nouă, data lansării fiind 14 septembrie 2016. Înainte să apară Angular exista un cadru de lucru numit AngularJS, care era bazat pe JavaScript, fiind dezvoltat tot de compania Google, cu 6 ani înainte de apariția noului cadru de lucru Angular. Angular este o rescriere a cadrului de lucru AngularJS, fiind introduse multe îmbunătățiri, cum ar fi limbajul TypeScript, sintaxă și directive, structura modulară, performanță și optimizare prin concepte precum lazy loading. Angular este utilizat pentru a dezvolta și construi aplicații web de tip single-page, în care interacțiunea utilizatorului se realizează fără a fi necesară încărcarea paginii întregi.

Angular este utilizat la scară largă în întreaga lume, fiind un cadru de lucru apreciat pentru opțiunile avantajoase pe care le oferă, iar documentația și comunitatea mare de utilizatori fac din Angular o unealtă potrivită pentru orice aplicație web, fie că vrei să faci o aplicație în cadrul universitar, educațional, fie că este vorba de o aplicație complexă, cu scopul de a fi utilizată de persoane din întreaga lume. Câteva aplicații și platforme notabile care au la bază Angular sunt: Weather.com, LinkedIn, 9GAG, Vevo, WeTransfer.

Pentru a putea folosi Angular trebuie să am deja cunoștințe de bază despre programarea web în JavaScript, CSS și HTML, fără acestea voi întâmpina dificultăți majore în dezvoltarea aplicațiilor web cu Angular. În rândurile de mai jos voi explica ce trebuie să fac ca să pot folosi Angular pe computerul meu personal.

Înainte de a instala Angular trebuie să mă asigur că am următoarele lucruri instalate:

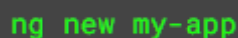
- **Node.js** – este un mediu de execuție apărut în 2009, fiind dezvoltat de Ryan Dahl folosind JavaScript. Este folosit de o comunitate uriașă de programatori la dezvoltarea serverelor aplicațiilor web.
- **npm package manager** – este o unealtă pe care programatorii o folosesc pentru a găsi, descărca, instala și folosi mai ușor module.

Următorul pas este să deschid o fereastră de terminal și să instalez Angular CLI folosind următoarea comandă:

```
npm install -g @angular/cli
```

Figura 2.0: Comandă instalare Angular CLI

După ce am reușit să instalez cele de mai sus pot crea un proiect utilizând comanda următoare:



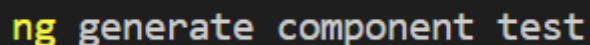
```
ng new my-app
```

Figura 2.1: Comandă creare proiect Angular

O parte foarte importantă din Angular sunt componentele, care sunt niște blocuri ce stau la baza oricărei aplicații Angular, fiind folosite pentru a face interfața utilizator. Componentele ajută la gestionarea și afișarea datelor și sunt responsabile de interacțiunea cu utilizatorul. O componentă Angular este alcătuită din trei fișiere: un fișier TypeScript în care avem o clasă cu partea de logică, un fișier șablon HTML și un fișier de tip CSS pentru stilizare.

Două mari avantaje ale componentelor care le fac să fie atât de utilizate și necesare sunt modularitatea și reutilizabilitatea, ele oferind posibilitatea programatorului de a le folosi împreună cu alte componente și de a crea o interfață mai complexă și plăcută vizual. Componentele au proprietatea de a reacționa la evenimente, cum ar fi apăsarea unei taste de la tastatură sau a unui clic de mouse, și cu ajutorul lor se pot declanșa acțiuni. Este recomandat să se folosească componentele cât de mult posibil într-o aplicație Angular, deoarece ușurează cu mult munca programatorului, ajutând la dezvoltarea unei interfețe plăcută ochiului, ușor de întreținut și ușor de utilizat de către utilizator, și de asemenea, prin folosirea componentelor pot separa și încapsula mai ușor logica aplicației.

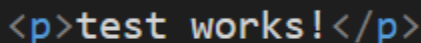
Angular oferă posibilitatea de a genera componente fără conținut printr-o singură comandă, practic îmi generează cele trei fișiere, HTML, CSS, TypeScript, fără să mai fie nevoie să le fac eu manual unul cate unul, în felul acesta aș putea să economisesc destul de mult timp. Comanda pentru generarea unei componente în Angular este următoarea:



```
ng generate component test
```

Figura 2.2: Comandă generare componentă Angular

Mai jos voi arăta un exemplu de componentă:



```
<p>test works!</p>
```

Figura 2.3: Fișier HTML dintr-o componentă Angular

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-test',
  templateUrl: './test.component.html',
  styleUrls: ['./test.component.css']
})
export class TestComponent {

}
```

Figura 2.4: Fișier TypeScript dintr-o componentă Angular

Am ales să pun doar fișierul TypeScript și cel HTML, deoarece atunci când generez o componentă nouă, în fișierul CSS nu este scris nimic.

Până aici am văzut câte posibilități îmi oferă Angular și ce avantaje ar avea dacă ar folosi această tehnologie în dezvoltarea unei aplicații, însă ceea ce am menționat mai sus, legat de componente și comenzi, nu este tot ce poate oferi Angular, ci e abia începutul. Componentele și comenzile mă ajută mult să fac o aplicație web, dar Angular oferă ceva ce mă poate ajuta să fac o aplicație mult mai ușor, mai rapid și mai eficient, și anume Angular Material.

Angular Material este o bibliotecă în care se găsesc componente și stiluri predefinite special făcute pentru a fi integrate într-o aplicație Angular. Angular Material a fost dezvoltat de Google în 2014 și de atunci cei din echipa Angular aduc constant îmbunătățiri, actualizări și funcționalități noi tehnologiei, ceea ce o face să fie apreciată și aleasă de mulți programatori pentru a-și dezvolta aplicațiile.

Ca și Angular, Angular Material este actualizat foarte des, vorbind de câteva luni, deci trebuie să fii mereu la curent cu versiunile. Aplicațiile care folosesc Angular Material pot fi folosite pe diferite platforme: Android, iOS, Windows, MacOS, tehnologia fiind foarte versatilă.

Pentru a putea folosi Angular Material trebuie doar să rulez comanda de mai jos în terminalul proiectului în care vreau să folosesc biblioteca, și să aleg opțiunile care îmi vor fi oferite.

