Universitatea
Transilvania
din Brașov
FACULTATEA DE MATEMATICĂ
ȘI INFORMATICĂ

LUCRARE DE LICENȚĂ

Conducător științific:
Lector dr. Nănău Corina-Ștefania

Absolvent:
Hosszú Zsolt

BRAȘOV, 2024

Universitatea
Transilvania
din Brașov
FACULTATEA DE MATEMATICĂ
ȘI INFORMATICĂ

LUCRARE DE LICENȚĂ

InvoiceJet - Aplicație web pentru emiterea și gestionarea facturilor între firme

Absolvent: Hosszú Zsolt
Conducător științific: Lector dr. Nănău Corina-Ștefania

BRAȘOV, 2024

Cuprins

Capitolul 1

Introducere

1.1 Prezentarea lucrării

Emiterea și gestionarea facturilor reprezintă o activitate esențială pentru orice societate comercială din România. Aplicația InvoiceJet vine ca o soluție modernă, destinată pentru a simplifica acest proces esențial. Dezvoltată cu ajutorul tehnologiilor .NET și Angular, InvoiceJet oferă o platformă robustă și intuitivă pentru administrarea eficientă a facturilor, adresându-se atât companiilor mici/medii, cât și profesioniștilor independenți (PFA).

În contextul actual, existența unor aplicații similare precum SmartBill, Oblio, și SAGA demonstrează necesitatea și utilitatea unor astfel de soluții digitale. Totuși, majoritatea acestor aplicații implică costuri de utilizare care pot reprezenta un impediment pentru afacerile la început de drum sau pentru profesioniștii independenți.

InvoiceJet oferă o soluție pentru această problemă prin oferirea unei alternative gratuite, dezvoltată în special din necesitatea proprie de a emite facturi, în calitate de PFA. Scopul principal al acestei aplicații este de a facilita procesul de facturare și de a simplifica gestionarea relațiilor comerciale, fără a impune costuri suplimentare utilizatorilor săi.

Prin integrarea celor mai recente tehnologii, InvoiceJet asigură o experiență de utilizare optimizată, securitate si o interfață prietenoasă. Această lucrare va explora detaliat aspectele tehnice ale aplicației, beneficiile sale, și modul în care poate transforma procesul de facturare într-o activitate simplă și eficientă.

1.2 Motivația alegerii temei

Motivația alegerii temei aplicatiei a venit din necesitatea emiterii facturilor pentru clienții proprii, în calitate de PFA.

Utilizând diferite aplicații web pentru îndeplinirea acestei sarcini, am observat că există restricții financiare impuse de aplicații pentru a utiliza diverse funcționalități sau pentru a emite facturi sau alte documente specifice peste un număr prestabilit de exemplare gratuite/lună.

Prin urmare, s-a ivit dorința de a dezvolta o soluție software proprie, cu ajutorul căreia să pot trece de aceste limitări.

1.3 Aplicații similare

O aplicație comparabilă este Oblio, o platformă digitală dedicată emiterii facturilor și gestionării stocurilor pentru o societate comercială. Această aplicație oferă utilizatorilor săi un instrument eficient pentru organizarea produselor, clienților, conturilor bancare și a altor setări specifice, facilitând ulterior generarea diverselor documente specifice, cum ar fi: facturi, facturi storno, proforme și avize.

Oblio se remarcă prin faptul că, în primul an de utilizare, accesul este gratuit, oferind astfel utilizatorilor o perioadă extinsă de familiarizare și evaluare a funcționalităților platformei. Cu toate acestea, după expirarea perioadei de probă, utilizatorii sunt limitați la emiterea a doar trei documente pe lună, ceea ce poate reprezenta o constrângere semnificativă pentru afacerile în creștere sau pentru cele care necesită emiterea unui număr mai mare de documente.

Un alt aspect important de menționat este că aplicația Oblio nu permite previzualizarea documentelor în format PDF înainte de emiterea acestora. Această limitare poate conduce la rezultate inconsistente, întrucât utilizatorii nu au posibilitatea de a verifica și corecta eventualele erori sau omisiuni înainte ca documentele să fie emise și salvate în baza de date.

Prin urmare, deși Oblio oferă numeroase avantaje prin funcționalitățile sale de organizare și gestionare, limitările menționate, cum ar fi restricțiile privind numărul de

documente emise după perioada de probă și lipsa opțiunii de previzualizare a documentelor PDF, trebuie luate în considerare atunci când se evaluează adecvarea acestei platforme pentru nevoile specifice ale unei societăți comerciale.

1.4 Compozitia lucrarii

Primul capitol oferă o prezentare concisă a contextului și obiectivelor aplicației, pregătind terenul pentru detalierea funcționalităților și a implementării în capitolele următoare.

Capitolul doi se concentrează pe aspectele tehnice ale aplicației, inclusiv descrierea tehnologiilor utilizate în proiect.

Capitolul trei este dedicat implementării, explicând în detaliu specificațiile funcționale ale aplicației.

Capitolul patru, numit ghid de utilizare, detaliază funcționalitățile aplicației și oferă instrucțiuni clare de utilizare.

Capitolul final rezumă lucrarea, evidențiind concluziile și sugerând direcții de extindere viitoare posibile.

Capitolul 2

Implementarea proiectului

2.1. Instrumente de dezvoltare

2.1.1. Visual Studio

Visual Studio, dezvoltat de Microsoft, reprezintă un mediu integrat de dezvoltare (IDE[1]), conceput pentru crearea aplicațiilor pentru diverse platforme, precum Windows, Android, iOS și web. Acesta oferă un set cuprinzător de instrumente și servicii care facilitează programarea în diverse limbaje, precum C#, C++, Python și JavaScript. Visual Studio se remarcă prin caracteristicile sale avansate, cum ar fi suportul pentru depanare, testare unitară, sugestii de refactorizare cod automată și integrarea cu sistemele de control al versiunilor, precum GIT.

De asemenea, include funcționalități pentru dezvoltarea colaborativă, cum ar fi live share, care permite programatorilor să colaboreze în timp real. Datorită versatilității și puterii sale, Visual Studio este un instrument esențial pentru dezvoltatorii de software profesioniști.

2.1.2. Visual Studio Code

Visual Studio Code (VS Code) este un editor de cod sursă creat de Microsoft, renumit pentru simplitatea și adaptabilitatea sa. Acesta suportă o gamă largă de limbaje de programare și oferă funcții avansate precum completarea automată a codului și debugging integrat.

Cea mai notabilă funcționalitate a acestuia o reprezintă gama largă de extensii disponibile prin intermediul ecosistemului oferit de Microsoft.

Prin urmare, fiind un editor de cod ușor, multiplatformă[2] și open-source, nu este de mirat că reprezintă o primă alegere mai ales în rândul dezvoltatorilor de front-end, dar nu numai.

---

[1] Integrated Development Environment

[2] eng. Cross-platform

2.1.3. Sql Server Management Studio

SQL Server Management Studio (SSMS) este un instrument dezvoltat de Microsoft pentru gestionarea, configurarea și administrarea bazelor de date SQL Server. SSMS oferă o interfață grafică intuitivă care permite utilizatorilor să creeze, să modifice și să interogheze baze de date, să gestioneze securitatea și să monitorizeze performanța serverului. Este un instrument esențial pentru administratori de baze de date și dezvoltatori, facilitând eficientizarea și simplificarea activităților de gestionare a bazelor de date.

SSMS este în mod special benefic pentru cei care lucrează în ecosistemul .NET din mai multe motive:

- Integrare strânsă cu .NET: SSMS se integrează perfect cu tehnologiile .NET, facilitând dezvoltarea aplicațiilor care interacționează cu baze de date SQL Server. Aceasta permite dezvoltatorilor să folosească limbaje familiare, cum ar fi C# și .NET Framework, pentru a scrie proceduri stocate, a executa scripturi și a gestiona date.
- Securitate sporită: SSMS vine cu funcționalități robuste de securitate care ajută la protejarea datelor, aspect crucial pentru aplicațiile .NET, mai ales în contextul dezvoltării enterprise și a aplicațiilor sensibile din punct de vedere al datelor.
- Scalabilitate: Oferă suport pentru gestionarea bazelor de date de mari dimensiuni, ceea ce este esențial pentru aplicațiile .NET în medii enterprise.

2.2. Unelte dezvoltare back-end

2.2.1. .NET Framework

.NET Framework este o platformă de dezvoltare software creată de Microsoft, concepută pentru a facilita dezvoltarea și rularea aplicațiilor pe Windows. Acesta oferă un mediu de execuție gestionat, cunoscut sub numele de Common Language Runtime (CLR), care gestionează execuția codului și asigură servicii precum gestionarea memoriei, securitatea și tratarea excepțiilor.