```
ng add @angular/material
```

Figura 2.5: Comandă adaugare Angular Material în proiect

2.1.2 ASP.NET Core Web API

ASP.NET Core Web API face parte din cadrul de lucru ASP.NET Core, și este folosit pentru a dezvolta servicii web API.

Înainte de a merge mai departe și să intru în detalii legate de ASP.NET Web API ar fi mai bine să explic ce este o aplicație de tip Web API. API este prescurtarea de la Application Programming Interface, care s-ar traduce Interfață de Programare a Aplicațiilor. O aplicație Web API este o aplicație care oferă servicii și funcționalități accesibile prin intermediul protocoalelor și standardelor web. Într-o aplicație de acest tip, funcționalitățile sunt expuse prin intermediul unor puncte finale de acces numite endpoint-uri care pot fi apelate de alte aplicații sau servicii. Aplicația primește cereri HTTP, cum ar fi GET, POST, PUT, DELETE, de la clienți și returnează răspunsuri HTTP care conțin datele solicitate.

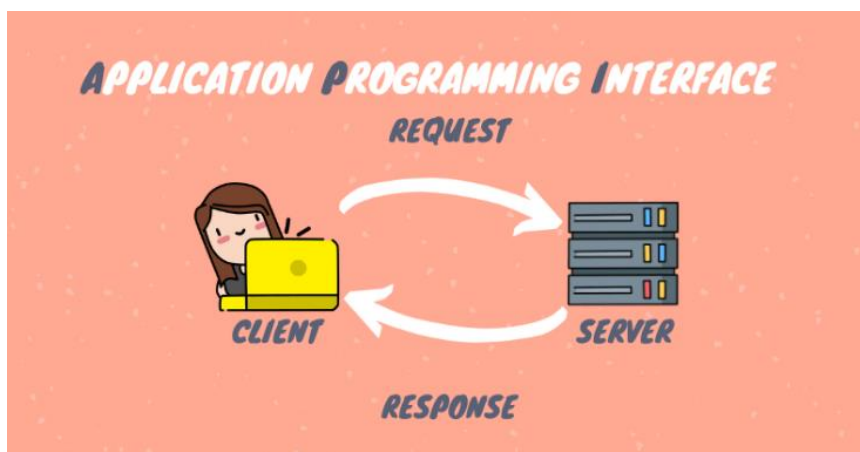


Figura 2.6: Intrefața de programare a aplicațiilorⁱⁱⁱ

ASP.NET Core Web API este un cadru de lucru puternic și flexibil dezvoltat de Microsoft, care permite crearea de servicii web RESTful² scalabile și interoperabile. Această tehnologie face parte din ecosistemul ASP.NET și oferă dezvoltatorilor un set robust de instrumente și biblioteci pentru construirea și implementarea aplicațiilor web bazate pe arhitectura API.

ASP.NET Core Web API se bazează pe fundamentul ASP.NET și permite dezvoltatorilor să creeze servicii web care pot fi consumate de diverse clienți, inclusiv:

- aplicații web,

² Representational state transfer

- aplicații mobile,
- aplicații de tip SPA (Single Page Application),
- dispozitive IoT (Internet of Things) și multe altele.

Unul dintre punctele forte ale acestui cadru de lucru este abilitatea sa de a returna date într-un format standardizat, cum ar fi JSON³ sau XML⁴, oferind astfel o interacțiune eficientă între server și client.

Caracteristica centrală a ASP.NET Core Web API este reprezentată de suportul său pentru arhitectura REST⁵. Aceasta implică utilizarea standardului HTTP⁶ și a metodelor sale (GET, POST, PUT, DELETE) pentru a crea, citi, actualiza și șterge resursele expuse prin intermediul API-ului. Prin adoptarea unei abordări bazate pe resurse și folosind conceptele de identificator al resursei (URI⁷) și de reprezentare a resursei, pot proiecta și implementa servicii web scalabile și ușor de utilizat.

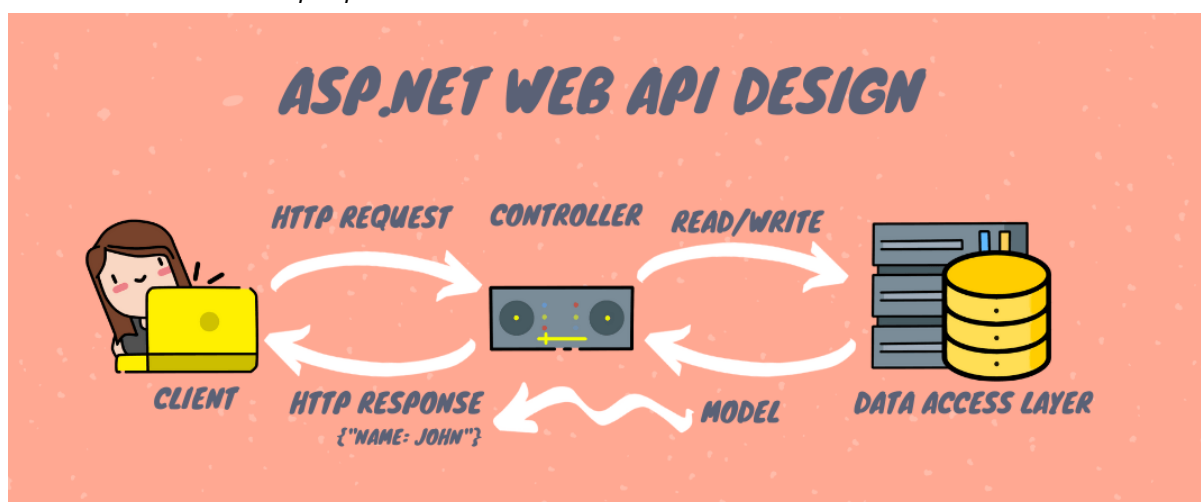


Figura 2.7: Modul de funcționare al ASP.NET Core Web API^{iv}

ASP.NET Core Web API facilitează, de asemenea, crearea de servicii web securizate, prin integrarea cu mecanisme precum autentificarea și autorizarea. Prin utilizarea sistemului de autentificare și autorizare bazat pe token-uri, pot proteja resursele expuse prin API și pot controla accesul utilizatorilor sau aplicațiilor la acestea.

³ JavaScript Object Notation

⁴ eXtensible Markup Language

⁵ Representational State Transfer

⁶ Hypertext Transfer Protocol

⁷ Uniform Resource Identifier

Un alt aspect important al ASP.NET Web API este flexibilitatea sa. Acesta poate fi utilizat împreună cu diverse tehnologii și biblioteci, cum ar fi Entity Framework pentru accesul la baze de date, biblioteca Newtonsoft.Json pentru serializarea și deserializarea datelor în format JSON, și multe altele. De asemenea, este compatibil cu diferite formate de date, cum ar fi XML și JSON, permițând astfel integrarea cu o varietate de clienți și servicii.

ASP.NET Core Web API pune la dispoziție, de asemenea, un set bogat de instrumente pentru testare și depanare. Prin intermediul unor instrumente precum Postman sau Swagger, pot testa și interoga serviciile API, pot crea documentație automată și pot monitoriza traficul și performanța acestora.

2.1.3 Microsoft SQL Server

Microsoft SQL Server este un sistem de gestionare a bazelor de date relaționale (RDBMS⁸) utilizat extensiv în industrie datorită capacităților sale versatile și documentației sale cuprinzătoare și detaliate. Acesta este considerat unul dintre cele mai apreciate sisteme de gestiune a bazelor de date disponibile.

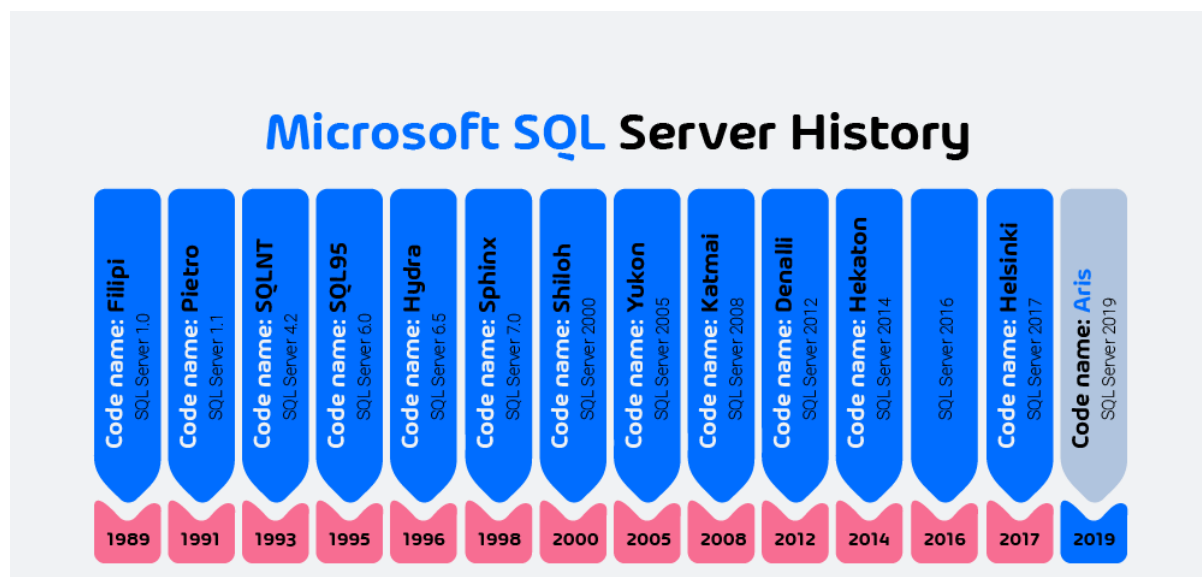


Figura 2.8: Istoria versiunilor Microsoft SQL Server^v

Inițial, SQL Server a fost dezvoltat în anii '80 de către compania Sybase Inc., iar în prezent este deținut și dezvoltat de Microsoft. De-a lungul anilor, SQL Server a evoluat și s-a îmbunătățit continuu, devenind un produs robust și fiabil pentru gestionarea datelor în medii de afaceri diverse.

⁸ Relational Database Management System

„Securitatea este esențială în orice sistem de stocare și procesare a datelor, iar SQL Server s-a mândrit că este cea mai securizată bază de date în ultimii opt ani, conform bazei de date a Institutului Național de Standarde și Tehnologie(NIST). SQL Server susține securitatea și conformitatea în mediul enterprise prin funcționalități de securitate precum Transparent Data Encryption, Auditing, Row-Level Security, Dynamic Data Masking și Always Encrypted.”[1]

Transparent Data Encryption⁹ este o caracteristică puternică a SQL Server care criptează automat datele în timpul stocării pe disc, asigurând protecția acestora chiar și în situația în care dispozitivul de stocare este compromis. Aceasta asigură confidențialitatea datelor și respectarea cerințelor de conformitate cu privire la securitatea informațiilor.

Auditing¹⁰ este o funcționalitate esențială pentru monitorizarea și urmărirea activităților desfășurate în baza de date. SQL Server permite configurarea auditării la nivel de bază de date și de tabele, oferind informații detaliate despre acțiunile efectuate asupra datelor.

Row-Level Security¹¹ permite definirea politicilor de securitate la nivel de rând, permițând accesul și vizualizarea datelor doar pentru utilizatorii autorizați. Aceasta oferă un control fin asupra accesului la informații și protejează datele sensibile împotriva accesului neautorizat.

Dynamic Data Masking¹² permite ascunderea sau mascarea datelor sensibile în timp real, în funcție de drepturile de acces ale utilizatorului. Aceasta previne expunerea accidentală a informațiilor confidențiale în aplicații sau rapoarte și asigură protecția datelor sensibile.

Always Encrypted¹³ este o funcționalitate avansată care permite criptarea datelor sensibile, inclusiv cheile de criptare, astfel încât acestea să fie păstrate în mod sigur chiar și atunci când sunt în tranzit sau stocate în medii nesigure. Aceasta oferă un nivel înalt de confidențialitate și protecție a datelor.

Prin intermediul acestor funcționalități de securitate, SQL Server asigură protecția datelor și respectarea standardelor și cerințelor de securitate în medii enterprise. Aceasta

⁹ Criptare Transparentă a Datelor

¹⁰ Auditarea

¹¹ Securitate la nivel de rând

¹² Mascarea dinamica al datelor

¹³ Întotdeauna criptate

face din SQL Server o alegere potrivită pentru aplicații și sisteme care necesită un nivel înalt de securitate și conformitate.

În lumea actuală, gestionarea eficientă a datelor este esențială pentru succesul organizațiilor de orice dimensiune. Un sistem de gestiune a bazelor de date puternic și bogat în funcționalități este cheia pentru a asigura stocarea, manipularea și administrarea corespunzătoare a acestor date. Unul dintre aceste sisteme este Microsoft SQL Server.

SQL Server, dezvoltat de Microsoft, oferă o platformă robustă pentru gestionarea datelor într-un mod eficient și sigur. Este recunoscut pentru caracteristicile sale avansate de securitate, opțiunile de scalabilitate și suportul pentru disponibilitatea înaltă și recuperarea în caz de dezastru. Aceste calități fac din SQL Server o alegere populară pentru organizații de diferite dimensiuni, de la întreprinderi mici până la corporații mari.

Cartea „Microsoft SQL Server 2019: A Beginner's Guide” de Dusan Petkovic, publicată în 2020, servește drept ghid pentru începători în lumea SQL Server. Această lucrare cuprinde conceptele fundamentale ale SQL Server, interogările SQL, proiectarea bazelor de date și sarcinile de administrare. Prin intermediul acestui ghid, pot învăța să utilizez SQL Server în mod eficient și să mă familiarizez cu principiile și practicile esențiale ale gestionării bazelor de date.

„SQL Server este un sistem puternic și bogat în funcționalități pentru gestionarea bazelor de date relaționale, dezvoltat de Microsoft. Oferă o platformă robustă pentru stocarea, gestionarea și manipularea datelor, fiind o opțiune populară pentru organizații de toate dimensiunile. Cu funcționalitățile sale avansate de securitate, opțiunile de scalabilitate și suportul pentru disponibilitate înaltă și recuperare în caz de dezastru, SQL Server oferă o soluție cuprinzătoare pentru gestionarea datelor. Această carte servește ca ghid pentru începători în utilizarea SQL Server, acoperind conceptele fundamentale, interogările SQL, proiectarea bazelor de date și sarcinile de administrare.”[2]

2.2 Limbaje de programare

2.2.1 TypeScript

TypeScript este un limbaj de programare open-source dezvoltat de Microsoft, bazat pe JavaScript. A fost creat pentru a extinde JavaScript prin adăugarea de tipuri statice și funcționalități specifice dezvoltării de aplicații de mari dimensiuni și complexitate.

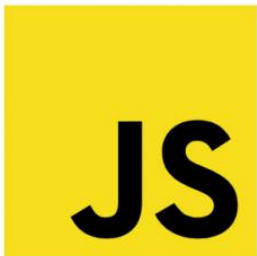

	
JAVASCRIPT	TYPESCRIPT
VS	
1 It does not support optional parameters.	1 It supports optional parameters.
2 It is interpreted language that is why it highlights the errors at runtime.	2 It compiles the code and highlights errors during the development time.
3 JavaScript does not support modules.	3 TypeScript gives support for modules.
4 In this, number, string are the objects.	4 In this, number, string are the interfaces.
5 JavaScript does not support generics.	5 TypeScript supports generics.

Figura 2.10: JavaScript vs TypeScript^{vi}

Unul dintre principalele avantaje ale TypeScript este posibilitatea de a adăuga tipuri de date la variabile, funcții și obiecte. Aceasta aduce beneficii semnificative în dezvoltarea software, deoarece oferă un sistem puternic de verificare a tipurilor în timpul compilării și elimină o serie de erori comune de tipuri care pot apărea în JavaScript.

TypeScript aduce o serie de funcționalități suplimentare față de JavaScript, cum ar fi tipurile de date avansate(inclusiv tipuri personalizate), interfețe, clase, moștenire, module și multe altele. Aceste caracteristici îmbunătățesc productivitatea dezvoltatorului, oferind suport pentru dezvoltarea de aplicații scalabile și ușor de întreținut.

Un alt aspect important al TypeScript este că acesta este transpilat în JavaScript standard, ceea ce înseamnă că codul TypeScript poate fi rulat pe orice browser sau platformă care suportă JavaScript.

TypeScript a devenit tot mai popular în comunitatea dezvoltatorilor datorită beneficiilor pe care le aduce în dezvoltarea aplicațiilor web și a aplicațiilor pe partea de server. Este folosit în special în cadrul cadrelor de lucru și bibliotecilor populare, precum Angular, React și Vue.js.

Cu ajutorul TypeScript, pot scrie cod mai sigur, mai ușor de întreținut și mai scalabil, beneficiind totodată de avantajele ecosistemului JavaScript.

2.2.2 C#

C# este un limbaj de programare modern, orientat pe obiecte, dezvoltat de Microsoft în anul 2000. A fost conceput pentru a fi un limbaj puternic, versatil și ușor de înțeles, adaptat dezvoltării aplicațiilor de tip enterprise, web și mobile.

C# face parte din familia de limbaje de programare C și este strâns legat de platforma .NET. Acesta beneficiază de un sistem de tipuri puternic, gestionarea automată a memoriei și o sintaxă clară și expresivă, ceea ce îl face alegerea preferată pentru dezvoltarea de aplicații robuste și scalabile.

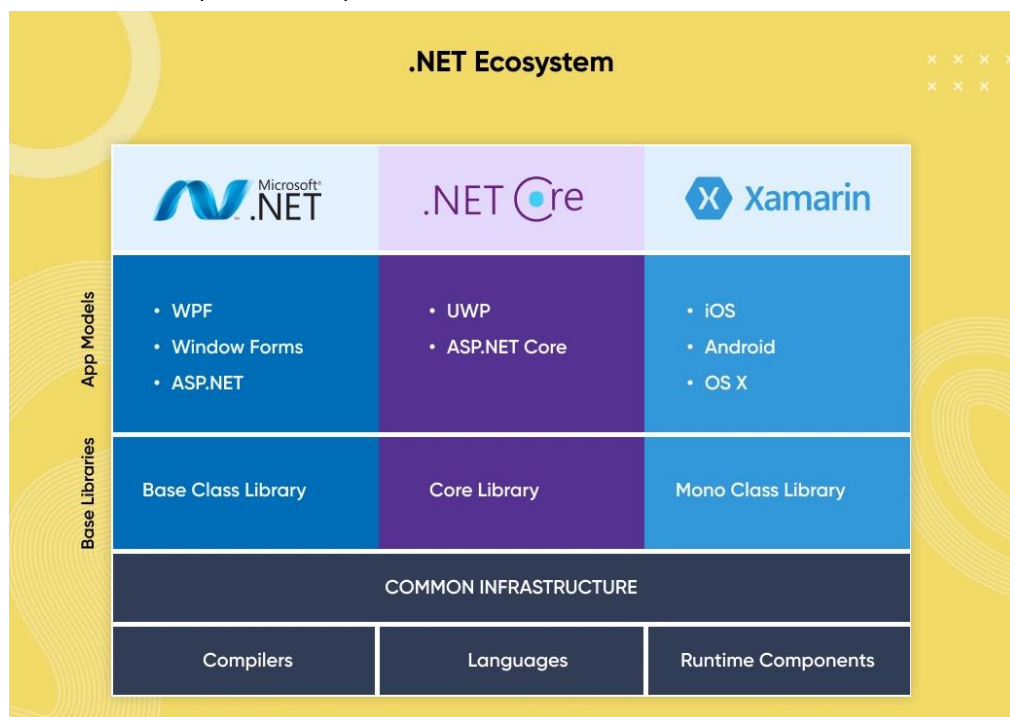


Figura 2.11: Ecosistemul .NET^{vii}

Una dintre caracteristicile-cheie ale limbajului C# este orientarea sa pe obiecte. Acesta imi permite să modelez structuri de date și să creez obiecte care să interacționeze între ele prin intermediul conceptelor de clasă, moștenire, încapsulare și polimorfism. Aceasta facilitează dezvoltarea de cod modular și extensibil, care poate fi mai ușor de întreținut pe parcursul timpului.

Un alt aspect important al limbajului C# este tipizarea sa statică. Acest lucru înseamnă că toate variabilele și expresiile trebuie să fie declarate cu un tip de date specific înainte de a fi utilizate și că tipurile nu pot fi schimbate ulterior. Tipizarea statică aduce beneficii precum detectarea mai ușoară a erorilor de tip în timpul compilării și oferă o mai mare siguranță și claritate în cod.

C# oferă, de asemenea, suport puternic pentru programarea asincronă. Acest lucru este deosebit de util în aplicațiile care trebuie să răspundă la evenimente și să desfășoare operații de lungă durată fără a bloca firul principal de execuție. Prin utilizarea cuvântului cheie "async" și a bibliotecii de tipuri "Task", pot crea operații asincrone și pot beneficia de o performanță sporită în aplicațiile lor.

C# este, de asemenea, strâns integrat cu platforma .NET, ceea ce înseamnă că pot avea acces la o gamă largă de biblioteci și cadre de lucru pentru dezvoltarea aplicațiilor. Aceasta include .NET Framework, care este utilizat pentru dezvoltarea de aplicații Windows, precum și .NET Core, care este utilizat pentru dezvoltarea de aplicații cross-platform. Există și cadre de lucru specifice domeniului, cum ar fi ASP.NET pentru dezvoltarea aplicațiilor web și Xamarin pentru dezvoltarea aplicațiilor mobile.

C# este utilizat într-o gamă largă de domenii, inclusiv dezvoltarea de aplicații desktop, aplicații web, aplicații mobile, jocuri, sisteme înglobate și multe altele. Comunitatea dezvoltatorilor C# este activă și oferă suport și resurse pentru îmbunătățirea cunoștințelor și abilităților în acest limbaj.

Cu toate acestea, C# este un limbaj în continuă evoluție, iar Microsoft lansează în mod regulat noi versiuni și actualizări pentru a adăuga funcționalități noi și pentru a îmbunătăți performanța și productivitatea dezvoltatorilor. Prin urmare, rămâne un limbaj viu și actual în industria software, oferind posibilități nelimitate pentru crearea de aplicații moderne și inovatoare.

2.3 Medii de programare

2.3.1 Visual Studio Code

Visual Studio Code (VS Code) este un editor de cod sursă dezvoltat de Microsoft, care s-a bucurat de o popularitate rapidă în rândul comunității dezvoltatorilor. Am ales să utilizez Visual Studio Code pentru dezvoltarea aplicației noastre datorită numeroaselor beneficii pe care le oferă.

Unul dintre avantajele majore ale utilizării Visual Studio Code este experiența de lucru eficientă și rapidă. Cu ajutorul completării automate a codului și a evidențierii sintaxei, am putut scrie codul mai repede și mai precis. Aceasta mi-a permis să creez aplicația într-un timp mai scurt, fără a face compromisuri în ceea ce privește calitatea.

Un alt instrument care mi-a fost de mare ajutor în dezvoltarea aplicației este GitHub Copilot. Acesta este un sistem de inteligență artificială dezvoltat de GitHub și OpenAI, care oferă sugestii și completări de cod bazate pe contextul și nevoile mele. Cu ajutorul GitHub Copilot, am putut scrie cod mai rapid și mai eficient, beneficiind de recomandări inteligente și adaptate specificului proiectului meu.

2.3.2 Visual Studio 2022

Visual Studio 2022 este cea mai nouă versiune a mediului de dezvoltare integrat (IDE) oferit de Microsoft, având un set bogat de funcționalități și îmbunătățiri care îi susțin utilizatorii în dezvoltarea software. Această versiune aduce o serie de avantaje și facilități pentru programatori, în special pentru proiectele bazate pe platforma ASP.NET Core Web API.

Unul dintre aspectele cheie ale Visual Studio 2022 este suportul avansat pentru cadrul de lucru Entity Framework (EF). EF este o tehnologie dezvoltată de Microsoft care facilitează lucrul cu bazele de date relaționale într-un mod simplu și eficient. Prin intermediul Visual Studio 2022, dezvoltatorii pot utiliza instrumentele EF pentru a defini modelele de date, a efectua operații de interogare și manipulare a datelor și a realiza migrări ale schemelor bazei de date. Aceasta aduce un nivel ridicat de productivitate și ușurință în dezvoltarea și gestionarea bazei de date.

2.4 Tactici și tehnici de dezvoltare

2.4.1 Single page application

Single Page Application (SPA) este o abordare modernă în dezvoltarea aplicațiilor web, care se diferențiază de modelul tradițional de navigare între pagini multiple. Într-o SPA, întreaga aplicație este încărcată o singură dată în browser, iar toate interacțiunile ulterioare cu serverul sunt gestionate prin intermediul comunicării asincrone cu servicii web.

	SPAs	MPAs
PERFORMANCE	Faster loading time	Slower loading time
DEBUGGING	More difficult	Well supported by debugging tools
DEVELOPMENT	Fast	Slower, more complex
MAINTENANCE	Fast & easy	Slower
SECURITY	simplified	More challenging
SEO	Limited	Easier & more effective
COST	More expensive	Less expensive
SCALABILITY	Not scalable	Scalable

Figura 2.12: Single Page Application vs Multiple Page Application^{viii}

Principala caracteristică a unei SPA este că toate schimbările de conținut și de stare sunt realizate dinamic, fără a se reîncărca întreaga pagină. În schimb, se utilizează tehnici de manipulare a DOM¹⁴ pentru a actualiza și reafirma secțiunile specifice ale aplicației, oferind astfel o experiență fluidă și interactivă utilizatorului.

¹⁴ Document Object Model

Una dintre principalele avantaje ale unei SPA este viteza și reactivitatea, deoarece paginile nu trebuie reîncărcate în mod constant. Acest lucru conduce la o interacțiune mai fluidă și la o utilizare îmbunătățită. De asemenea, o SPA permite încărcarea diferită a resurselor, ceea ce duce la o utilizare mai eficientă a bândei de internet.

O altă caracteristică importantă a SPA este reutilizarea codului și separarea clară între frontend și backend. SPA-urile utilizează API-uri pentru a comunica cu serviciile web și pentru a obține sau a trimite date. Aceasta permite dezvoltatorilor să creeze și să mențină logică de afișare și de interacțiune în frontend, în timp ce serverul se ocupă de prelucrarea datelor și de logica de afaceri în backend.

Pentru a realiza o SPA, se utilizează de obicei un cadru de lucru de dezvoltare a frontend-ului, cum ar fi Angular, React sau Vue.js. Aceste cadre oferă un set de instrumente și biblioteci care facilitează gestionarea stării aplicației, rutarea, manipularea DOM-ului și comunicarea cu serviciile web.

Cu toate acestea, există și câteva aspecte de luat în considerare atunci când se dezvoltă o SPA. Deoarece întreaga aplicație este încărcată o singură dată, inițializarea inițială a aplicației poate necesita mai mult timp, iar gestionarea memoriei și a resurselor trebuie abordată cu atenție pentru a evita problemele de performanță.

2.4.2 Comunicarea HTTP

Comunicarea HTTP (Hypertext Transfer Protocol) reprezintă un protocol standard utilizat pentru transmiterea datelor între client și server pe World Wide Web. În contextul arhitecturii REST (Representational State Transfer), comunicarea HTTP joacă un rol central în interacțiunea între client și serviciul web.

Principala caracteristică a comunicării HTTP în cadrul unei arhitecturi RESTful este reprezentată de utilizarea metodelor HTTP pentru a accesa și manipula resursele. Cele mai comune metode utilizate în acest context sunt:

- GET: Această metodă este utilizată pentru a obține o resursă specifică de la server. De exemplu, un client poate utiliza o cerere GET pentru a obține informații despre un produs specific de pe un site de comerț electronic.
- POST: Metoda POST este folosită pentru a trimite date către server pentru a crea o nouă resursă. De exemplu, atunci când un utilizator completează un formular de înregistrare pe un site web, datele vor fi trimise prin intermediul unei cereri POST pentru a crea un nou cont.

- PUT: Metoda PUT este utilizată pentru a actualiza o resursă existentă sau pentru a crea o resursă nouă într-un anumit loc. De exemplu, un client poate utiliza o cerere PUT pentru a actualiza detaliile unui produs dintr-un magazin online.
- DELETE: Metoda DELETE este utilizată pentru a șterge o resursă specifică de pe server. De exemplu, un client poate utiliza o cerere DELETE pentru a șterge un articol dintr-un blog.

Pe lângă metodele menționate mai sus, comunicarea HTTP în cadrul arhitecturii RESTful se bazează pe alte elemente importante:

- URI (Uniform Resource Identifier): Reprezintă un identificator unic care identifică resursa la care se face referire în cerere. De exemplu, într-o cerere GET, URI-ul poate indica adresa URL a unei pagini web sau a unui serviciu web.
- Header-uri HTTP: Acestea conțin informații suplimentare despre cerere sau răspuns, cum ar fi tipul de conținut acceptat sau trimis, informații de autentificare, cache-control și multe altele.
- Body: Acesta reprezintă conținutul datelor trimise sau primite în cadrul unei cereri sau a unui răspuns HTTP. De exemplu, într-o cerere POST, corpul poate conține datele formularului trimise de utilizator.

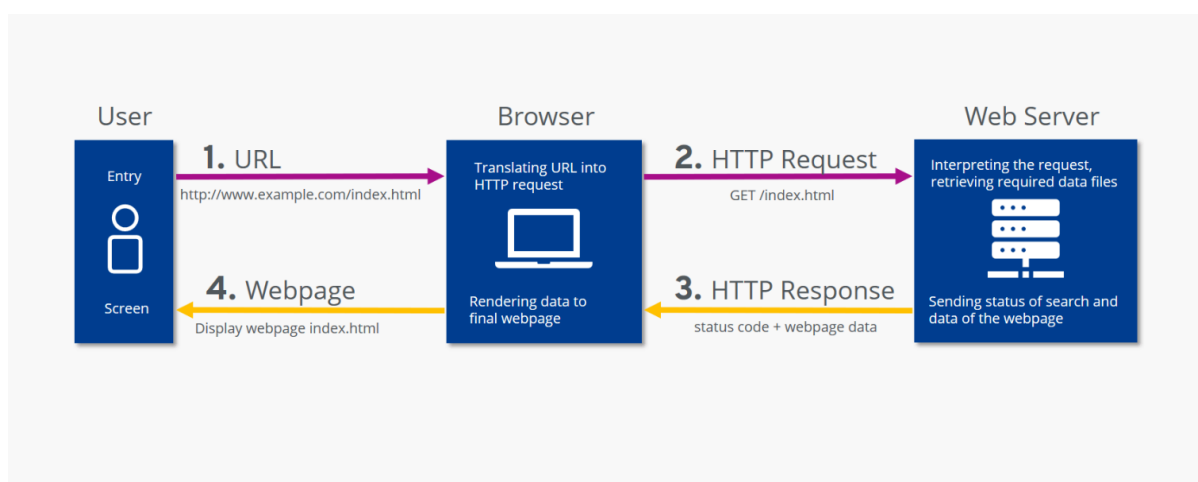


Figura 2.13: Comunicarea HTTP^{ix}

Comunicarea HTTP în arhitectura RESTful facilitează interoperabilitatea și integrarea între sisteme diferite. Folosind metodele HTTP standard și respectând principiile REST, pot crea servicii web scalabile, ușor de utilizat și independente de platformă.