Una dintre caracteristicile cheie ale .NET Framework este biblioteca sa extinsă de clase (FCL[3]), care oferă o gamă largă de funcționalități predefinite pentru manipularea datelor, accesul la baze de date, dezvoltarea interfețelor grafice, conectivitatea de rețea și

[3] Framework Class Library

multe altele. Acest set de biblioteci ajută dezvoltatorii să construiască aplicații robuste și eficiente fără a trebui să scrie cod de la zero pentru funcționalități comune.

.NET Framework suportă mai multe limbaje de programare, inclusiv C#, VB.NET și F#. Această interoperabilitate între limbaje este posibilă datorită infrastructurii comune oferite de CLR, care compilează codul în IL[4], permițând astfel diferitelor limbaje să fie utilizate în cadrul aceleași aplicații.

În concluzie, .NET Framework reprezintă o platformă completă și versatilă pentru dezvoltarea de aplicații pe Windows, oferind un set bogat de instrumente și biblioteci care simplifică procesul de dezvoltare și îmbunătățesc productivitatea dezvoltatorilor.

2.2.2. Swagger

Swagger este un set de instrumente open-source utilizate pentru proiectarea, construirea, documentarea și utilizarea serviciilor web REST[5]ful. Acesta oferă o modalitate standardizată de a descrie structura API[6]-urilor, facilitând atât dezvoltarea, cât și integrarea acestora.

Prin utilizarea Swagger, dezvoltatorii pot crea documentație interactivă pentru API-uri, care permite utilizatorilor să exploreze și să testeze endpoint-urile direct dintr-o interfață web. Documentația generată de Swagger include informații detaliate despre resursele disponibile, metodele HTTP suportate, parametrii necesari și structura răspunsurilor.

Unul dintre principalele avantaje ale Swagger este faptul că poate genera specificații în format OpenAPI, un standard recunoscut pentru descrierea API-urilor RESTful. Aceste specificații pot fi utilizate pentru a genera automat cod client și server în diverse limbaje de programare, reducând semnificativ timpul necesar pentru dezvoltarea și testarea API-urilor.

De asemenea, Swagger este preconfigurat atunci când se creează un proiect ASP.NET Web API în Visual Studio, fiind inclus automat în configurările din Program.cs. La rularea API-ului, documentația Swagger este accesibilă la adresa localhost/swagger.

În concluzie, Swagger este un instrument esențial pentru dezvoltatorii de API-uri RESTful, oferind un set complet de soluții pentru crearea și testarea eficientă a serviciilor web, îmbunătățind în același timp comunicarea și colaborarea între echipe.

---

[4] Intermediate Language

[5] REpresentational State Transfer

[6] Application Programming Interface

2.2.3. Entity Framework

Entity Framework (EF) este un ORM[7] open-source creat de Microsoft, care facilitează interacțiunile dintre aplicațiile .NET și bazele de date relaționale. Utilizând EF, dezvoltatorii pot opera cu date folosind obiecte și clase .NET.

Entity Framework oferă, de asemenea, suport pentru LINQ[8], permițând operații CRUD[9] (creare, citire, actualizare și ștergere) asupra datelor direct din intermediul limbajului C#, ceea ce face codul mai intuitiv și mai ușor de scris, , evitând astfel necesitatea scrierii directe a codului SQL.

EF suportă mai multe abordări de dezvoltare, inclusiv:

- Code First: Permite dezvoltatorilor să definească modelul de date folosind clase .NET, iar baza de date este generată din aceste clase. La fiecare actualizare a bazei de date, se produce un fișier de migrare, care conține codul aferent modificărilor realizate asupa bazei de date. În acest fel se poate ține și o evidență a modificărilor aplicate, și în același timp există și posibilitatea de a restaura baza de date la un stadiu anterior în cazul apariției unei erori.

- Database First: Permite generarea de modele de date dintr-o bază de date existentă, creând automat clasele corespunzătoare cu ajutorul unei comenzi de scaffold, în care sunt specificate credențialele serverului pentru a realiza conexiunea la baza de date, dar și folder-ul din intermediul proiectului în care se dorește generarea claselor aferente.

- Model First: Permite crearea unui model de date vizual, din care se generează atât clasele din C#, cât și baza de date.

Am ales abordarea Code First în utilizarea Entity Framework datorită flexibilității sale în adaptarea rapidă a modelului de date conform cerințelor dinamice ale aplicației.

În concluzie, Entity Framework este o unealtă populară pentru dezvoltarea aplicațiilor .NET, oferind eficiență și flexibilitate în gestionarea bazelor de date relaționale.

---

[7] Object Relational Mapping

[8] Language Integrated Query

[9] Create, Read, Update, Delete

2.3. Librării .NET folosite

2.3.1. AutoMapper

AutoMapper este o bibliotecă open-source pentru .NET care facilitează maparea obiectelor între diferite modele. Aceasta este extrem de utilă în scenariile în care este necesar să se transfere date între obiecte cu structuri similare, dar nu identice, cum ar fi maparea entităților de date la modele de vizualizare sau DTO[10]-uri.

Principalele avantaje ale utilizării AutoMapper includ:

- Reducerea codului boilerplate: AutoMapper elimină necesitatea de a scrie manual cod pentru maparea proprietăților între obiecte, automatizând procesul și reducând astfel volumul de cod repetitiv și predispus la erori.

- Simplificarea mentenanței: Prin centralizarea logicii de mapare, AutoMapper face ca modificările ulterioare în structura obiectelor să fie mai ușor de gestionat, deoarece actualizările trebuie făcute doar în configurarea mapării.

- Configurare flexibilă: AutoMapper oferă multiple opțiuni de configurare pentru a personaliza procesul de mapare, inclusiv mapări detaliate prin intermediul metodelor ajutătoare eficiente, care eficientizează realizarea diverselor verificări și asignări.

- Utilizarea AutoMapper implică definirea unui profil de mapare, unde se specifică regulile de mapare între tipurile de obiecte. Odată definit profilul, maparea efectivă a obiectelor devine simplă și directă, folosind metoda Map.

2.3.2. QuestPDF

QuestPDF este o bibliotecă open-source pentru .NET, specializată în generarea documentelor PDF. Aceasta oferă dezvoltatorilor o modalitate simplă și eficientă de a crea documente PDF complex, utilizând un API intuitiv.

---

[10] Data Transfer Objects

QuestPDF se remarcă prin următoarele caracteristici principale:

- API Fluent: QuestPDF utilizează un API fluent care permite dezvoltatorilor să definească structura și conținutul documentelor PDF într-o manieră clară și ușor de citit. Acest lucru face ca procesul de creare a PDF-urilor să fie natural și intuitiv.

- Layout Dinamic: Biblioteca suportă layout-uri dinamice, adaptându-se automat la conținutul generat. Aceasta asigură că elementele se aranjează corespunzător, indiferent de cantitatea de conținut, eliminând necesitatea ajustărilor manuale.

- Elemente de Design: QuestPDF oferă suport pentru diverse elemente de design, cum ar fi tabele, imagini, grafică vectorială și stiluri de text, permițând dezvoltatorilor să creeze documente PDF vizual atractive și profesionale.

- Performanță și Eficiență: Bibliotecă este optimizată pentru performanță, generând documente PDF rapid și eficient, chiar și atunci când se lucrează cu documente mari și complexe.

2.3.2. Libraria Jwt Bearer

JWT[11] Bearer este o bibliotecă utilizată pentru implementarea autentificării bazate pe token-uri în aplicațiile .NET, și nu numai, fiind un standard recunoscut în acest domeniu.

Aceasta permite crearea, semnarea și validarea tokenurilor JWT, facilitând un mod sigur și eficient de autentificare și autorizare a utilizatorilor pentru accesul la proiectul de API.

Caracteristicile principale ale bibliotecii JWT Bearer includ:

- Autentificare securizată: Tokenurile JWT sunt utilizate pentru a autentifica utilizatorii prin transmiterea unui token în fiecare cerere HTTP[12], eliminând astfel necesitatea de a trimite datele de autentificare de fiecare dată.

[11] JSON Web Token

[12] Hypertext Transfer Protocol Request

- Autorizare flexibilă: JWT permite includerea de informații suplimentare (claims) în token, care pot fi utilizate pentru a controla accesul la resursele aplicației pe baza unor roluri de exemplu.

- Suport pentru diverși algoritmi: Biblioteca suportă mai mulți algoritmi de semnare, inclusiv HMAC, RSA și ECDSA, asigurând flexibilitate în alegerea metodei de securizare a tokenurilor.

- Integrare simplă: JWT Bearer poate fi integrată ușor în aplicațiile ASP.NET Core, oferind mecanisme de configurare și extensie pentru a personaliza comportamentul autentificării, de exemplu validarea integrității și a datei de expirare a token-ului la fiecare cerere HTTP.