De asemenea, comunicarea HTTP oferă suport pentru diferite formate de date, cum ar fi JSON și XML, permițând astfel transmiterea și recepționarea datelor într-un mod flexibil și eficient.

2.4.3 JSON Web Token

JSON Web Token (JWT) este o tehnologie de securitate care permite transmiterea sigură a datelor între diferite entități într-un format compact și autenticat. Este utilizat în mod obișnuit în aplicațiile web și mobile pentru autentificare și autorizare.

JWT este compus din trei părți principale: header, payload și semnătură. Header-ul conține informații despre tipul de token și algoritmul de criptare utilizat. Payload-ul conține datele pe care doresc să le transmit, cum ar fi informații despre utilizator sau detalii de autorizare. Semnătura este rezultatul aplicării unui algoritm de criptare asupra header-ului, payload-ului și unei chei secrete, asigurând astfel autenticitatea și integritatea tokenului.

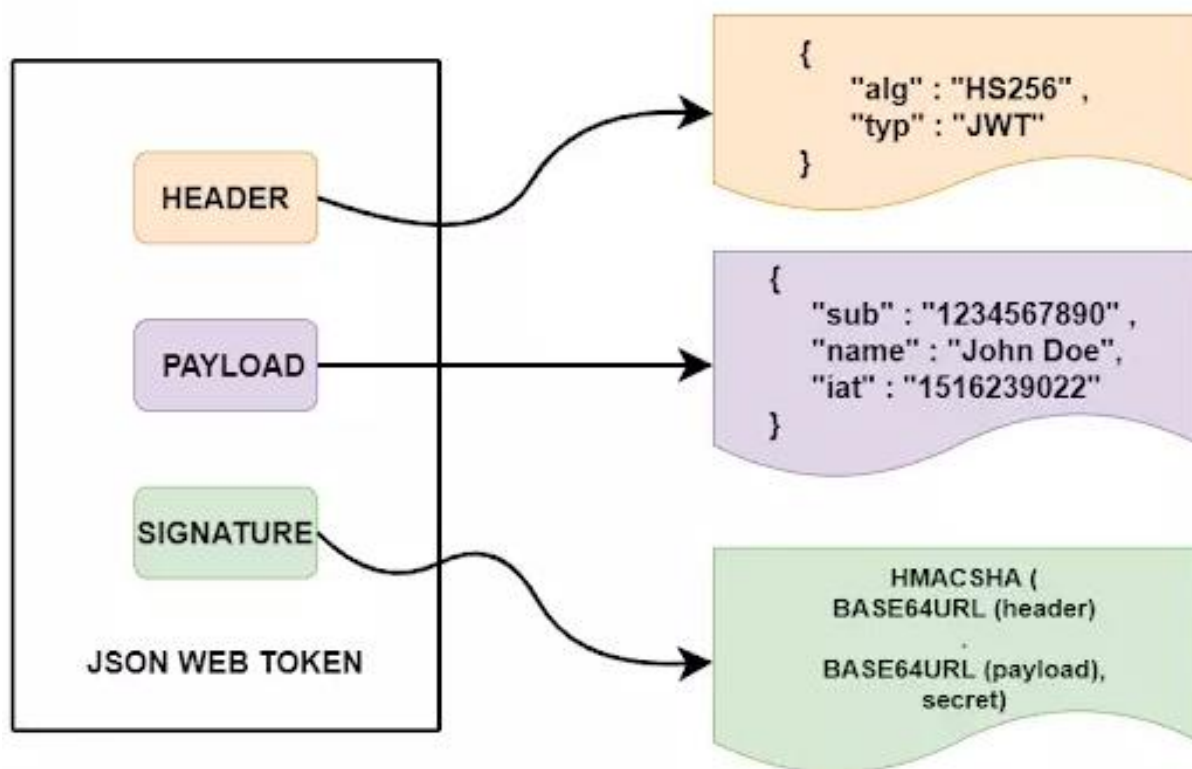


Figura 2.14: Structura JWT^x

Unul dintre principalele avantaje ale JWT este faptul că poate fi verificat rapid și eficient de către părțile implicate, fără a necesita interogarea unei baze de date sau un

server de autentificare. Acest aspect face ca JWT să fie o soluție scalabilă și eficientă în aplicațiile distribuite.

JWT este folosit în principal pentru autentificare și autorizare. După ce un utilizator se autentifică cu succes, un token JWT este generat și trimis către client. Acesta poate fi inclus în cererile ulterioare către server pentru a confirma identitatea utilizatorului. Serverul poate verifica și valida token-ul primit pentru a autoriza accesul la resursele protejate.

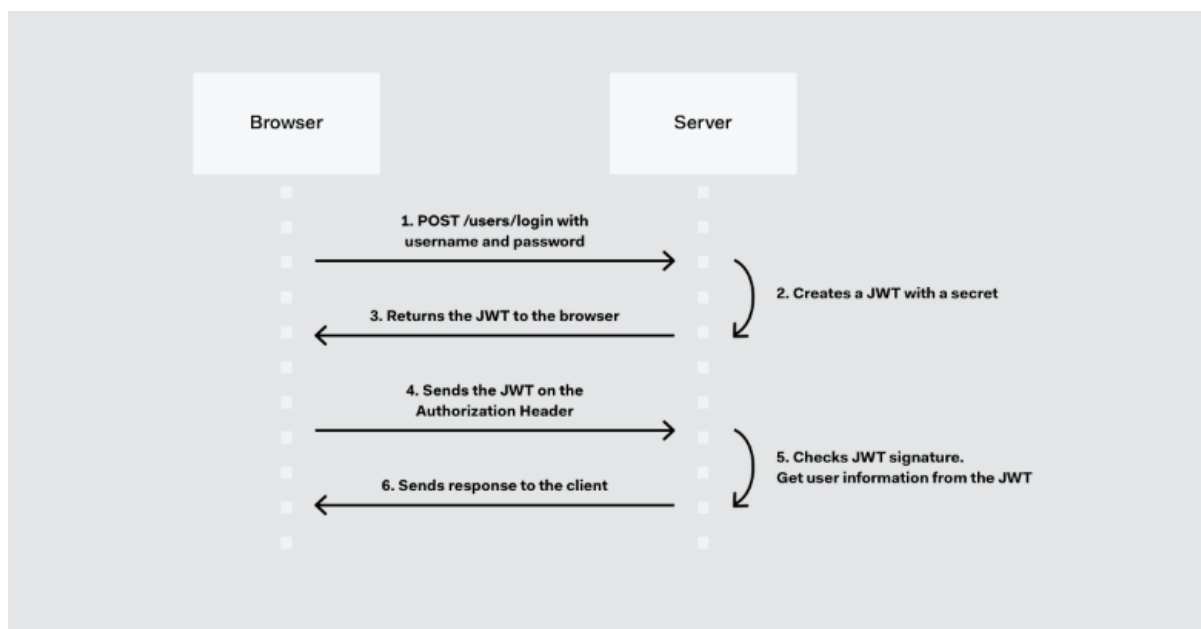


Figura 2.15: Modul de funcționare al JWT^{xi}

Un alt aspect interesant al JWT este că poate transporta informații suplimentare despre utilizator, numite și claim-uri. Aceste claim-uri pot conține privilegii speciale, roluri, date de profil sau alte informații relevante pentru aplicație.

Este important de menționat că securitatea JWT depinde de păstrarea în siguranță a cheii secrete utilizate pentru semnătură. Aceasta trebuie protejată și accesibilă doar entităților autorizate. De asemenea, dimensiunea și complexitatea token-ului trebuie luate în considerare, deoarece acesta este trimis cu fiecare cerere, ceea ce poate afecta performanța aplicației.

2.4.4 Metoda Code First

Metoda Code First este o abordare de dezvoltare utilizată în cadrul de lucru Entity Framework (EF) pentru gestionarea bazei de date în aplicații .NET. Ea pune accent pe definirea și configurarea modelelor de date în codul sursă al aplicației, urmând ca baza de date să fie generată automat pe baza acestor modele.

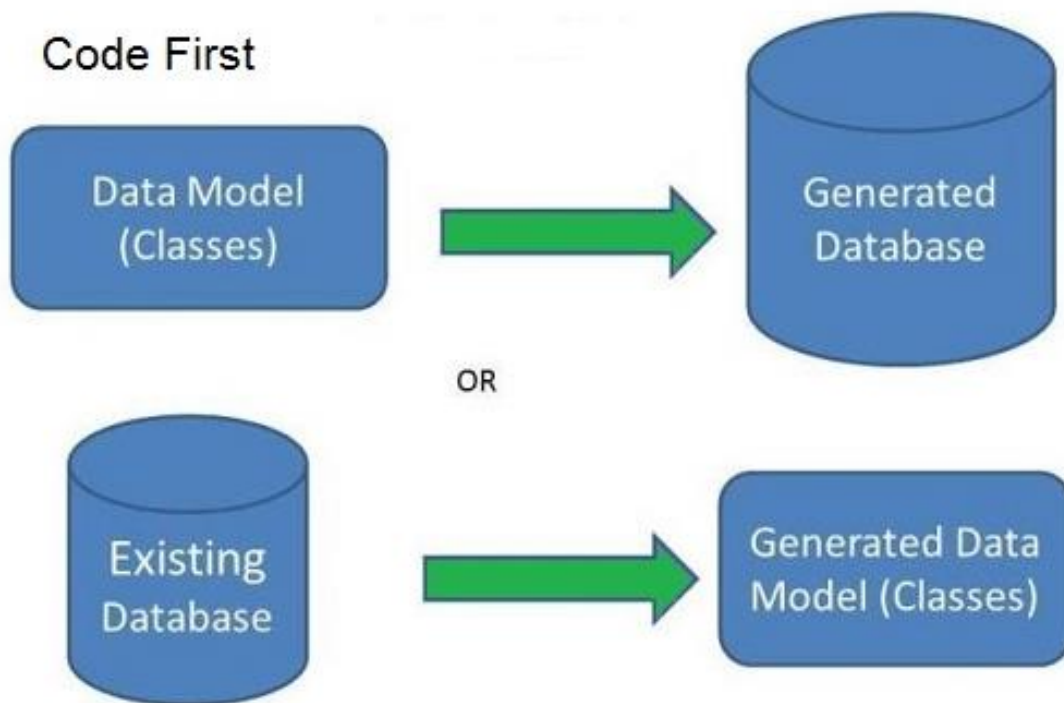


Figura 2.16: Code First vs Database First^{xii}

Procesul de utilizare a metodei Code First implică următorii pași:

- 1) Definirea modelelor de date: definesc clase în codul sursă al aplicației pentru a reprezenta entitățile și relațiile din domeniul de lucru. Aceste clase pot fi decorate cu attribute specifice pentru a configura aspecte precum cheile primare, relațiile între entități și restricțiile.
- 2) Crearea contextului de bază de date: creez o clasă care extinde clasa DbContext din Entity Framework. Această clasă reprezintă contextul de bază de date și conține seturile de date (DbSet) pentru fiecare entitate. De asemenea, se poate configura conexiunea la baza de date și alte opțiuni specifice.

- 3) Generarea și aplicarea migrărilor: După ce modelele de date au fost definite, Entity Framework permite generarea automată a migrărilor. Acestea reprezintă schimbările necesare în schema bazei de date pentru a reflecta modelele de date. Migrările pot fi apoi aplicate pentru a crea sau actualiza baza de date.
- 4) Utilizarea bazei de date: După generarea bazei de date, pot utiliza contextul de bază de date pentru a interacționa cu datele. Acest lucru include operații precum adăugare, actualizare, ștergere și interogare a entităților.

Metoda "Code First" oferă un nivel ridicat de control și flexibilitate în gestionarea bazei de date în aplicații .NET. Dezvoltatorii pot defini modelele de date în mod explicit și personalizat, iar Entity Framework se ocupă de generarea schemelor și migrărilor necesare. Acest lucru facilitează dezvoltarea rapidă și iterativă, oferind totodată posibilitatea de a adăuga modificări și funcționalități suplimentare pe măsură ce aplicația evoluează.

2.4.5 Repository Pattern

Repository Pattern este o strategie de proiectare utilizată în dezvoltarea software pentru a organiza și gestiona interacțiunea cu baza de date într-o aplicație. În loc să scriu cod specific pentru fiecare operațiune de acces la date, acest șablon imi permite să definesc o interfață comună pentru accesul la date și să implementez această interfață în clase separate numite repository-uri.

Interfața repository definește metodele generice pentru a crea, citi, actualiza și șterge datele din baza de date. Fiecare repository concret implementează aceste metode în funcție de logica specifică a entităților și a mecanismului de stocare. Astfel, repository pattern imi oferă un nivel de abstractizare care separă logica de afaceri de detalii specifice ale bazei de date.

Am ales să utilizez Repository Pattern în cadrul aplicației mele pentru a obține o structură eficientă și modulară a accesului la date.

Unul dintre principalele motive pentru care am optat pentru Repository Pattern este reutilizabilitatea și flexibilitatea pe care o oferă. Prin intermediul acestui pattern, pot defini interfețe clare și abstracte pentru operațiile de acces la date, cum ar fi crearea, citirea, actualizarea și ștergerea (CRUD). Astfel, pot implementa diferite repository-uri care să utilizeze diferite tehnologii de stocare a datelor, precum baze de date relaționale sau non-relaționale, fără a afecta logica de afaceri a aplicației.

3.Structura și dezvoltarea aplicației

În acest capitol voi examina arhitectura generală a aplicației, evidențiind componentele și fluxul de date între ele, mă voi concentra pe detaliile de implementare ale aplicației și voi explora diferite aspecte ale funcționalităților principale.

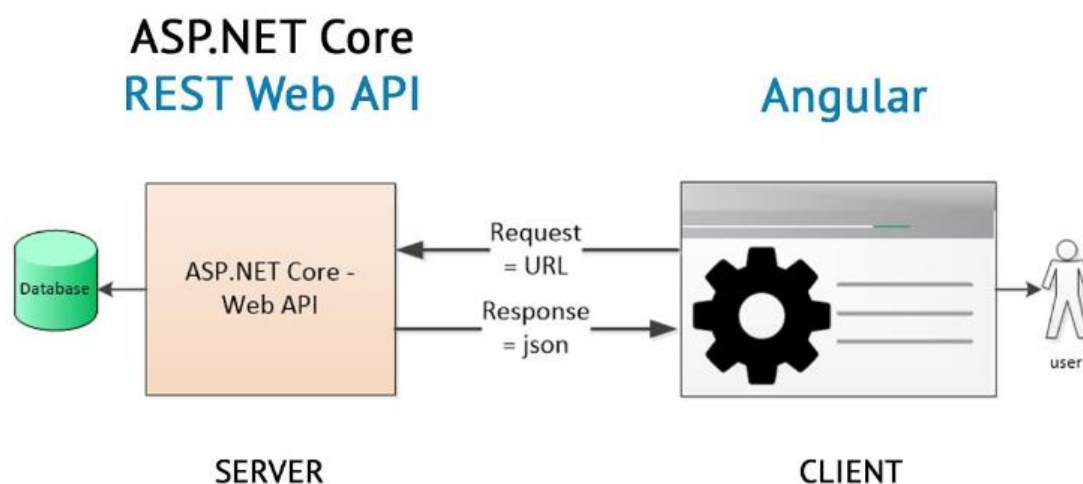


Figura 3.0: Arhitectura aplicației^{xiii}

Arhitectura aplicației implică două componente principale: clientul (Angular) și serverul (ASP.NET Core Web API). Aceste componente lucrează împreună pentru a asigura funcționalitatea aplicației și pentru a permite interacțiunea cu utilizatorii.

Comunicația între client și server se realizează prin intermediul cererilor HTTP. Clientul Angular trimite cereri HTTP către server pentru a solicita date sau pentru a trimite informații. Serverul primește aceste cereri, le procesează și returnează răspunsurile corespunzătoare. De obicei, aceste răspunsuri sunt în format JSON și conțin datele solicitate sau rezultatele operațiilor efectuate.

Angular se concentrează pe partea de prezentare și interacțiune cu utilizatorul, în timp ce ASP.NET Core Web API se ocupă de gestionarea logicii de afaceri și interacțiunea cu baza de date. Această separare facilitează dezvoltarea, testarea și mentenanța aplicației, și permite scalabilitatea și extensibilitatea ulterioară.

3.1 Arhitectura aplicației – diagrame UML

Diagramele UML reprezintă o modalitate eficientă de a vizualiza și de a comunica aspecte cheie ale proiectului, precum componentele, relațiile și interacțiunile dintre acestea.

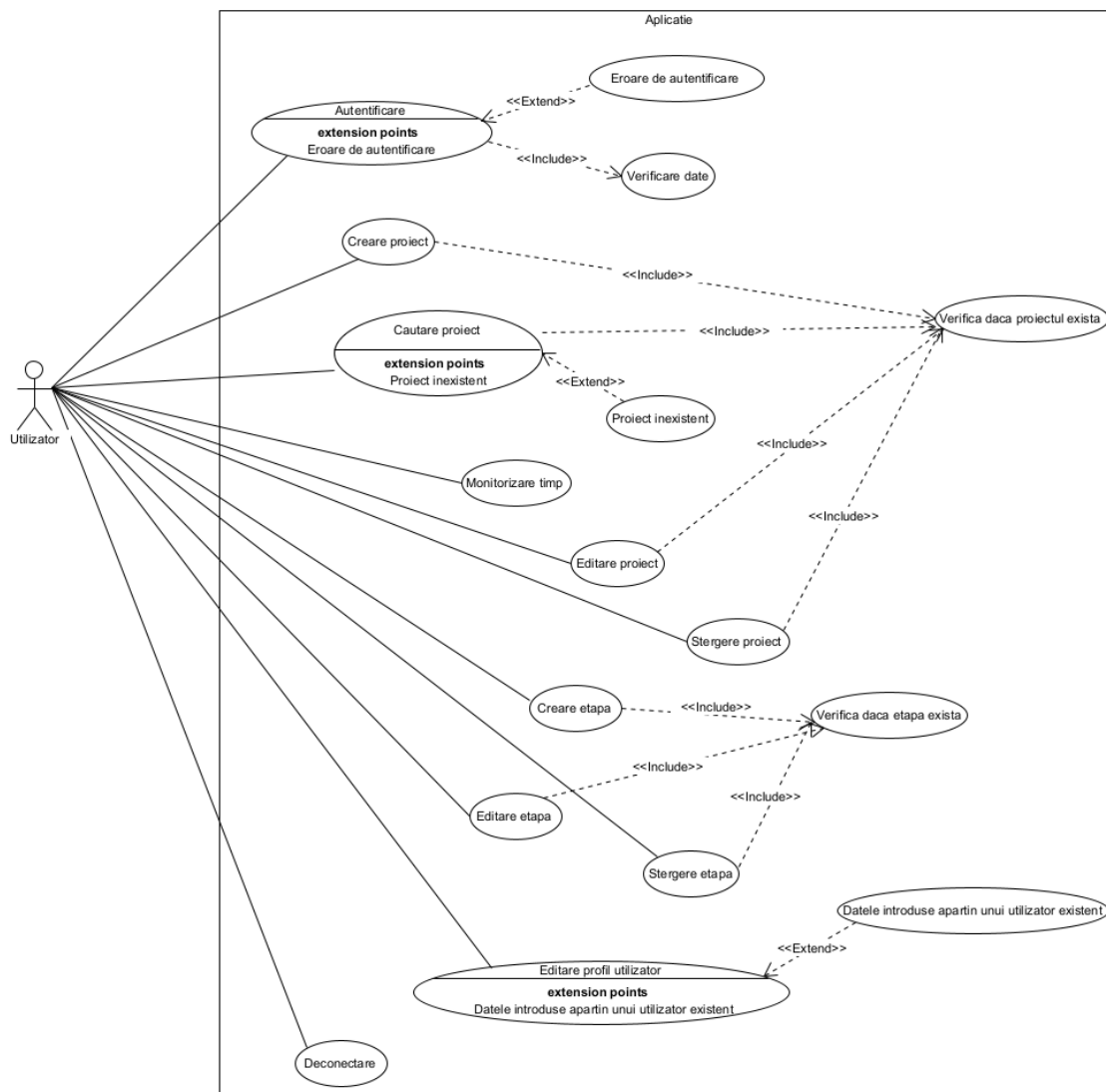


Figura 3.1: Diagrama de cazuri

Așa cum este ilustrat în diagrama de cazuri de mai sus, există un singur actor principal, utilizatorul, care poate face următoarele operațiuni:

- Autentificare/Deconectare/Editare profil
- Operații pe proiecte: creare, căutare, editare, ștergere
- Operații pe etape: creare, căutare, editare, ștergere
- Monitorizare timp

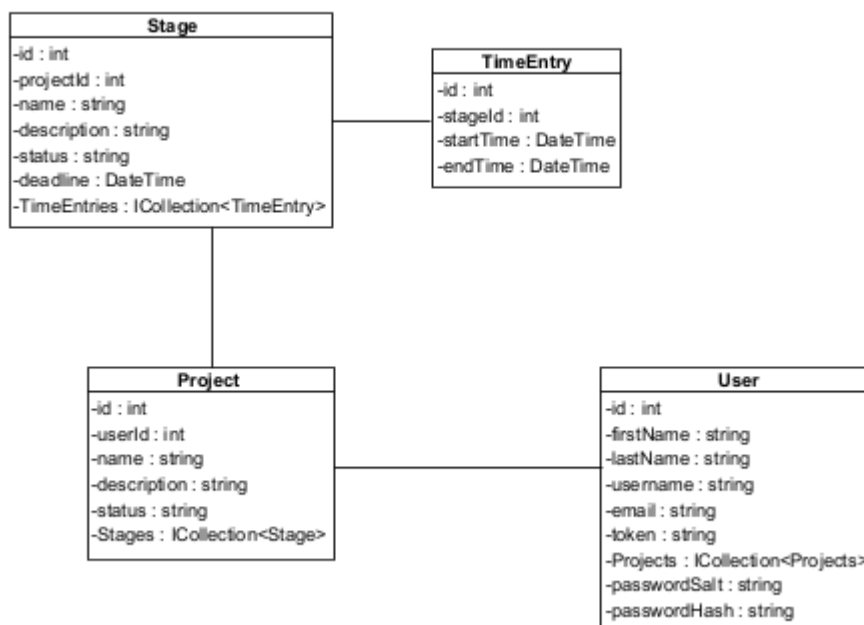


Figura 3.2: Diagrama de clase

Diagrama de clase ilustrată prezintă structura și relațiile dintre clasele cheie ale aplicației. În diagrama dată, există patru clase principale: User, Project, Stage și TimeEntry. Fiecare dintre aceste clase este asociată cu clasele DTO și Repository corespunzătoare.

Clasa User reprezintă un utilizator al aplicației și conține informații precum numele, adresa de email și detalii legate de autentificare. Clasa Project reprezintă un proiect și conține detalii despre nume, descriere și stadiul proiectului. Clasa Stage reprezintă o etapă în cadrul unui proiect și include informații despre nume, descriere, stadiu și termen limită. Clasa TimeEntry reprezintă o înregistrare a timpului dedicat pentru o anumită etapă și conține informații despre începutul și sfârșitul activității.

Fiecare dintre aceste clase este asociată cu o clasă DTO, și anume UserDTO, ProjectDTO, StageDTO și TimeEntryDTO. Aceste clase DTO sunt utilizate pentru transferul de date între diferitele straturi ale aplicației sau între aplicație și client.

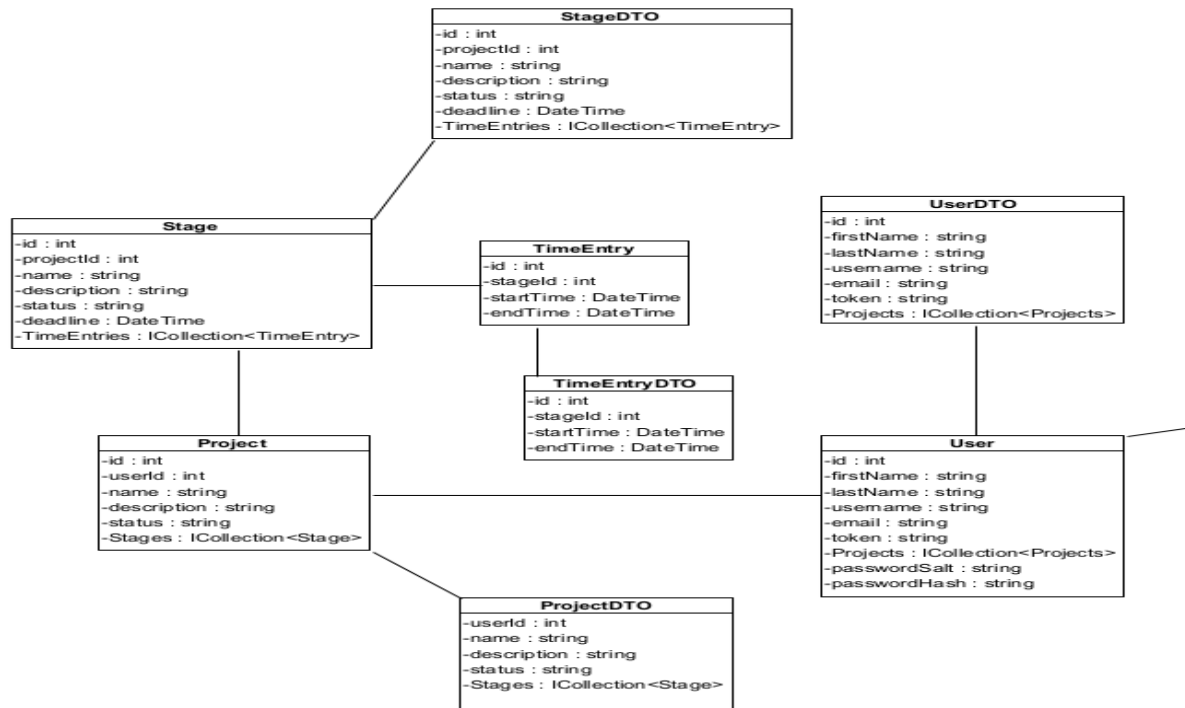


Figura 3.3: Diagrama de clase cu DTO

De asemenea, fiecare clasă principală are asociată o clasă Repository corespunzătoare, și anume UserRepository, ProjectRepository, StageRepository și TimeEntryRepository. Aceste clase Repository oferă metode pentru accesul și manipularea datelor asociate fiecărei clase, precum crearea, citirea, actualizarea și ștergerea înregistrărilor.

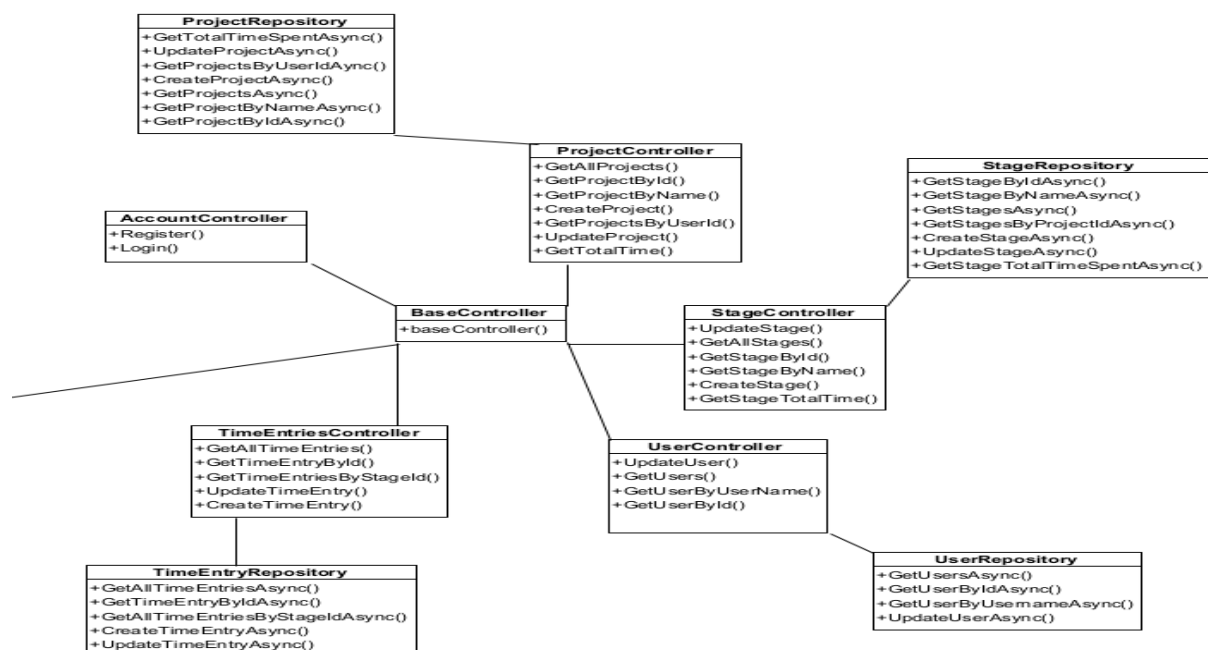


Figura 3.4: Diagrama de clase cu repository și controlere

Diagrama de clase evidențiază relațiile dintre aceste clase, precum asocierea între User și Project pentru a indica faptul că un utilizator poate avea mai multe proiecte sau relația de agregare între Project și Stage pentru a indica faptul că un proiect poate conține mai multe stadii. De asemenea, se poate observa relația de asociere între Stage și TimeEntry pentru a indica că un stadiu poate avea mai multe înregistrări de timp.

3.2 Arhitectura interfeței

Interfața utilizator reprezintă componenta principală a aplicației prezentate, și este responsabilă de interacțiunea cu utilizatorul. Pentru implementarea interfeței am ales Angular, cadru de lucru pe care l-am descris mai în detaliu și cu mai multe amănunte în subcapitolul Cadre de lucru. Am ales Angular deoarece oferă un set bogat de funcționalități și abstracții care simplifică dezvoltarea unei aplicații web complexe, precum și o structură modulară.

3.2.1 Injectarea dependențelor

Un aspect arhitectural important al interfeței în Angular este injectarea dependențelor(DI)¹⁵. Acesta este un mecanism prin care furnizez o instanță nouă a unei clase cu dependențele sale complete. În Angular, DI este gestionat de Injector, un serviciu de gestionare a dependențelor care se ocupă de înregistrarea dependențelor și de crearea lor.

```
@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrls: ['./dashboard.component.css']
})

export class DashboardComponent {
  username?: string;
  newProjectName: string = "";
  showAddProjectForm: boolean = false;

  constructor(private userService: UserService, private dialog: MatDialog) {
    this.username = this.userService.user.username;
  }
}
```

Figura 3.5: Injectarea unui serviciu într-o componentă

¹⁵ Dependency Injection(DI)

În Angular, DI se bazează pe constructorii claselor. Atunci când Angular creează un component, acesta solicită injectorului serviciile de care componenta are nevoie. Injectorul menține o colecție de instanțe de servicii create anterior. Atunci când o instanță de serviciu este solicitată, injectorul verifică dacă aceasta există deja în colecție. Dacă serviciul nu se află în colecție, injectorul creează o nouă instanță și o adaugă în colecție înainte de a o returna către Angular. Odată ce toate serviciile solicitate au fost rezolvate și returnate, Angular poate apela constructorul componentei și poate furniza acele servicii ca argumente.

3.2.2 Module

În Angular, aplicațiile sunt modulare și utilizează conceptul de NgModule. Fiecare aplicație are cel puțin un NgModule, numit AppModule, care este modulul rădăcină și este utilizat pentru pornirea aplicației. NgModule descrie modul în care părțile aplicației se încadrează împreună și are câteva proprietăți cheie în configurarea sa.

```
@NgModule({
  declarations: [
    AppComponent,
    AuthComponent,
    SidebarComponent,
    DashboardComponent,
    ProjectsComponent,
    ReportsComponent,
    AddProjectComponent,
    StopwatchComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserModuleAnimationsModule,
    FormsModule,
    MaterialModule,
    CommonModule,
    HttpClientModule,
    NgxMatTimepickerModule,
    NgxMatDatettimePickerModule,
    NgxMatMomentModule,
    ReactiveFormsModule
  ],
  providers: [{ provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }],
  bootstrap: [AppComponent],
  entryComponents: [AddProjectComponent]
})
export class AppModule { }
```

Figura 3.6: Captură de ecran cu modulul rădăcină AppModule

Un modul Angular este o colecție logică de componente, directive, servicii și alte resurse legate, care sunt grupate împreună pentru a oferi o funcționalitate specifică. Modulul reprezintă o unitate modulară independentă, care poate fi încărcată și utilizată în cadrul aplicației.

3.2.3 Directive și componente

Directivele sunt clase decorate cu decoratorul @Directive și pot fi aplicate ca attribute pe elemente HTML existente sau pot crea elemente HTML personalizate. Există două tipuri principale de directive: directivele structurale și directivele de atribut.

```
<div class="auth-wrapper">
  <form #authForm="ngForm" (ngSubmit)="onSubmit(authForm)" class="form-wrapper">
```

Figura 3.7: Formular de autentificare și înregistrare în HTML

Această secțiune de cod reprezintă formularul de autentificare și înregistrare. Utilizând directiva ngForm, formularul este legat de componenta corespunzătoare și este gestionat prin intermediul metodei onSubmit().

```
<mat-form-field appearance="fill" class="example-full-width" *ngIf="!isLoginMode">
  <mat-label>First Name</mat-label>
  <input matInput placeholder="Enter your firstname" type="name" name="firstname" ngModel [required]="true">
</mat-form-field>
<mat-form-field appearance="fill" class="example-full-width" *ngIf="!isLoginMode">
  <mat-label>Last Name</mat-label>
  <input matInput placeholder="Enter your lastname" type="name" name="lastname" ngModel [required]="true">
</mat-form-field>
<mat-form-field appearance="fill" class="example-full-width">
  <mat-label>Username</mat-label>
  <input matInput placeholder="Enter your username" type="username" name="username" ngModel [required]="true">
  username
</mat-form-field>
<mat-form-field appearance="fill" class="example-full-width" *ngIf="!isLoginMode">
  <mat-label>Email</mat-label>
  <input matInput placeholder="Enter your email" type="email" name="email" ngModel [required]="true">
</mat-form-field>
<mat-form-field appearance="fill" class="example-full-width">
  <mat-label>Password</mat-label>
  <input matInput placeholder="Enter your password" type="password" name="password" ngModel [required]="true">
</mat-form-field>
```

Figura 3.8: Secțiune de cod cu câmpuri care conțin directiva mat-form-field

În funcție de valoarea variabilei isLoginMode, sunt afișate sau ascunse câmpurile pentru nume, prenume și email. Câmpurile sunt definite folosind directiva mat-form-field și sunt validate utilizând atributul [required].

```

<div>
  <button mat-raised-button color="primary" type="submit" [disabled]="!authForm.valid">{{ isLoginMode ?
    'Login' : 'Sign Up' }}</button>
</div>
<div>
  <span>
    {{
      isLoginMode ? 'Don\'t have an account?' : 'Already have an account?'
    }}
  </span>
  <button mat-flat-button type="button" (click)="onSwitchMode()">{{ isLoginMode ? 'Sign Up' : 'Login'
    }}</button>
</div>

```

Figura 3.9: Secțiune de cod cu buton de comutare

Această secțiune permite utilizatorului să comute între modurile de autentificare și înregistrare. Textul și butonul corespunzător se schimbă în funcție de valoarea variabilei `isLoginMode`. Aceasta este realizată prin intermediul directivei `*ngIf` și prin gestionarea evenimentului click al butonului.

Componentele sunt definite prin intermediul decoratorului `@Component` și pot fi considerate ca și clase cu metode și proprietăți specifice. Componentele sunt definite prin intermediul acestui decorator, care primește un obiect de metadate ce descrie comportamentul și aspectul componentei.

```

@Component({
  selector: 'app-stopwatch',
  templateUrl: './stopwatch.component.html',
  styleUrls: ['./stopwatch.component.css']
})
export class StopwatchComponent {

  @Input() start: boolean;
  @Input() showTimerControls: boolean;

  ngOnChanges(changes: SimpleChanges) {
    console.log(changes['start']);
    if (changes['start'].currentValue) {
      this.startTimer();
    }
    else {
      this.clearTimer();
    }
  }
}

```

Figura 3.10: Fragment de cod din componenta Stopwatch

Componenta este definită prin intermediul acestui decorator, care specifică selectorul (`selector: 'app-stopwatch'`) utilizat pentru a include componenta în template-urile altei componente, șablonul HTML (`templateUrl: './stopwatch.component.html'`) care definește aspectul componentei și fișierele CSS (`styleUrls: ['./stopwatch.component.css']`) asociate componentei pentru stilizare.

3.2.4 Servicii

Serviciile reprezintă unul dintre cele mai importante concepte în arhitectura Angular. Ele sunt utilizate pentru a împărți și gestiona logica de afaceri, funcționalitățile comune și accesul la resurse externe în întreaga aplicație. Serviciile furnizează o modalitate eficientă de a separa preocupările și de a încapsula funcționalitățile specifice într-un mod reutilizabil și modular.