2.3.2. Libraria Bcrypt.Net

BCrypt.Net este o bibliotecă pentru .NET care implementează algoritmul de hash BCrypt, fiind utilizată în principal pentru securizarea parolelor. Algoritmul BCrypt este preferat pentru stocarea parolelor datorită caracteristicilor sale de securitate, cum ar fi rezistența la atacuri de forță brută[13] și capacitatea de a include un salt unic pentru fiecare parolă.

Caracteristicile principale ale bibliotecii BCrypt.Net includ:

- Hashing securizat al parolelor: BCrypt utilizează un algoritm de hashing adaptiv care face ca generarea hash-urilor să fie intenționat lentă, îngreunând astfel atacurile de forță brută.

- Salt unic: Fiecare parolă este hash-uită împreună cu un salt unic, ceea ce previne atacurile de tip rainbow table.

- Facilitatea de ajustare a costului: BCrypt permite ajustarea costului de hashing (numărul de runde), oferind flexibilitate în echilibrarea securității și performanței.

[13] eng. Brute Force Attack

2.3. Unelte dezvoltare front-end

2.3.1. Angular

Angular prezintă aspecte valoroase ale dezvoltării pe server și le aplică pentru a optimiza HTML-ul în cadrul browserului. Acest lucru stabilește o bază care facilitează construirea de aplicații complexe și îmbogățite. Aplicațiile Angular sunt concepute în jurul unui model de design bine definit, axat pe dezvoltarea de aplicații care sunt extensibile, ușor de întreținut și de testat, și care respectă standardele industriei. Aceasta abordare face ca Angular să fie o alegere excelentă pentru dezvoltarea de aplicații web avansate și conforme cu cele mai recente tehnologii HTML și cu alte instrumente și framework-uri populare.

Aplicațiile Angular sunt construite în jurul unui model de proiectare clar, care pune accent pe crearea de aplicații care sunt:

- Extensibile: Odată ce înțelegeți elementele de bază, este ușor să descifrați modul în care funcționează chiar și o aplicație Angular complexă—ceea ce înseamnă că puteți îmbunătăți cu ușurință aplicațiile pentru a crea caracteristici noi și utile pentru utilizatorii dvs.

- Mentenabile: Aplicațiile Angular sunt ușor de depanat și reparat, ceea ce înseamnă că întreținerea pe termen lung este simplificată.

- Testabile: Angular are un suport bun pentru testele unitare și cele de tip end-to-end, ceea ce înseamnă că se pot găsi și repara defectele înainte ca utilizatorii să le descopere.

- Standardizate: Angular se bazează pe capacitățile native ale browserului web fără a le constrânge, permițând crearea aplicațiilor web conforme cu standardele și valorificând cele mai recente caracteristici HTML, precum și alte unelte și cadre populare. [1]

2.3.2. Angular CLI

Angular CLI este un instrument de interfață în linie de comandă care permite utilizatorului crearea de șabloane de fișiere specifice aplicației Angular, precum: componente, servicii, interfețe etc.

Angular CLI este disponibil pe npm sub numele de pachet @angular/cli și include un instrument de linie de comandă numit ng. Comenzile care invocă ng utilizează Angular CLI.

Acesta oferă o serie de comenzi utile care ajută la gestionarea ciclului de viață al unei aplicații Angular, începând de la inițializarea până la desfășurarea și întreținerea acesteia. Iată câteva dintre cele mai comune comenzi:

- ng new [nume-proiect]: Creează un nou proiect Angular, configurând toate fișierele de bază necesare.

- ng serve -o: Lansează un server virtual prin localhost și deschide aplicația în browser, cu reactualizare automată a browser-ului la la fiecare modificare a codului sursă.

- ng generate component [nume-componentă]: Generează o nouă componentă Angular în proiectul curent, creând fișierele corespunzătoare pentru template (.HTML[14]), stiluri (.CSS[15]) și logica componentei (.TS[16]).

- ng build: Compilează aplicația în fișiere statice, pregătite pentru a fi desfășurate pe un server web. Aceasta include optimizarea codului pentru producție.

- ng test: Rulează testele unitare definite în aplicație, folosind Karma și Jasmine.

- ng deploy: Ajută la deploy-area aplicației pe platforme specifice, având integrare pentru diferite servicii de hosting.

[14] HyperText Markup Language

[15] Cascading Style Sheets

[16] TypeScript

2.4. Pachete folosite Angular

2.4.1. Angular Material

Angular Material este o bibliotecă de componente web pentru interfața utilizatorilor (UI[17]) pe care dezvoltatorii o pot folosi în proiectele lor Angular pentru a accelera dezvoltarea interfețelor responsive și user-friendly. Angular Material oferă componente UI reutilizabile și atractive, precum Cărți (Cards), Câmpuri de Intrare (Inputs), Tabele de Date (Data Tables), Selector de Dată (Datepickers) și multe altele.

Fiecare componentă este gata de utilizare cu un stil implicit care respectă Specificațiile Designului Material. Cu toate acestea, aspectul componentelor Angular Material poate fi personalizat cu ușurință prin suprascrierea acestora prin metode CSS, sau prin extinderea componentelor cu proprietăți proprii.

Lista componentelor disponibile în Angular Material continuă să se extindă odată cu fiecare versiune nouă a bibliotecii. Aceasta facilitează și îmbunătățește procesul de dezvoltare a aplicațiilor, permițând crearea unor interfețe utilizator estetice și funcționale.

2.4.2. Angular Jwt

Biblioteca @auth0/angular-jwt este un instrument util pentru gestionarea token-urilor JSON Web Token (JWT) în aplicațiile Angular. Aceasta facilitează atât decodarea și verificarea tokenurilor JWT, cât și atașarea automată a acestora la cererile HTTP pentru autentificarea și autorizarea utilizatorilor la comunicarea cu proiectul de back-end.

Caracteristici Principale:

- Decodare Token: Permite aplicațiilor să decodeze tokenurile JWT pentru a extrage informațiile de utilizator sau alte metadate necesare.

- Verificare Automată: Poate fi configurată pentru a verifica automat valabilitatea tokenurilor JWT, inclusiv expirarea și integritatea acestora.

- Interceptarea Cererilor HTTP: Integrează un interceptor HTTP care atașează automat tokenul JWT la anteturile cererilor HTTP care sunt trimise către server. Acest lucru simplifică implementarea autentificării pe bază de token.

[17] User Interface

- Managementul Tokenurilor: Ajută la gestionarea stocării și reînnoirii tokenurilor JWT, asigurând că aplicația folosește tokenuri valide pentru cererile autentificate.

2.4.3. Chart.js si ng2charts

Chart.js este o bibliotecă open source de grafice JavaScript care simplifică crearea graficelor interactive și responsive direct în browser. Aceasta oferă o varietate de tipuri de grafice, cum ar fi graficele liniare, cele de tip bar, radar, doughnut, și pie, printre altele. Biblioteca folosește canvas-ul HTML pentru a desena graficele, ceea ce îi conferă o performanță bună și compatibilitate largă cu browserele moderne.

ng2-charts este un set de directive Angular care învăluie funcționalitățile Chart.js, permițând integrarea ușoară a graficelor Chart.js în aplicații Angular. Această bibliotecă facilitează utilizarea Chart.js prin oferirea de directive care pot fi folosite direct în template-urile Angular, gestionând automat actualizările și interacțiunea dintre Angular și Chart.js.

Capitolul 3

Aspecte ale implementării

3.1 Structura aplicatiei

Aplicația este organizată în două proiecte principale: back-end și front-end. Proiectul de back-end utilizează .NET Core Web API, datorită capacității sale robuste de a gestiona API-uri și servicii web. În ceea ce privește interfața utilizatorului, se utilizează framework-ul Angular pentru a construi un front-end dinamic în manieră SPA[18]. Pentru stocarea și administrarea datelor necesare funcționării aplicației, este folosit SQL Server ca sistem de gestionare a bazelor de date.

Această combinație de tehnologii asigură o arhitectură solidă, facilitând atât manipularea eficientă a datelor, cât și interacțiunea fluidă cu utilizatorul. Această structură permite dezvoltarea unei aplicații eficiente, capabilă să răspundă nevoilor complexe ale utilizatorilor și să proceseze datele cu încredere și siguranță.

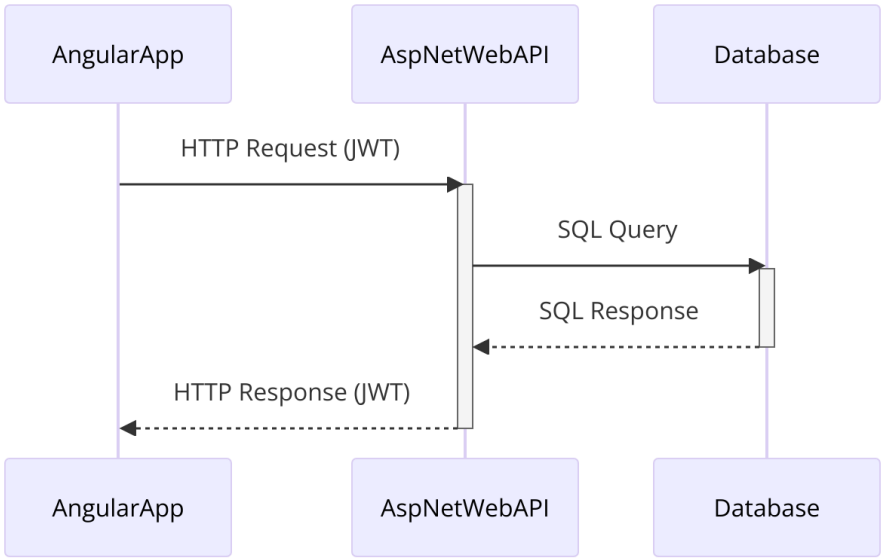


Figura 1.1: Structura Aplicatiei

[18] Single Page Application

3.2 Arhitectura server-side

Arhitectura curată (Clean Architecture) este un concept dezvoltat de Robert C. Martin (cunoscut și sub numele de Uncle Bob), care se concentrează pe organizarea și structurarea codului în aplicațiile software astfel încât să fie ușor de întreținut, extins și testat. Principalele concepte ale arhitecturii curate sunt stratificarea și separarea clară a responsabilităților.

Acesta aduce numeroase beneficii, precum: indepentența față de framework-uri, interfața utilizatorului, baza de date, agenți externi, precum API-urile externe și nu în ultimul rând, testarea unitară. [2]

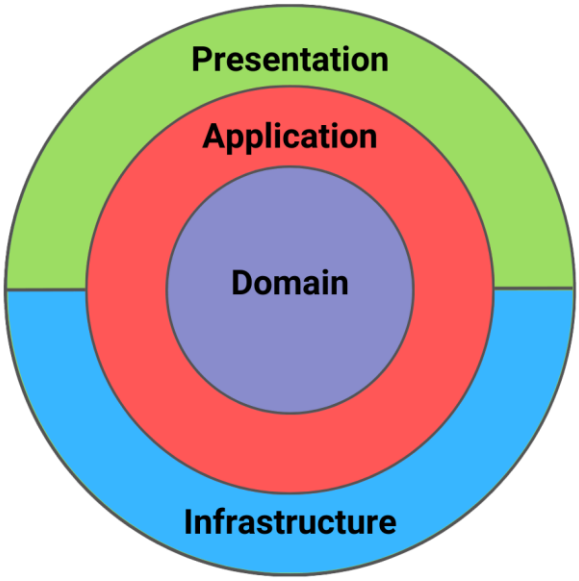


Figura 1.2: Arhitectura curată

3.2.1. Stratul de prezentare

Nivelul de prezentare furnizează funcționalitatea pentru interfața utilizatorului într-o aplicație. Acesta prezintă datele utilizatorului și preia intrări, respectiv trimite date necesare populării interfeței de utilizator.

Într-o aplicație ASP.NET Web API, acest strat este reprezentat de Controller, care are rolul de a trimite datele mai departe la stratul de business logic pentru a fi prelucrate, iar în același timp el este responsabil și pentru trimiterea unui cod de stare (status code) specific interogăriilor REST prin API, care aparține uneia dintre următoarele categorii:

- **1XX: Răspunsuri informative**

  Acestea sunt mesaje preliminare care informează clientul că serverul a primit cererea și procesul este în curs.

- **2XX: Răspunsuri de succes**

  Aceste coduri indică faptul că cererea a fost primită și procesată cu succes de către server.

- **3XX: Mesaje de redirecționare**

  Aceste răspunsuri informează clientul că mai sunt acțiuni suplimentare pentru a finaliza cererea, de obicei sub forma unei redirecționări către o altă adresă URL.

- **4XX: Erori de client**

  Aceste coduri indică faptul că cererea nu a putut fi procesată din cauza unei probleme care ține de client (de exemplu, o cerere formatată greșit sau acces neautorizat).

- **5XX: Erori de server**

  Aceste coduri semnalează că serverul a întâmpinat o eroare internă și nu poate procesa cererea.

Un exemplu de Controller din aplicație putem vedea în figura 1.3.

Nivelul de Controller face parte din spațiul de nume[19] cu același nume “Controllers”, iar clasa din interiorul acesteia este denotată cu atributul [ApiController] pentru a semnala aplicației că următoarele linii vor conține implementarea funcțiilor specifice unui controller.

[19] namespace

Al doilea atribut, [Route(“api/[controller]”)] denotă numele rutei de bază, care în acest caz va fi api/[nume-controller] prin care se pot apela endpoint-urile din interiorul acestuia prin concatenarea acestei rute de bază cu atributele din interiorul atributelor specifice http, cele mai comune fiind HttpGet, HttpPost, HttpPut și HttpDelete.

```
using InvoiceJetAPI.Models.Dto;
using InvoiceJetAPI.Services;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace InvoiceJetAPI.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    [Authorize(Roles = "User")]
    public class DocumentController : ControllerBase
    {
        private readonly IDocumentService _documentService;
        public DocumentController(IDocumentService documentService)
        {
            _documentService = documentService;
        }

        [HttpGet("GetDocumentAutofillInfo/{userId}/{documentTypeId}")]
        public async Task<ActionResult<DocumentAutofillDto>> GetDocumentAutofillInfo(Guid userId, int documentTypeId)
        {
            try
            {
                var documentAutofillDto = await _documentService.GetDocumentAutofillInfo(userId, documentTypeId);
                return Ok(documentAutofillDto);
            }
            catch (Exception ex)
            {
                return BadRequest(ex.Message);
            }
        }
```

Figura 1.3: Exemplu implementare Controller

A treia caracteristică a figurii indică faptul că accesul la Controller este restricționat doar la solicitările venite din partea utilizatorilor care au rolul de „User”.

3.2.2. Stratul de infrastructură

Stratul de infrastructură este unul dintre cele mai importante componente ale Arhitecturii Curate.

Acesta este responsabil pentru implementarea detaliilor tehnice și pentru asigurarea comunicării între straturile interioare ale aplicației și lumea exterioară. Stratul de infrastructură se află la periferia arhitecturii și include diverse framework-uri, drivere și alte componente tehnice esențiale pentru funcționarea aplicației.

Principiile cheie ale stratului includ: dependența de interfețe, separea responsabilităților, reutilizarea codului, testabilitatea.

În cazul proiectului InvoiceJet, acesta conține librăriile externe aferente generării documentelor: QuestPDF, comunicarea și preluarea datelor prin intermediul API-ului oferit de ANAF, și nu în ultimul rând înregistrarea și autentificarea în aplicație folosing librăria JWT Bearer.

De asemenea stratul de infrastructură este responsabil și pentru persistența datelor, aici fiind prezent fișierul de context al bazei de date, care derivă din DbContext și specifică setări adiționale legate de modele și migrări.

În plus, el conține și implementarea stratului de Repository, al cărui rol este de a interacționa eficient direct cu baza de date prin intermediul Entity Framework și a efectua operații CRUD, în timp ce asigură integritatea datelor.

3.2.3. Stratul de aplicație

Stratul de aplicație este un component esențial în cadrul Arhitecturii Curate, având rolul de a coordona fluxul de date între entități, infrastructură și prezentare, de a implementa logica specifică cazurilor de utilizare.

Aceasta se ocupă de implementarea regulilor de afaceri și de procesarea datelor necesare pentru a răspunde nevoilor specifice ale aplicației.

În proiectul prezentat, acesta conține definirea contractelor și implementarea serviciilor ce țin în mod direct de logica aplicației, prin apelarea metodelor și serviciilor externe din interiorul stratul de infrastructură.

Pe lângă asta, mai sunt prezente și mapările aferente acestei logici, pentru utilizarea facilă, după sfârșitul sau la începutul interacționării cu baza de date.

Pentru folosirea mapărilor este nevoie de asemenea și de stratul de Domeniu la care stratul de aplicație are referință, dar și de obiectele de tip DTO.

Din această considerent, acest strat conține ș definirea DTO-urilor, care mai apoi sunt folosite direct prin intermediul serviciilor.

3.2.3. Stratul de domeniu

Stratul de domeniu, cunoscut și sub denumirea de “Domain”, este nucleul sistemului și trebuie să fie cât mai independent posibil de alte straturi și de infrastructura tehnologică.

În cazul proiectului de față, în acest strat sunt definite entitățile, obiecte de tip enum, care sunt folosite la declararea unor câmpuri din intermediul modelelor, contractul

pentru IUnitOfWork, folosind pattern-ul Unit of Work și interfețele de tip Repository isntanțiate în cadrul acestuia.