```
@Injectable({
  providedIn: 'root'
})
export class AuthService {
  constructor(private http: HttpClient) { }

  login(user: User) {
    return this.http.post(environment.userManagement.baseUrl + 'account/login', user);
  }

  register(user: User) {
    return this.http.post(environment.userManagement.baseUrl + 'account/register', user);
  }

  storeToken(tokenVal: string) {
    localStorage.setItem('token', tokenVal);
  }

  getToken() {
    return localStorage.getItem('token');
  }

  isLoggedIn(): boolean {
    return !!localStorage.getItem('token');
  }
}
```

Figura 3.11: Captura de ecran cu serviciul AuthService

Serviciul AuthService utilizează injectarea de dependențe pentru a obține o instanță a serviciului HttpClient, care este folosit pentru a efectua cereri HTTP către serverul API. Astfel, serviciul se bazează pe HttpClient pentru comunicarea cu serverul pentru a realiza autentificarea și înregistrarea utilizatorilor. Prin intermediul metodelor sale (login(), register(), storeToken(), getToken(), isLoggedIn()), AuthService oferă funcționalități precum trimiterea cererilor de autentificare, gestionarea tokenului de autentificare și verificarea stării de autentificare a utilizatorului.

Prin separarea funcționalităților de autentificare într-un serviciu dedicat, aplicația devine mai modulară, ușor de întreținut și extensibilă. Alte componente pot utiliza serviciul AuthService prin injectarea sa în constructorul lor, permițându-le să beneficieze de funcționalitățile de autentificare fără a duplica codul și logica asociată.

3.3 Arhitectura serverului

Arhitectura curată în ASP.NET Core Web API este o abordare arhitecturală care promovează modularitatea, separarea responsabilităților și testabilitatea în dezvoltarea aplicațiilor web. Aceasta se bazează pe principii precum împărțirea proiectului în straturi și dependențe inversate pentru a obține un design flexibil și modular.

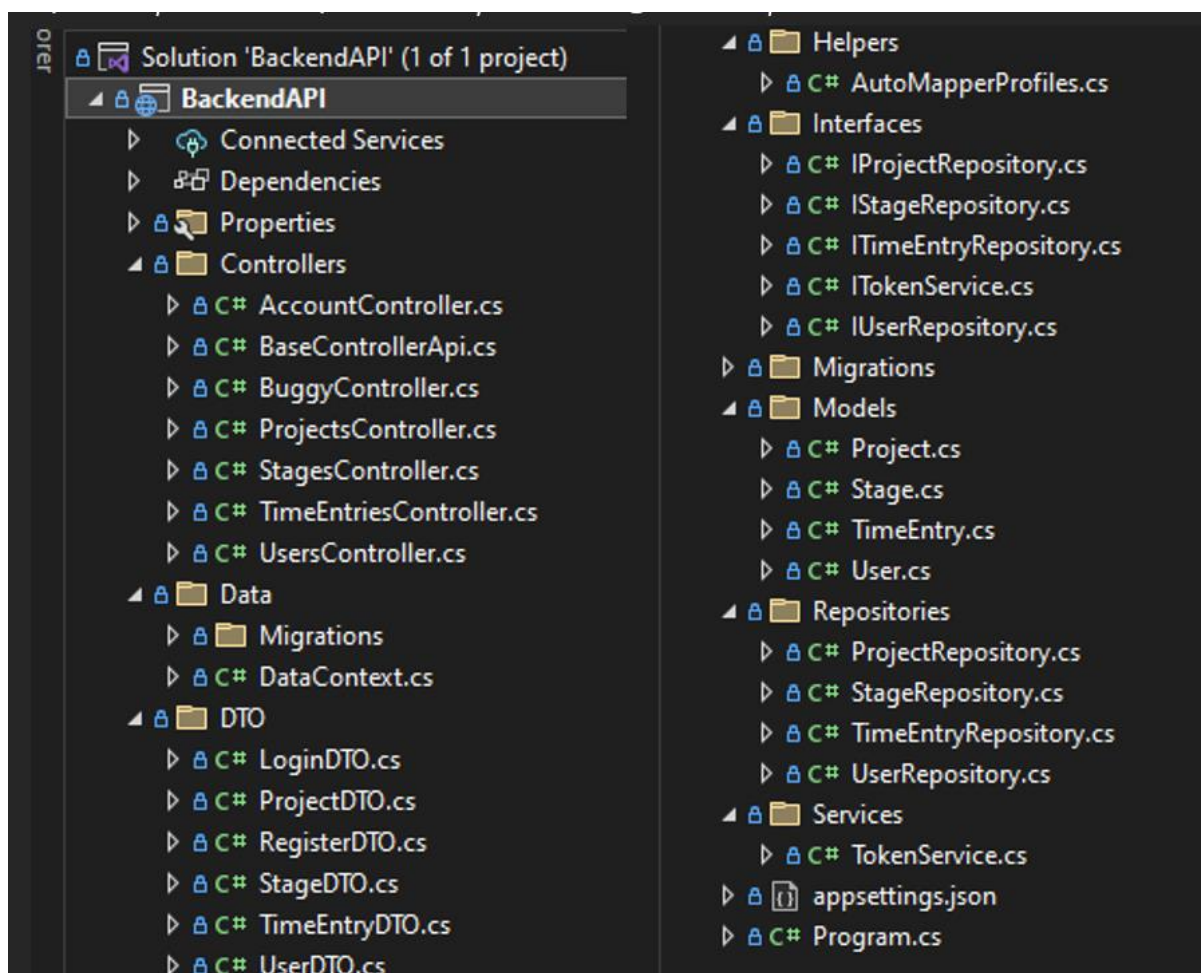


Figura 3.12: Structura serverului

Aceste foldere reprezintă componente cheie în structura aplicației și joacă un rol crucial în organizarea și modularizarea codului sursă. Ele contribuie la crearea unei arhitecturi robuste și scalabile, care facilitează dezvoltarea, întreținerea și extinderea aplicației pe termen lung. Fiecare folder are un scop bine definit și îndeplinește anumite responsabilități, asigurând separarea logică a funcționalităților și a modulelor.

3.3.1 Modele

Modelele reprezintă componente esențiale ale arhitecturii unei aplicații și joacă un rol central în reprezentarea și manipularea datelor specifice domeniului aplicației. Acestea sunt utilizate pentru a defini structura și comportamentul datelor, precum și relațiile între acestea. Modelele capturează entități, obiecte de valoare și alte concepte specifice domeniului și constituie nucleul logicii aplicației.

```
public class Stage
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    [Required]
    public int Id { get; set; }

    public int ProjectId { get; set; }

    [Required]
    [MaxLength(50)]
    public string Name { get; set; }

    [Required]
    [MaxLength(150)]
    public string Description { get; set; }

    [Required]
    [MaxLength(50)]
    public string Status { get; set; }

    [Required]
    public DateTime Deadline { get; set; }

    public ICollection<TimeEntry> TimeEntries { get; set; } = new List<TimeEntry>();
}
```

Figura 3.13: Exemplu de clasă model

Clasa Stage reprezintă o entitate cheie în cadrul domeniului aplicației BackendAPI și este utilizată pentru gestionarea informațiilor despre etapele proiectelor. Aceasta încapsulează proprietățile și funcționalitățile specifice etapelor, oferind o abstracție a datelor și comportamentului asociate acestora în cadrul domeniului de business al aplicației.

Entitățile în ASP.Net Core Web API sunt de obicei implementate ca și clase, care conțin proprietăți care reprezintă caracteristicile și atributele entității respective. Aceste proprietăți pot fi adnotate cu atribute speciale, cum ar fi [Required] sau [MaxLength], pentru a specifica reguli de validare și constrângeri asupra datelor.

3.3.2 Obiecte de transfer de date(DTO)¹⁶

Obiectele de transfer de date reprezintă o clasă utilizată pentru a transfera date între straturi sau componente diferite ale aplicației. Scopul principal al DTO-urilor este de a facilita transferul eficient și sigur al datelor între client și server.

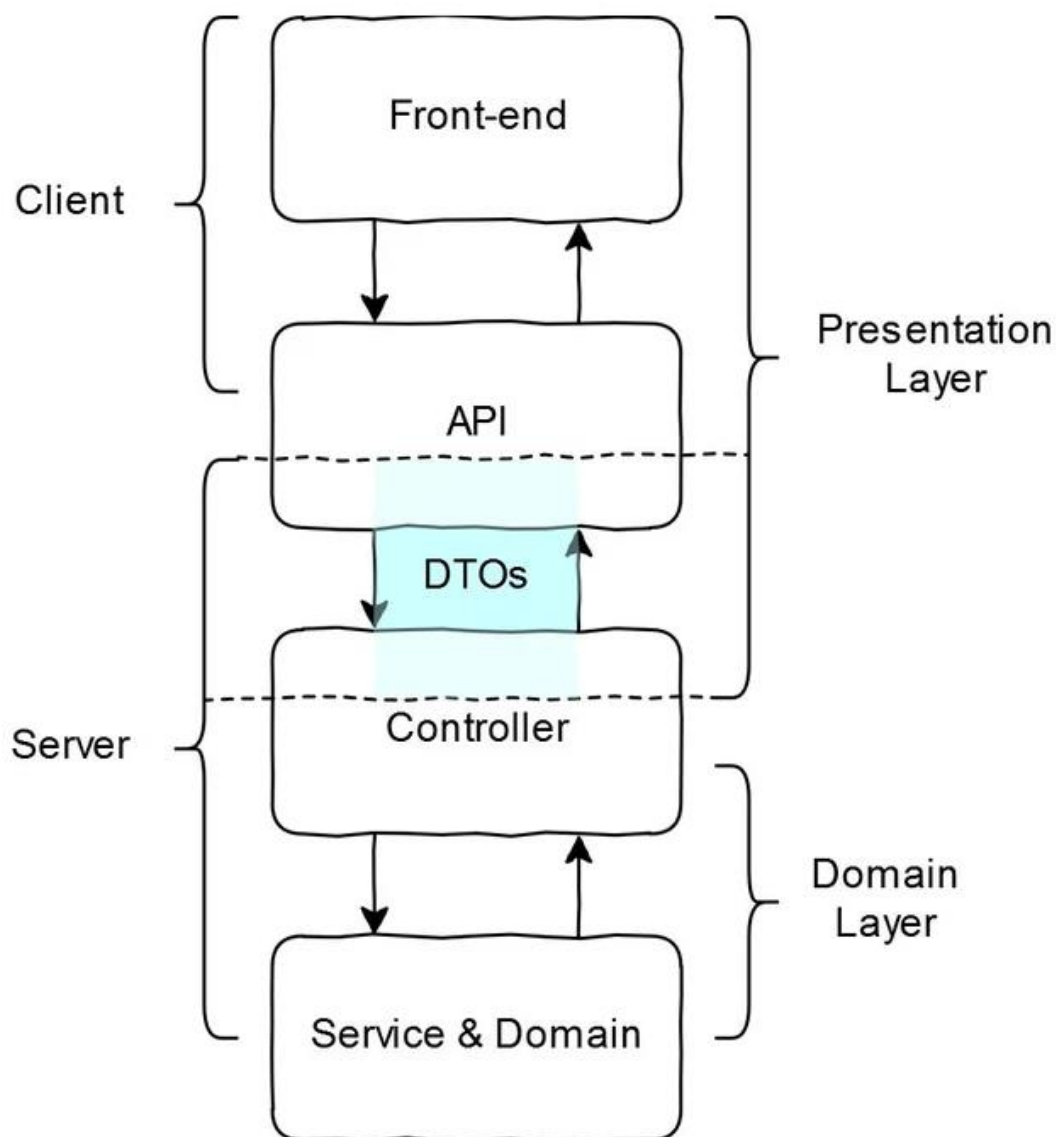


Figura 3.14: Rolul DTO^{xiv}

DTO-urile sunt adesea utilizate pentru a defini structura și conținutul datelor care sunt trimise sau primite prin intermediul serviciilor API. Acestea pot fi utilizate pentru a izola entitățile de baza de date sau alte modele de date interne și pentru a expune doar informațiile relevante către client.

¹⁶ Data Transfer Object (DTO)

3.3.3 Controlere

În cadrul unei aplicații ASP.Net Core Web API, controlerele reprezintă componente cheie responsabile de gestionarea cererilor HTTP primite de la client și de furnizarea răspunsurilor corespunzătoare. Ele acționează ca intermediari între client și serviciile backend, preluând datele de intrare, efectuând prelucrări și returnând rezultatele.

Controlerele în ASP.Net Core Web API sunt definite ca și clase care utilizează atributul [ApiController] și sunt decorate cu diferite atribute și adnotări pentru a configura comportamentul și rutarea cererilor HTTP.

Clasa TimeEntriesController reprezintă un controler în cadrul aplicației ASP.Net Core Web API responsabil de gestionarea cererilor HTTP referitoare la entitatea TimeEntry. Acesta este un exemplu de implementare a unui controler API simplu, care utilizează o interfață ITimeEntryRepository pentru a accesa și manipula datele asociate cu entitatea TimeEntry.

```
[Route("api/[controller]")]
[ApiController]
public class TimeEntriesController : ControllerBase
{
```

Figura 3.15: Atribut ApiController

Controlerul este atribuit cu atributul [Route("api/[controller]")], ceea ce definește ruta de bază pentru toate acțiunile din acest controler. De exemplu, pentru această clasă, ruta de bază va fi "api/TimeEntries". Controlerul utilizează și atributul [ApiController], care oferă suport integrat pentru validarea automată a cererilor, serializarea și alte funcționalități comune pentru controlerele API.

```
[HttpGet]
public async Task<ICollection<TimeEntry>> GetAllTimeEntries()
{
    return await _timeEntryRepository.GetAllTimeEntriesAsync();
}
```

Figura 3.16: Metoda din cadrul controlerului TimeEntriesController

Metoda GetAllTimeEntries este o acțiune definită în controlerul TimeEntriesController și este marcată cu atributul [HttpGet]. Această acțiune este responsabilă de gestionarea cererilor HTTP de tip GET adresate către ruta "api/TimeEntries".

3.3.4 Repository

Un repository este responsabil pentru abstractizarea detaliilor specifice ale accesului la date și oferă o interfață consistentă și simplificată pentru a realiza operații de citire, scriere, actualizare și ștergere a datelor. Astfel, repository-ul oferă un nivel de abstractizare între aplicație și resursele de date, permițând o separare clară a responsabilităților și o gestionare mai ușoară a datelor.

Utilizarea unui repository în cadrul controlerului permite separarea logică dintre gestionarea cererilor HTTP și accesul la date. Astfel, controlerul nu trebuie să cunoască detalii specifice despre modul în care sunt stocate și accesate datele, ci doar să utilizeze interfețele oferite de repository pentru a efectua operațiile dorite.