De asemenea, aici se află și excepțiile personalizare ale aplicației, care sunt aruncate pe parcursul rulării aplicației, în cazul unei erori. [7]

3.3. Middleware

Middleware-ul „ExceptionMiddleware" din codul sursă prezentat în figura 1.4 are rolul de a intercepta și de a gestiona excepțiile care apar în timpul procesării unei cereri într-o aplicație ASP.NET Core.

Este o componentă esențială pentru gestionarea erorilor la nivelul întregii aplicații, asigurându-se că excepțiile sunt tratate într-o manieră consistentă și că răspunsurile adecvate sunt trimise clientului.

```
namespace InvoiceJetAPI.Exceptions.Middleware
{
    public class ExceptionMiddleware
    {
        private readonly RequestDelegate _next;

        public ExceptionMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task Invoke(HttpContext context)
        {
            try
            {
                await _next(context);
            }
            catch (AnafFirmNotFoundException ex)
            {
                context.Response.StatusCode = 404;
                context.Response.ContentType = "application/json";
                await context.Response.WriteAsync($"{ex.Message}");
            }
            catch (Exception ex)
            {
                context.Response.StatusCode = 500;
                context.Response.ContentType = "application/json";
                await context.Response.WriteAsync($"{ex.Message}");
            }
        }
    }
}
```

Figura 1.4: Exemplu Exception Middleware

3.4. Baza de date

Schema prezentată ilustrează structura și relațiile dintre entitățile bazei de date destinate aplicației InvoiceJet.

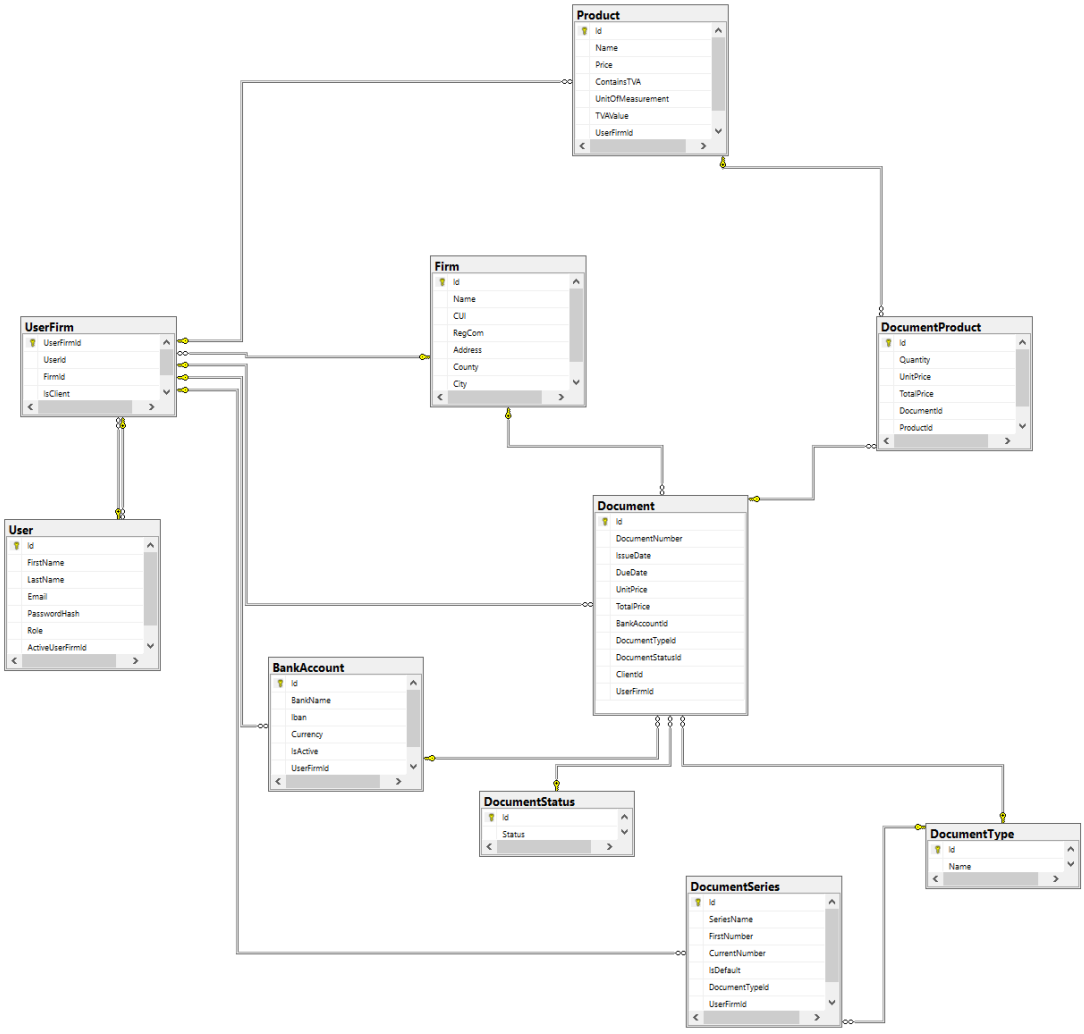


Figura 1.5: Schema relațională a bazei de date

Descriere detaliată a relațiilor dintre entitățile prezente în figura 1.4.

Relațiile User, UserFirm și Firm:

Indică relația dintre utilizatori și firme diferite la care aceștia pot avea acces sau cu care sunt afiliați. Fiecare înregistrare în UserFirm leagă un utilizator de o firmă specifică și stochează informații despre rolul acestuia (exemplu: client sau furnizor).

Relația User și BankAccount:

Un User poate avea multiple conturi bancare, ceea ce permite utilizatorilor să gestioneze mai multe conturi bancare. Aceste conturi sunt folosite pentru tranzacții financiare legate de documente sau facturi emise de utilizatori.

Relațiile Document, DocumentType, DocumentSeries și DocumentStatus:

Un document poate include un tip, o serie și un număr, având de asemenea un status asociat (de ex: neîncasat, încasat, stornat).

Aceasta facilitează categorizarea și organizarea documentelor conform tipului, seriei și stării acestora. În plus, documentele stabilesc relații cu utilizatorii și firmele, pentru a indica cine a emis documentul și cine este beneficiarul.

Relația Product și DocumentProduct:

Produsele reprezintă articolele sau serviciile care pot fi listate în documente. DocumentProduct este o tabelă de joncțiune care leagă documentul de produse, permițând documentelor să includă multiple produse. Această entitate stochează cantități, prețuri unitare și prețuri totale, oferind detalii despre produsele specifice incluse în fiecare document.

Relaţtia BankAccount și Document:

Relația între contul bancar și documente sugerează că fiecare document poate avea asociat un cont bancar specific pentru tranzacții financiare, cum ar fi plata facturilor.

Această structură de baza de date este proiectată să suporte complexitatea gestionării documentelor într-un mediu corporativ, asigurând flexibilitate în administrarea relațiilor financiare și comerciale dintre utilizatori, firme și tranzacțiile lor. Relațiile stabilite între entități facilitează o varietate de funcții ale sistemului, de la facturare și gestionarea documentelor până la urmărirea statusului documentelor și administrarea detaliilor financiare ale utilizatorilor și firmelor.

3.4 Arhitectura client-side

Angular este adeptul arhitecturii MV*, care reprezintă o combinație dintre arhitecturile MVC[20] și MVVM[21].

Noul concept aici este ViewModel-ul, care reprezintă codul de legătură ce conectează view-ul cu modelul sau serviciul. În Angular, această legătură este cunoscută sub numele de binding (legare).

Cea mai de bază unitate a unei aplicații Angular este componenta. O componentă combină o clasă JavaScript, scrisă în TypeScript, și un șablon Angular, scris în HTML, CSS și TypeScript, ca un singur element. Clasa și șablonul se potrivesc împreună ca un puzzle prin intermediul binding-urilor, astfel încât acestea să poată comunica una cu cealaltă, după cum se arată în diagrama următoare: [3]

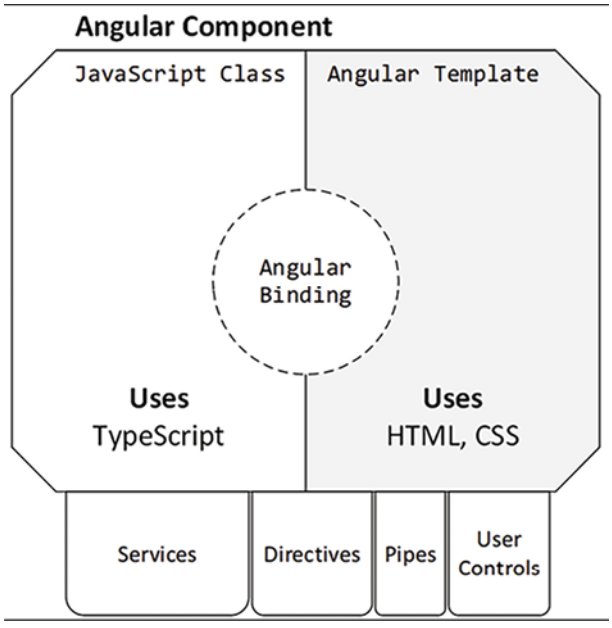


Figura 1.6: Arhitectura MV* în Angular

În același timp în Angular există și o convenție pentru crearea și separarea clară a acestor fișiere prin impunerea unei structuri de foldere, vezi figura 1.7.