```
public class TimeEntryRepository : ITimeEntryRepository
{
    private readonly DataContext _context;

    public TimeEntryRepository(DataContext context)
    {
        _context = context;
    }

    public async Task<ICollection<TimeEntry>> GetAllTimeEntriesAsync()
    {
        return await _context.TimeEntries.ToListAsync();
    }
}
```

Figura 3.17: Exemplu de repository

Repository-ul `TimeEntryRepository` implementează interfața `ITimeEntryRepository` și oferă metode pentru a realiza operații asupra entităților `TimeEntry`. Aceste metode includ obținerea tuturor înregistrărilor de tip `TimeEntry`, obținerea unei înregistrări după ID, obținerea înregistrărilor asociate unei anumite etape (stageId), actualizarea unei înregistrări și crearea unei noi înregistrări.

Acest repository este injectat în controlerul `TimeEntriesController` prin intermediul constructorului, utilizând mecanismul de injecție de dependențe al cadrului de lucru ASP.NET Core.

Prin utilizarea metodei `ToListAsync` din Entity Framework Core, operația de acces la date este efectuată în mod asincron, evitând blocarea firului de execuție și îmbunătățind performanța generală a aplicației.

3.4 Arhitectura bazei de date

În cadrul aplicației mele, am utilizat o bază de date relațională, gestionată cu ajutorul cadrului de lucru Entity Framework Core. Aceasta asigură o structură coerentă și organizată pentru stocarea informațiilor despre utilizatori, proiecte, etape și înregistrări de timp.

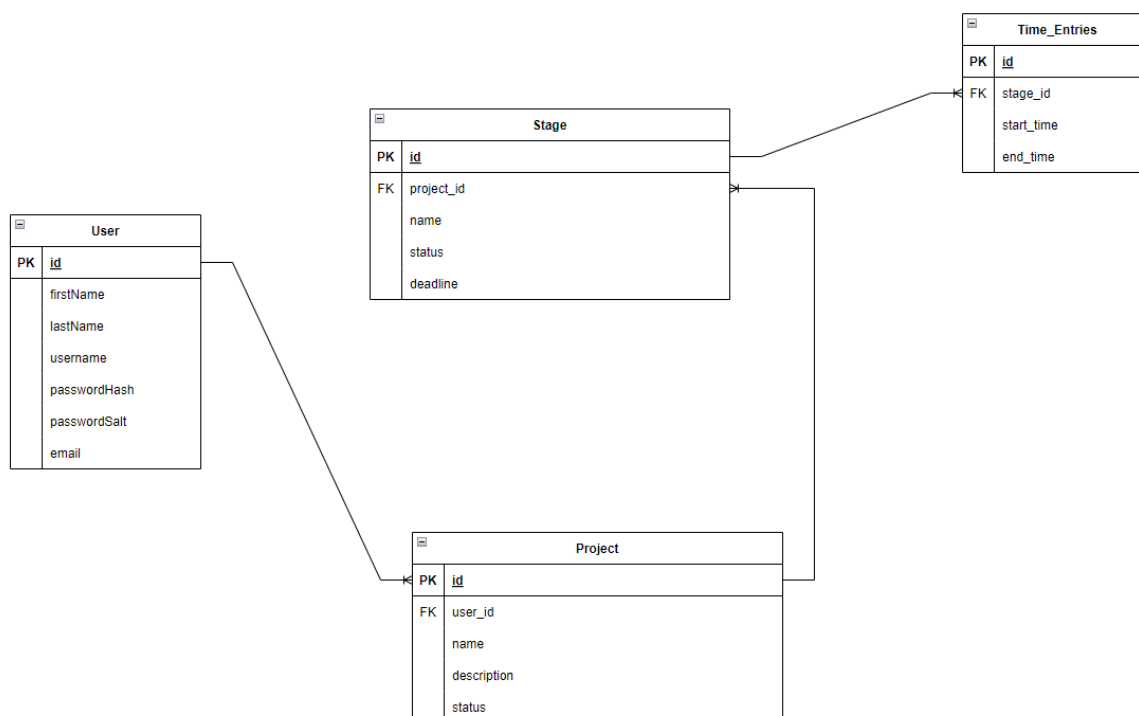


Figura 3.18: Schema bazei de date

Tabelul "Users" conține informații despre utilizatori, inclusiv nume, adrese de email și date de autentificare.

Tabelul "Projects" stochează detalii despre proiecte, cum ar fi descrierea, numele și statusul acestora. Există o relație între tabelul "Projects" și tabelul "Users", care indică asocierea fiecărui proiect cu un utilizator.

Tabelul "Stages" reprezintă etapele unui proiect și conține informații despre nume, descriere, status și termen limită. Acesta este legat de tabelul "Projects" prin intermediul unei chei străine care indică proiectul căruia aparține fiecare etapă.

Tabelul "TimeEntries" stochează înregistrări de timp asociate cu etapele. Fiecare înregistrare de timp conține informații despre timpul de început și sfârșit. Există o relație între tabelul "TimeEntries" și tabelul "Stages", care indică etapa la care se referă fiecare înregistrare de timp.

3.5 Implementare server

Serverul oferă un API RESTful care va permite comunicarea și schimbul de date între client și server. Acest API va defini operațiile și resursele disponibile pentru client și va respecta principiile REST pentru a asigura o arhitectură eficientă și scalabilă. La implementarea serverului am utilizat de asemenea și mapare obiect-relațională¹⁷ pentru a realiza interacțiunea cu baza de date. Pentru a asigura securitatea aplicației, serverul va implementa autentificare și autorizare utilizând JWT (JSON Web Tokens). Acest mecanism va permite autentificarea utilizatorilor și gestionarea accesului la resursele protejate.

3.5.1 Interacțiunea cu baza de date prin ORM

ORM este o tehnică care facilitează interacțiunea între baza de date relațională și codul aplicației, oferind o modalitate de a mapa structurile de date și tabelele bazei de date la obiecte din codul sursă.

```
public class DataContext : DbContext
{
    public DataContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<User> Users { get; set; }
    public DbSet<Project> Projects { get; set; }
    public DbSet<Stage> Stages { get; set; }
    public DbSet<TimeEntry> TimeEntries { get; set; }
}
```

Figura 3.19: Clasa DataContext

Clasa DataContext reprezintă clasa de bază a bazei de date în cadrul aplicației. Aceasta este derivată din clasa DbContext furnizată de Microsoft Entity Framework Core și definește entitățile și relațiile asociate bazei de date. Constructorul clasei DataContext primește un obiect de tip DbContextOptions care conține opțiunile de configurare a contextului bazei de date.

În clasa DataContext, sunt definite proprietăți de tip DbSet<T> pentru fiecare entitate a aplicației, cum ar fi Users, Projects, Stages și TimeEntries. Aceste proprietăți reprezintă tabelele corespunzătoare din baza de date și permit interacțiunea cu acestea prin operații de creare, citire, actualizare și ștergere.

¹⁷ Object-Relational Mapping(ORM)

Clasa DataContext acționează ca un mediu în care entitățile sunt mapate pe tabelele bazei de date și sunt gestionate prin intermediul cadrului de lucru Entity Framework Core. Ea facilitează comunicarea între aplicație și baza de date, permițând operații de interogare și manipulare a datelor.

```
public class User
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    [Required]
    public int Id { get; set; }

    [Required]
    [MaxLength(50)]
    public string Username { get; set; }

    [Required]
    [MaxLength(50)]
    public string FirstName { get; set; }

    [Required]
    [MaxLength(50)]
    public string LastName { get; set; }

    public byte[] PasswordHash { get; set; }

    public byte[] PasswordSalt { get; set; }

    [Required]
    [MaxLength(100)]
    public string Email { get; set; }

    public ICollection<Project> Projects { get; set; }
}
```

Figura 3.20: Clasa model User

Clasa User reprezintă o entitate în cadrul aplicației și reprezintă un utilizator al sistemului. Aceasta conține proprietăți care definesc caracteristicile și informațiile asociate unui utilizator.

Proprietatea Id reprezintă identificatorul unic al utilizatorului și este marcată cu atributul [DatabaseGenerated(DatabaseGeneratedOption.Identity)], ceea ce înseamnă că valoarea acestui câmp este generată automat de către baza de date la momentul inserării unui nou utilizator în sistem.

Proprietățile `UserName`, `FirstName` și `LastName` reprezintă informații despre numele utilizatorului și sunt marcate ca `[Required]`, indicând că aceste câmpuri trebuie completate și nu pot fi goale.

`PasswordHash` și `PasswordSalt` reprezintă hash-ul și salt-ul asociate parolei utilizatorului. Acestea sunt utilizate în scopuri de securitate pentru stocarea parolei într-o formă criptată, fiind de tip `byte[]`.

Proprietatea `Email` reprezintă adresa de email a utilizatorului și este obligatorie. Este limitată la maximum 100 de caractere și este de tip șir de caractere (`string`).

Proprietatea `Projects` este o colecție de obiecte de tip `Project` și indică faptul că un utilizator poate fi asociat cu mai multe proiecte.

```
public class UserDTO
{
    public int Id { get; set; }
    public string UserName { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public ICollection<ProjectDTO> Projects { get; set; }
    public string Token { get; set; }
}
```

Figura 3.21: Clasa UserDTO

Clasa `UserDTO` este o reprezentare simplificată a clasei `User` din modelul de date (`BackendAPI.Models.User`). Este folosită pentru a transfera informații despre un utilizator între diferite componente ale aplicației sau între server și client. Aceasta ajută la separarea detaliilor specifice ale modelului de date și permite transmiterea doar a informațiilor relevante către consumatorii de date.

Proprietățile din clasa `UserDTO` sunt echivalente cu cele din clasa `User`, cu excepția proprietății `Token`, care este specifică clasei `UserDTO`. Proprietatea `Token` este adăugată pentru a transmite un token de autentificare asociat utilizatorului, care poate fi utilizat pentru a autoriza accesul la anumite resurse sau acțiuni în cadrul aplicației.

3.5.2 Definirea serviciilor REST

Serviciile REST reprezintă punctele de intrare în aplicație prin intermediul cărora se pot accesa și manipula resursele.

```
public class AccountController : BaseControllerApi
{
    private readonly ITokenService _tokenService;
    private readonly DataContext _context;
    private readonly IMapper _mapper;

    public AccountController(DataContext context, ITokenService tokenService, IMapper mapper)
    {
        _tokenService = tokenService;
        _context = context;
        _mapper = mapper;
    }
}
```

Figura 3.22: Controlerul AccountController

Clasa AccountController este o clasă de controler în cadrul serverului, responsabilă de gestionarea operațiilor legate de autentificare și autorizare. În constructorul clasei AccountController, sunt injectate trei dependențe necesare pentru funcționarea controlerului: DataContext, ITokenService, IMapper.

```
[HttpPost("register")]
public async Task<ActionResult<UserDTO>> Register(RegisterDTO registerDTO)
{
    if (await UserExists(registerDTO.Username))
        return BadRequest("Username not available, choose another one");

    using var hmac = new HMACSHA512();
    var user = new User
    {
        UserName = registerDTO.Username.ToLower(),
        FirstName = registerDTO.FirstName,
        LastName = registerDTO.LastName,
        PasswordHash = hmac.ComputeHash(Encoding.UTF8.GetBytes(registerDTO.Password.ToLower())),
        PasswordSalt = hmac.Key,
        Email = registerDTO.Email.ToLower()
    };
    _context.Users.Add(user);
    await _context.SaveChangesAsync();

    var newUserDTO = _mapper.Map<UserDTO>(user);
    newUserDTO.Token = _tokenService.CreateToken(user);
    return newUserDTO;
}
```

Figura 3.23: Metoda Register din clasa AccountController

Pentru a crea un nou utilizator, se folosește clasa HMACSHA512 pentru generarea hash-ului parolei. Parola este convertită la litere mici folosind ToLower() pentru a asigura coerența. Hash-ul rezultat și cheia generată sunt asignate proprietăților PasswordHash și

PasswordSalt ale obiectului User. Aceste valori vor fi stocate în baza de date pentru verificarea ulterioară a parolei.

Apoi, obiectul User este adăugat în setul de utilizatori (Users) al contextului de date _context și se apelează metoda SaveChangesAsync() pentru a salva modificările în baza de date.

După înregistrare cu succes, se realizează o mapare a obiectului User într-un obiect UserDTO folosind serviciul de mapare _mapper. Token-ul JWT este generat pentru noul utilizator utilizând serviciul _tokenService și este asignat proprietății Token a obiectului UserDTO.

```
[HttpPost("login")]
public async Task<ActionResult<UserDTO>> Login(LoginDTO loginDTO)
{
    var user = await _context.Users.FirstOrDefaultAsync(u => u.UserName == loginDTO.Username);
    if (user == null)
        return Unauthorized("invalid username");

    using var hmac = new HMACSHA512(user.PasswordSalt);
    var computedHash = hmac.ComputeHash(Encoding.UTF8.GetBytes(loginDTO.Password));

    for (int i = 0; i < computedHash.Length; i++)
    {
        if (computedHash[i] != user.PasswordHash[i])
            return Unauthorized("invalid password");
    }

    var newUserDTO = _mapper.Map<UserDTO>(user);
    newUserDTO.Token = _tokenService.CreateToken(user);
    return newUserDTO;
}
```

Figura 3.24: Metoda Login din clasa AccountController

Metoda primește un obiect LoginDTO ca parametru, care conține numele de utilizator și parola introduse de utilizator în momentul autentificării.

Pentru verificarea parolei, se folosește cheia (PasswordSalt) stocată în obiectul User găsit în baza de date. Se utilizează clasa HMACSHA512 cu această cheie pentru a calcula hash-ul parolei introduse în formularul de autentificare (loginDTO.Password). Se compară hash-ul obținut cu hash-ul stocat în obiectul User pentru a valida parola. Dacă hash-urile nu se potrivesc, se returnează un răspuns Unauthorized cu mesajul "invalid password".

Dacă autentificarea este reușită, se realizează o mapare a obiectului User într-un obiect UserDTO folosind serviciul de mapare _mapper. Token-ul JWT este generat pentru utilizatorul autentificat utilizând serviciul _tokenService și este asignat proprietății Token a obiectului UserDTO.

3.5.3 Autentificare și autorizare cu JWT

Autentificarea și autorizarea cu JWT reprezintă o metodă populară și securizată de gestionare a identității utilizatorilor în aplicațiile web. JWT este un standard deschis care definește un format compact și autenticat pentru transmiterea de informații între părți în format JSON.