[20] Model View Controller

[21] Model Model-View Model

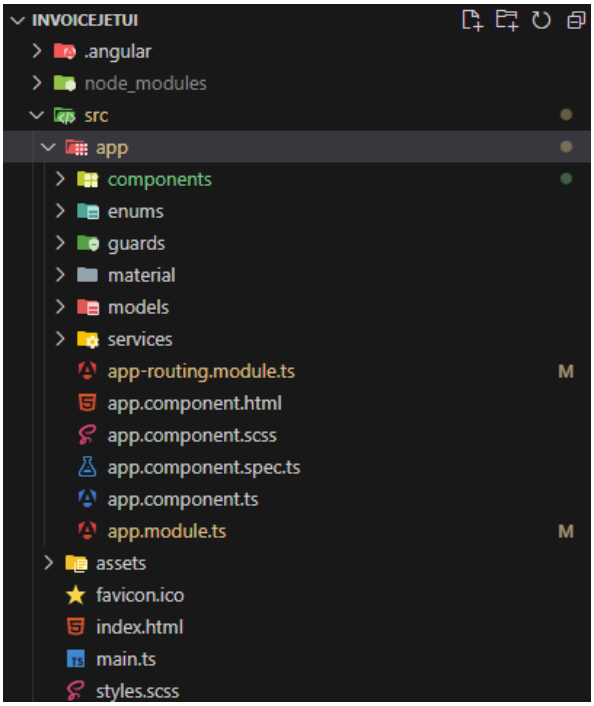


Figura 1.7: Exemplu Structură aplicație Angular

În figura alăturată este prezentată structura de foldere, împărțită pe 3 categorii principale: servicii, modele și componente, care conțin view-urile aferente paginilor din aplicație.

3.5. Specificații funcționale

Scopul principal al aplicației InvoiceJet este de a facilita și eficientiza procesul de emitere și gestionare a facturilor pentru companiile mici și medii, precum și pentru profesioniștii independenți. Această platformă se dorește a fi o soluție accesibilă, eliminând costurile suplimentare asociate cu alte software-uri de facturare.

InvoiceJet urmărește să ofere o interfață intuitivă și o experiență de utilizare îmbunătățită, simplificând astfel gestionarea relațiilor comerciale și transformând facturarea într-un proces simplu, rapid și sigur.

3.5.1. Actori

Diagrama UML de actori atașată prezintă procesul și acțiunile realizabile prin intermediul aplicației într-o manieră simplificată și ușor de înțeles.

Utilizatorul este considerat un actor, care poate efectua anumite operații succesive în cadrul aplicației.

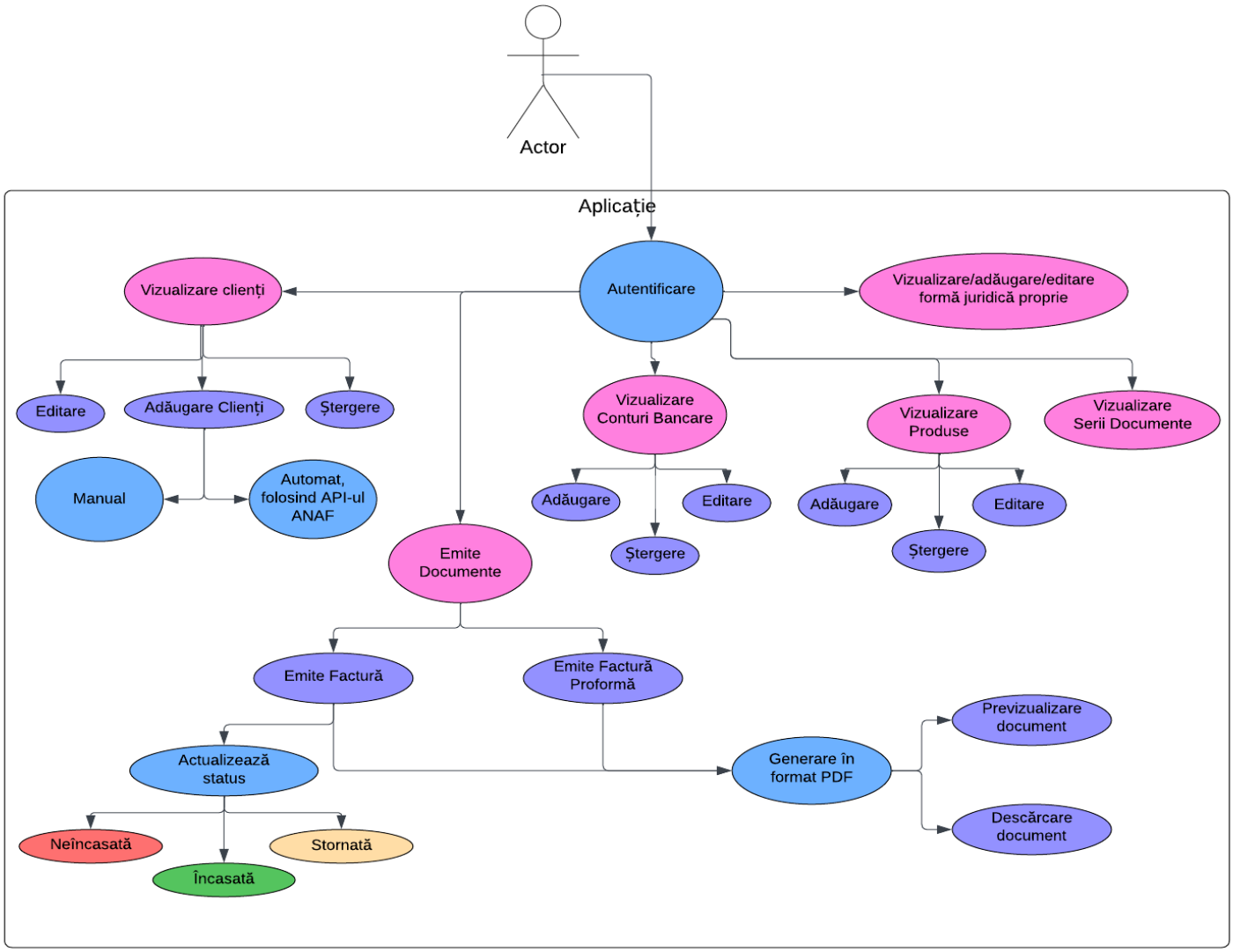


Figura 1.8: Diagramă UML de tip actor

3.5.2. Implementare back-end

Definirea modelelor și aplicarea migrărilor:

Entitățile aplicației se regăsesc în folderul "Models/Entity", care sunt mapate la tabelele bazei de date SQL Server.

Cu ajutorul interfeței din linie de comandă oferite de NuGet Package Manager Console, și folosind (Object Relational Mapping) ORM-ului Entity Framework se pot aplica următoarele comenzi folositoare pentru sincronizarea bazei de date:

- **Adăugarea unei migrări:**

  Comanda: "Add-Migration NumeMigrare"

Acesta creează o nouă migrație pentru a înregistra schimbările făcute în modelele de date ale aplicației.

- **Aplicarea migrărilor la baza de date:**

  Comanda: „Update-Database”

Această comandă aplică migrațiile pendinte la baza de date, actualizând schema bazei de date conform ultimelor modificări aduse modelelor.

- **Revenirea la o migrare anterioară:**

  Comanda: „Update-Database -Migration NumeMigrare”

  Cu această comandă se poate reveni la o anumită migrație, dacă este necesar să anulezi modificările recente.

- **Listarea migrărilor:**

  Comanda: „Get-Migrations”

  Această comandă listează toate migrațiile care au fost aplicate până în prezent.

De asemenea, este important de menționat și interfața de comandă dotnet ef, care este o alternativă la utilizarea NuGet Package Manager Console și este adesea preferată în medii de dezvoltare cross-platform sau atunci când se lucrează cu .NET Core sau .NET 5/6/7+. Comenzile sunt definite într-o manieră similară:

- **Adăugarea unei migrări:**

  Comanda: “dotnet ef migrations add NumeMigrare”

- **Aplicarea migrărilor la baza de date:**

  Comanda: „dotnet ef database update”

- **Revenirea la o migrare anterioară:**

Comanda: „dotnet ef database update NumeMigrare”

- Listarea migrărilor:

  Comanda: „dotnet ef migrations list”

Definirea Relațiilor:

Entity Framework suportă relații complexe între entități, incluzând one-to-one, one-to-many și many-to-many. Relații exemplificate pe baza figurii 1.9:

- One-to-one: Fiecare entitate UserFirm poate avea exact o enitate User și exact o entitate Firm asociată, aceasta fiind o tabelă de legătură între User și Firm.