```
public class TokenService : ITokenService
{
    private readonly SymmetricSecurityKey _key;

    public TokenService(IConfiguration configuration)
    {
        _key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(configuration["TokenKey"]));
    }

    public string CreateToken(User user)
    {
        var claims = new List<Claim>
        {
            new Claim(JwtRegisteredClaimNames.NameId, user.UserName)
        };

        var creds = new SigningCredentials(_key, SecurityAlgorithms.HmacSha256Signature);
        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);
    }
}
```

Figura 3.25: Clasa TokenService

Clasa TokenService este un serviciu în cadrul aplicației responsabil de generarea și semnarea token-urilor JWT (JSON Web Token). Metoda CreateToken primește un obiect de tip User și generează un token JWT pe baza acestuia. În cadrul metodei, se definesc claim-urile (informațiile) asociate utilizatorului și se realizează semnătura token-ului utilizând o cheie secretă. Token-ul rezultat este returnat sub forma unui șir de caractere și poate fi utilizat pentru autentificare și autorizare în aplicație. Astfel, TokenService oferă un mecanism sigur și eficient pentru gestionarea autentificării și securității în aplicație.

3.6 Implementare interfata

3.6.1 Definirea componentelor Angular pentru interfață

În implementarea aplicației, am decis să utilizez componentele Angular pentru a crea o interfață utilizator modernă, interactivă și ușor de utilizat. Aceste componente oferă o structură modulară și reutilizabilă, care mi-a permis să dezvoltăm rapid și eficient diferite elemente ale interfeței.

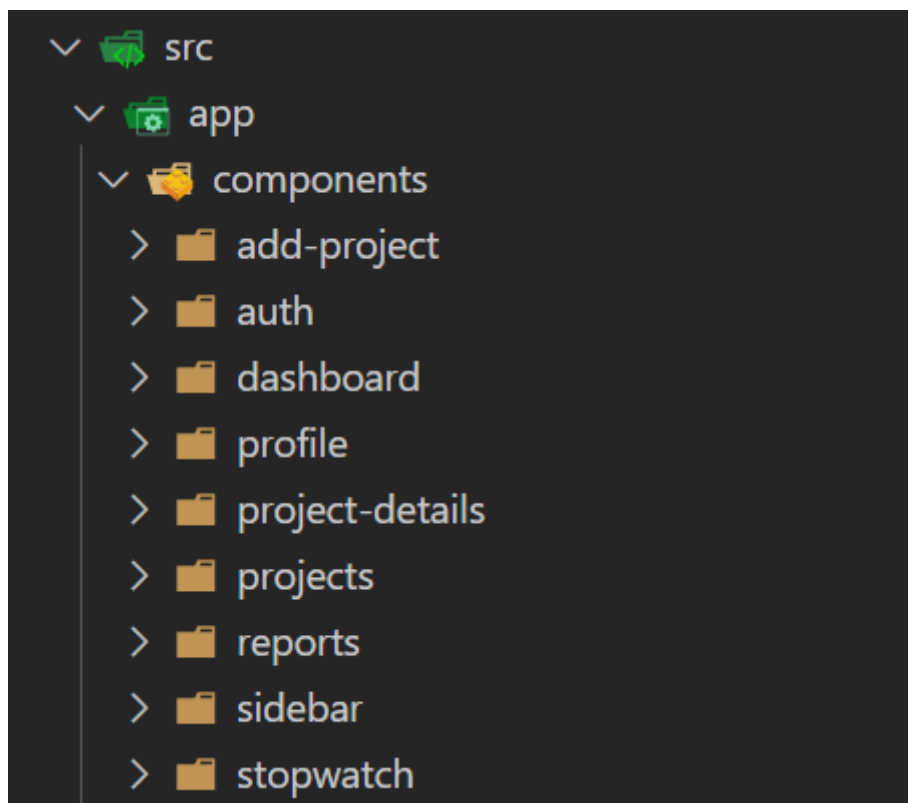


Figura 3.26: Componentele Angular din aplicație

Prin utilizarea componentelor Angular, am putut defini și organiza elementele interfeței într-un mod clar și coerent. Am creat componente separate pentru bara de navigare, secțiunea principală și lista de proiecte, ceea ce a facilitat gestionarea și reutilizarea codului în întreaga aplicație. De asemenea, am putut utiliza caracteristici precum legături între componente și transmiterea datelor între ele, pentru a asigura o interacțiune fluidă și sincronizată între diversele elemente ale interfeței.

Componenta AuthComponent este responsabilă de gestionarea procesului de autentificare și înregistrare a utilizatorilor în cadrul aplicației. Această componentă conține un formular care permite utilizatorilor să completeze informațiile necesare pentru

autentificare sau înregistrare. La trimiterea formularului, componenta utilizează serviciul AuthService pentru a efectua cererile HTTP corespunzătoare către server. Componenta utilizează, de asemenea, alte dependențe Angular, cum ar fi NgForm pentru a accesa și valida datele introduse în formularul de autentificare/inregistrare și MatSnackBar pentru a afișa notificări de succes sau eroare.

Dacă utilizatorul selectează modul de autentificare, componenta AuthComponent folosește metoda login() din serviciul AuthService pentru a trimite cererea de autentificare la server. În cazul în care autentificarea este reușită, utilizatorul este redirecționat către o altă pagină, iar un mesaj de notificare de succes este afișat. În caz contrar, un mesaj de eroare este afișat.

```
isLoginMode: boolean = true;
onSubmit(form: NgForm) {
  if (form.valid) {
    if (this.isLoginMode) {
      this.authService.login(form.value).subscribe({
        next: async (res: any) => {
          form.reset();
          this.authService.storeToken(res.token, res.id);
          this.authService.storeUser(res);
          this._snackBar.open('Login Successful', 'Dismiss', {
            duration: 3000,
            horizontalPosition: 'center',
            verticalPosition: 'top',
          });
          this.router.navigate(['sidebar']);
        },
        error: (err: any) => {
          this._snackBar.open('Login Failed', 'Dismiss', {
            duration: 3000,
            horizontalPosition: 'center',
            verticalPosition: 'top',
          });
        }
      });
    }
  }
}
```

Figura 3.27: Partea de autentificare a funcției onSubmit

Pe de altă parte, dacă utilizatorul selectează modul de înregistrare, componenta AuthComponent folosește metoda register() din serviciul AuthService pentru a trimite cererea de înregistrare la server. Dacă înregistrarea este reușită, utilizatorul este redirecționat către o altă pagină, iar un mesaj de notificare de succes este afișat. În caz contrar, un mesaj de eroare este afișat.

```

else {
  this.authService.register(form.value).subscribe({
    next: (res: any) => {
      form.reset();
      this.authService.storeToken(res.token, res.id);
      this._snackBar.open('Registration Successful', 'Dismiss', {
        duration: 3000,
        horizontalPosition: 'center',
        verticalPosition: 'top',
      });
      this.router.navigate(['sidebar']);
    },
    error: (err: any) => {
      this._snackBar.open('Registration Failed', 'Dismiss', {
        duration: 3000,
        horizontalPosition: 'center',
        verticalPosition: 'top',
      });
    }
  })
}
}

```

Figura 3.28: Partea de înregistrare a funcției onSubmit

3.6.2 Gestionarea datelor de la server

Pentru a facilita comunicarea cu serverul, am creat servicii Angular care au fost responsabile de realizarea cererilor HTTP și de gestionarea răspunsurilor primite. Aceste servicii au fost configurate pentru a utiliza adresele URL corecte și metodele HTTP corespunzătoare pentru a obține, crea, actualiza sau șterge datele din baza de date.

Aplicația include trei servicii Angular: AuthService, DialogService și ProjectService, și UserService. Fiecare dintre aceste servicii are un set specific de funcționalități pentru a gestiona anumite aspecte ale aplicației.

AuthService gestionează autentificarea și înregistrarea utilizatorilor. Acesta utilizează obiectul HttpClient pentru a efectua cereri HTTP către serverul de autentificare. Metodele login și register trimit cereri POST către endpoint-urile corespunzătoare pentru autentificare și înregistrare. Metoda storeToken este responsabilă de stocarea informațiilor relevante în stocarea locală a browser-ului pentru utilizarea ulterioară. Alte metode, cum ar fi getToken și isLoggedIn, furnizează informații relevante despre utilizator și autentificare.

```

export class AuthService {
  constructor(private http: HttpClient) { }

  login(user: User) {
    return this.http.post(environment.userManagement.baseUrl + 'account/login', user);
  }

  register(user: User) {
    return this.http.post(environment.userManagement.baseUrl + 'account/register', user);
  }

  storeToken(tokenVal: string, userId: number) {
    localStorage.setItem('token', tokenVal);
    localStorage.setItem('userId', userId.toString());
  }

  getToken() {
    return localStorage.getItem('token');
  }

  isLoggedIn(): boolean {
    return !!localStorage.getItem('token');
  }
}

```

Figura 3.29: Serviciul AuthService

DialogService este responsabil de gestionarea dialogurilor modale în aplicație. Aceasta utilizează MatDialog din pachetul Angular Material pentru a deschide și gestiona dialogurile. Metoda openDialog deschide un dialog modal folosind componenta AddProjectComponent și ascultă evenimentul afterClosed pentru a obține rezultatul dialogului.

```

@Injectable({
  providedIn: 'root'
})
export class DialogService {

  constructor(private dialog: MatDialog) { }

  openDialog() {
    let dialogRef = this.dialog.open(AddProjectComponent, {
      width: '400px',
    });

    dialogRef.afterClosed().subscribe(result => {
      console.log(`Dialog result: ${result}`);
    });

    dialogRef.close();
  }
}

```

Figura 3.30: Serviciul DialogService

ProjectService furnizează funcționalități pentru gestionarea proiectelor în aplicație. Metodele precum getUserProjects, createProject, getProject și createStage trimit cereri HTTP către endpoint-urile corespunzătoare pentru a obține și a crea proiecte și etape în cadrul acestora.

```
export class ProjectService {
  constructor(private http: HttpClient) { }

  getUserProjects(): Observable<Project[]> {
    const idLS = parseInt(localStorage.getItem("userId"));
    return this.http.get<Project[]>(environment.userManagement.baseUrl + 'projects/userId/' + idLS);
  }

  createProject(project: Project) {
    return this.http.post(environment.userManagement.baseUrl + 'projects/createProject', project);
  }
}
```

Figura 3.31: Secțiune de cod din serviciul ProjectService

3.6.3 Logica de rutare și navigare între pagini

Această funcționalitate este gestionată în Angular prin intermediul modulului RouterModule și a serviciului Router.

În implementarea aplicației, am definit mai multe rute pentru diferite pagini ale aplicației. Aceste rute sunt definite în fișierul app-routing.module.ts, unde asociez fiecare rută cu o componentă specifică.

```
const routes: Routes = [
  { path: '', redirectTo: 'auth', pathMatch: 'full' },
  { path: 'auth', component: AuthComponent },
  {
    path: 'sidebar', component: SidebarComponent, children: [
      { path: '', redirectTo: 'dashboard', pathMatch: 'full' },
      { path: 'dashboard', component: DashboardComponent },
      { path: 'projects', component: ProjectsComponent },
      { path: 'reports', component: ReportsComponent },
      { path: 'profile', component: ProfileComponent }
    ]
  },
  { path: 'projectDetails/:id', component: ProjectDetailsComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  providers: []
})

export class AppRoutingModule { }
```

Figura 3.32: Logica de rutare din fișierul app-routing.module.ts

Modulul începe prin importarea necesară a modulelor `NgModule` și `RouterModule` din pachetul `@angular/router`. Apoi, am definit rutele într-un obiect `routes` de tip `Routes`. Fiecare rută este reprezentată de un obiect care specifică `path`-ul și componenta asociată.

Rutele pot avea și copii, așa cum este cazul pentru ruta `'sidebar'`, unde se definește un set de rute pentru copiii acesteia. Astfel, am rute pentru `'dashboard'`, `'projects'`, `'reports'` și `'profile'`, care sunt componente afișate în interiorul componentei `'SidebarComponent'`.

După definirea rutelor, am utilizat metoda `RouterModule.forRoot()` pentru a configura rutarea cu rutele definite. Modulul de rutare este exportat prin adăugarea `RouterModule` în secțiunea `exports` a modulului.

Prin intermediul serviciului `Router`, pot accesa diferite funcționalități legate de rutare, cum ar fi navigarea către o rută specifică, redirecționarea către o altă rută, obținerea informațiilor despre ruta curentă și multe altele. De asemenea, pot utiliza serviciul `Router` pentru a gestiona parametrii de rută și pentru a comunica între componente în cadrul aplicației prin intermediul rutelor.

Serviciul `Router` poate fi injectat în componente sau alte servicii utilizând injectia de dependență. Astfel, putem utiliza metodele și proprietățile furnizate de acest serviciu pentru a controla fluxul de navigare al aplicației și pentru a asigura o interacțiune fluidă între diferitele pagini.

4. Concluzii

În această lucrare, am prezentat procesul de dezvoltare a unei aplicații full stack, utilizând Angular pentru interfața utilizator și ASP.NET Core Web API pentru serverul backend. Am explorat diferite aspecte ale dezvoltării aplicației, începând de la proiectarea arhitecturii și a bazelor de date, până la implementarea funcționalităților principale și gestionarea autentificării și autorizării utilizatorilor.

Prin intermediul acestui proiect, am învățat să proiectez o arhitectură coerentă și eficientă, care să permită o comunicare fluidă între componentele frontend și backend. Am înțeles importanța separării responsabilităților și utilizării principiilor SOLID pentru a crea cod modular și ușor de întreținut.

De asemenea, am învățat să lucrez cu baze de date și să implementez operațiuni CRUD (Create, Read, Update, Delete) prin intermediul unui ORM. Am dobândit abilități în manipularea datelor și gestionarea relațiilor între entități.

Un aspect deosebit de important în dezvoltarea aplicației a fost securitatea. Am învățat să implementez autentificarea și autorizarea utilizatorilor utilizând JWT, asigurând astfel că doar utilizatorii autentificați și autorizați au acces la resursele protejate.

Contribuția mea în acest proiect constă în implementarea funcționalităților cheie, în respectarea standardelor de calitate și în asigurarea unei experiențe plăcute pentru utilizatori. Am reușit să integrez tehnologii și cadre de lucru, să realizez o interfață intuitivă și atractivă și să construiesc un backend robust și securizat.

Pe parcursul dezvoltării, am depășit provocări și am învățat din greșeli, construind astfel un fundament solid pentru dezvoltarea ulterioară a aplicației. Această experiență mi-a adus cunoștințe valoroase și m-a pregătit pentru proiecte și provocări viitoare în domeniul dezvoltării software.

În concluzie, acest proiect full stack a reprezentat o oportunitate de creștere personală și profesională, în care am dobândit cunoștințe și abilități în dezvoltarea aplicațiilor web complexe.

Bibliografie

- [1] Kellyn Gorman, Allan Hirt, Dave Noderer, James Rowland-Jones, Arun Sirpal, Dustin Ryan, Buck Woody (2019). *Introducing Microsoft SQL Server 2019*
- [2] Dusan Petkovik (2020). *Microsoft SQL Server 2019: A Beginner's Guide*

i <https://app.clockify.me/tracker>

ii <https://toggl.com/track/>

iii <https://www.telerik.com/blogs/aspnet-core-beginners-web-apis>

iv <https://www.telerik.com/blogs/aspnet-core-beginners-web-apis>

v <https://learnsql.com/blog/history-ms-sql-server/ms-sql-server-history-timeline.png>

vi <https://cynoteck.com/blog-post/typescript-vs-javascript-vs-ecmascript-know-the-difference/>

vii <https://www.netsolutions.com/insights/net-core-vs-net-framework/>

viii <https://cdn-blog.scalablepath.com/uploads/2022/12/spa-va-mpa-key-features-1024x800.png>

ix <https://www.ionos.com/digitalguide/hosting/technical-matters/what-is-http/>

x <https://blog.miniorange.com/what-is-jwt-json-web-token-how-does-jwt-authentication-work/>

xi <https://auth0.com/learn/json-web-tokens>

xii <http://entityframeworkschool.com/EntityFramework/code-first>

xiii https://i.ytimg.com/vi/_fxrldX1GIQ/maxresdefault.jpg

xiv https://external-preview.redd.it/vBjTiQvkQzyz_sAZGdtDRPicDqfm76weTXAg7JLY_d4.jpg?width=640&crop=smart&auto=webp&s=fe2c0aaa34797d4eef7592047112a5567f66bd74