- One-to-many: Entitatea UserFirm poate avea multiple conturi bancare, produse, serii de documente și documente asociate.

- Many-to-many: Entitatea UserFirm reprezintă în sine, alături de entitățile User și Firm o relație many-to-many, pentru că un utilizator poate avea mai multe firme, fie ele proprii sau a clienților lui, dar în același timp multiple firme pot face referire la același utilizator.

Navigare și Lazy Loading:

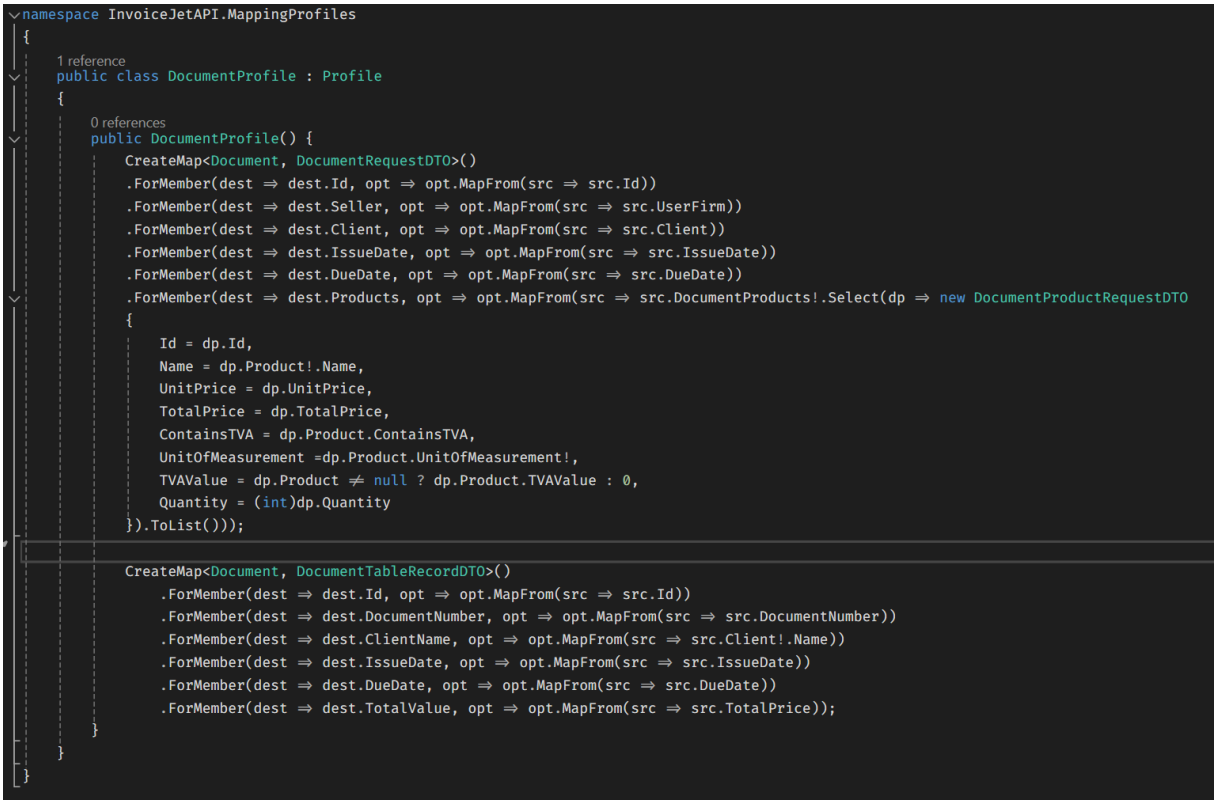
Proprietățile de tip virtual permit Entity Framework să utilizeze „lazy loading” pentru a încărca automat datele asociate când acestea sunt accesate pentru prima dată. De exemplu, BankAccounts, Products, DocumentSeries, Documents sunt încărcate doar când sunt necesare și implicit specificate în interogare.

Figura 1.9: Exemplu Entitate și Relații

Configurarea mapărilor între modele:

Utilizarea bibliotecii AutoMapper, disponibilă prin NuGet Package Manager, facilitează maparea eficientă între clasele DTO (Data Transfer Object) și entitățile de bază. Acest proces nu doar că eficientizează logica aplicației prin transmiterea și primirea datelor strict necesare, dar contribuie și la securitatea acesteia, prin evitarea expunerii directe a entităților în interfața cu utilizatorul.

Mai mult, implementarea claselor DTO sprijină scalabilitatea aplicației. Introducerea de noi proprietăți în entități poate fi gestionată fără a influența direct logica existentă, datorită acestui strat intermediar. Astfel, modificările structurale pot fi izolate de restul aplicației, menținând coerența și reducând riscul de erori.



```
namespace InvoiceJetAPI.MappingProfiles
{
    1 reference
    public class DocumentProfile : Profile
    {
        0 references
        public DocumentProfile() {
            CreateMap<Document, DocumentRequestDTO>()
            .ForMember(dest ⇒ dest.Id, opt ⇒ opt.MapFrom(src ⇒ src.Id))
            .ForMember(dest ⇒ dest.Seller, opt ⇒ opt.MapFrom(src ⇒ src.UserFirm))
            .ForMember(dest ⇒ dest.Client, opt ⇒ opt.MapFrom(src ⇒ src.Client))
            .ForMember(dest ⇒ dest.IssueDate, opt ⇒ opt.MapFrom(src ⇒ src.IssueDate))
            .ForMember(dest ⇒ dest.DueDate, opt ⇒ opt.MapFrom(src ⇒ src.DueDate))
            .ForMember(dest ⇒ dest.Products, opt ⇒ opt.MapFrom(src ⇒ src.DocumentProducts!.Select(dp ⇒ new DocumentProductRequestDTO
            {
                Id = dp.Id,
                Name = dp.Product!.Name,
                UnitPrice = dp.UnitPrice,
                TotalPrice = dp.TotalPrice,
                ContainsTVA = dp.Product.ContainsTVA,
                UnitOfMeasurement =dp.Product.UnitOfMeasurement!,
                TVAValue = dp.Product ≠ null ? dp.Product.TVAValue : 0,
                Quantity = (int)dp.Quantity
            }).ToList()));

            CreateMap<Document, DocumentTableRecordDTO>()
                .ForMember(dest ⇒ dest.Id, opt ⇒ opt.MapFrom(src ⇒ src.Id))
                .ForMember(dest ⇒ dest.DocumentNumber, opt ⇒ opt.MapFrom(src ⇒ src.DocumentNumber))
                .ForMember(dest ⇒ dest.ClientName, opt ⇒ opt.MapFrom(src ⇒ src.Client!.Name))
                .ForMember(dest ⇒ dest.IssueDate, opt ⇒ opt.MapFrom(src ⇒ src.IssueDate))
                .ForMember(dest ⇒ dest.DueDate, opt ⇒ opt.MapFrom(src ⇒ src.DueDate))
                .ForMember(dest ⇒ dest.TotalValue, opt ⇒ opt.MapFrom(src ⇒ src.TotalPrice));
        }
    }
}
```

Figura 2.1: Exemplu de mapare folosind AutoMapper

Deși maparea între două clase este adesea simplă, implicând transferul de entități similare, există situații când este necesară aplicarea unor logici suplimentare pentru a efectua astfel de mapări. AutoMapper facilitează aceste transformări, oferind apeluri eficiente și facilitând separarea responsabilităților în cadrul aplicației prin utilizarea claselor de tip Mapping Profile.

Aceste clase moștenesc din obiectul Profile și includ metode ajutătoare, precum funcția ForMember. Această funcție permite specificarea, în primul parametru, a clasei sursă de la care se dorește maparea, și în al doilea parametru, a clasei destinație. De asemenea, ForMember permite definirea logicii specifice de mapare pentru fiecare câmp în parte, personalizând astfel procesul de transformare a datelor.

Dependency Injection:

Injecția de dependențe[22] este un model de proiectare utilizat în dezvoltarea software pentru gestionarea dependențelor între clase. Acesta conduce la un cod mai ușor de întreținut, testabil și modular.

[22] Eng. Dependency Injection

Beneficiile Injecției de Dependințe:

1. **Decuplare:**

   DI ajută la decuplarea claselor de dependențele lor. Prin injectarea dependențelor, clasele nu mai instanțiază direct dependențele. Această decuplare face sistemul mai ușor de gestionat și actualizat.

2. **Flexibilitate Îmbunătățită:**

   Schimbările asupra dependențelor necesită modificări minime în clasele care le utilizează. De exemplu, dacă trebuie să treci la un alt UserService, poți face acest lucru fără a modifica consumatorii acelui serviciu, atâta timp cât interfața rămâne aceeași.

3. **Testare îmbunătățită:**

   Prin injectarea dependențelor, devine simplu să le înlocuiești cu mock-uri sau implementări false în timpul testării. Aceasta permite testarea unitară mai riguroasă și izolarea testelor de restul sistemului.

4. **Gestionarea Ciclului de Viață:**

   DI controlează ciclul de viață al dependențelor, ceea ce este crucial în aplicații complexe unde gestionarea resurselor și a memoriei este esențială. Configurarea timpurilor de viață ale serviciilor (cum ar fi singleton, scoped, transient) ajută la optimizarea utilizării resurselor. [5]

Deși există mai multe tipuri de injecție de dependințe, precum: Injecția de Constructor (Constructor Injection), Injecția de Setare (Setter Injection), Injecția de Interfață (Interface Injection), Injecția de Câmp (Field Injection), Injecția de Metodă (Method Injection), forma cea mai comună de injecție folosită este Injecția prin Constructor.

Este preferată pentru că garantează că toate dependențele sunt inițializate înainte de utilizarea obiectului. În același timp prezintă un avantaj și claritatea dependențelor, pentru că ele sunt prezente și vizibile în semnătura constructorului.

```
namespace InvoiceJet.Application.Services.Impl;


public class ProductService : IProductService
{
    private readonly IUnitOfWork _unitOfWork;
    private readonly IMapper _mapper;
    private readonly IUserService _userService;


    public ProductService(IMapper mapper, IUnitOfWork unitOfWork, IUserService userService)
    {
        _mapper = mapper;
        _unitOfWork = unitOfWork;
        _userService = userService;
    }
```

Figura 2.2: Exemplu dependency injection

Unit Of Work:

Interfața IUnitOfWork (Unitate de Lucru) este un concept esențial în arhitectura software, în special în contextul design-ului orientat pe domeniu (DDD - Domain-Driven Design) și în aplicațiile care utilizează un model bazat pe Repository Pattern. Aceasta este utilizată pentru a gestiona tranzacțiile și pentru a asigura consistența datelor în cadrul unei aplicații software.

IUnitOfWork reprezintă un contract (o interfață) care definește operațiunile necesare pentru a gestiona tranzacțiile asupra unui set de obiecte din cadrul unui context specific (de exemplu, o sesiune de lucru cu baza de date). Scopul principal al acestei interfețe este de a coordona salvarea modificărilor efectuate asupra obiectelor și de a gestiona tranzacțiile în mod coerent.

Un avantaj principal constă în instanțierea singulară a contextului bazei de date, prevenind astfel crearea multiplelor instanțe și accesarea multiplă în același timp, ceea ce ar putea cauza erori. Acest lucru se realizează prin injectarea contextului prin constructorul clasei UnitOfWork.

Aceasta, la rândul ei, instanțiază în interiorul său toate clasele de tip Repository, care sunt utilizate în interiorul serviciilor. În același timp, UnitOfWork conține două metode esențiale: CompleteAsync, care salvează toate modificările făcute în cadrul contextului de lucru curent într-un mod asincron, și metoda Dispose, care eliberează resursele ocupate de contextul _context, asigurându-se că orice conexiune la baza de date este închisă corespunzător.

```
namespace InvoiceJet.Infrastructure.Persistence;

1 reference
public class UnitOfWork(InvoiceJetDbContext context) : IUnitOfWork
{
    private readonly InvoiceJetDbContext _context = context;

    10 references
    public IBankAccountRepository BankAccounts { get; } = new BankAccountRepository(context);
    6 references
    public IDocumentProductRepository DocumentProducts { get; } = new DocumentProductRepository(context);
    17 references
    public IDocumentRepository Documents { get; } = new DocumentRepository(context);
    6 references
    public IDocumentSeriesRepository DocumentSeries { get; } = new DocumentSeriesRepository(context);
    2 references
    public IDocumentStatusRepository DocumentStatuses { get; } = new DocumentStatusRepository(context);
    4 references
    public IDocumentTypeRepository DocumentTypes { get; } = new DocumentTypeRepository(context);
    7 references
    public IFirmRepository Firms { get; } = new FirmRepository(context);
    11 references
    public IProductRepository Products { get; } = new ProductRepository(context);
    4 references
    public IUserFirmRepository UserFirms { get; } = new UserFirmRepository(context);
    15 references
    public IUserRepository Users { get; } = new UserRepository(context);

    18 references
    public async Task<int> CompleteAsync()
    {
        return await _context.SaveChangesAsync();
    }

    0 references
    public void Dispose()
    {
        _context.Dispose();
    }
}
```

Figura 2.3: Clasa UnitOfWork

Implementarea acestei clase se află în proiectul de Infrastructură, sub folderul Persistence, al cărui rol principal este de a declara modele și de a persista datele către baza de date.

Un alt aspect important este menționarea declarării proprietăților repository-urilor doar cu get, acestea având acces doar pentru citire din exterior, dar nu și pentru modificare.

Un alt avantaj al utilizării acestui șablon de proiectare îl reprezintă și ușurința de injectare în alte clase, unde dorim să efectuăm operații pe baza de date, scutindu-ne de injectarea a 10 repository-uri pe rând, comparativ cu injectarea unei singure interfețe, și anume IUnitOfWork, după cum se poate vedea și în Figura 2.2.

Repository pattern:

Repository Pattern este un șablon de proiectare folosit pentru a separa logica de acces la date de logica de business din aplicațiile noastre. Acest pattern creează o abstractizare a accesului la date, ceea ce face ca interacțiunea cu baza de date să fie mai ușoară și mai organizată.

Principalele Beneficii

- **Izolarea logicii de acces la date**: Repository Pattern permite separarea clară a logicii de acces la date de restul aplicației, facilitând astfel întreținerea și evoluția codului.

- **Testabilitate**: Deoarece repository-urile sunt interfețe, este ușor să le înlocuim cu mock-uri în timpul testării, ceea ce simplifică scrierea testelor unitare.

- **Reutilizare**: Repository-urile pot fi reutilizate în diverse părți ale aplicației fără a duce la duplicarea codului.

Interfața IGenericRepository reprezintă contractul tuturor repository-urilor aplicației, repository-uri care moștenesc și implementează la rândul lor metodele de bază următoare:

- AddAsync() – responsabilă pentru adăugarea unei noi înregistrări în baza de date în entitatea specificată în template-ul T.

- AddRangeAsync() - responsabilă pentru adăugarea multiplelor înregistrări în baza de date în entitatea specificată în template-ul T.

- UpdateAsync() - responsabilă pentru actualizarea înregistrărilor în baza de date în entitatea specificată în template-ul T.

- RemoveAsync() - responsabilă pentru eliminarea înregistrărilor în baza de date în entitatea specificată în template-ul T.

- RemoveRangeAsync() - responsabilă pentru eliminarea multiplelor înregistrări în baza de date în entitatea specificată în template-ul T.

- Query() – face public accesibilă interfața IQueryable, în cazul în care este necesară accesarea în mod public a repository-ului. Aceasta poate fi folosită pentru join-uri mai complexe, fără restricțiile impuse de pattern-ul repository. Accesarea în această manieră este evitată deoarece încalcă șablonul Repository.

În același timp, această interfață se poate extinde prin intermediul altor interfețe proprii și specifice, cu metode adiționale, care au sens doar în contextul accesului la anumite tabele din baza de date.

```
namespace InvoiceJet.Domain.Interfaces.Repositories;

12 references
public interface IGenericRepository<T> where T : class
{
    10 references
    Task<T?> GetByIdAsync(int id);
    1 reference
    Task<IEnumerable<T>> GetAllAsync();
    9 references
    Task AddAsync(T entity);
    2 references
    Task AddRangeAsync(IEnumerable<T> entities);
    6 references
    Task UpdateAsync(T entity);
    4 references
    Task RemoveAsync(T entity);
    4 references
    Task RemoveRangeAsync(IEnumerable<T> entities);
    19 references
    IQueryable<T> Query();
}
```

Figura 2.4: Interfața IGenericRepository

Este important de menționat că se folosește același context instanțiat prin intermediul clasei UnitOfWork, clasele repository fiind declarate de acolo.

În același timp, un alt aspect important este declararea și folosirea DbSet-urilor la scrierea operațiilor în interiorul repository-urilor, acest lucru aducând constrângeri directe asupra tabelelor la care se pot executa instrucțiuni. În acest fel, accesul și lucrul cu repository-urile devin mai riguroase și mai clare.

Factory method pattern:

Un alt sablon de proiectare notabil folosit in dezvoltarea aplicatiei este si design pattern-ul factory method.

În aplicația InvoiceJet, Factory Method Pattern este utilizat pentru a crea diferite tipuri de documente. Acest pattern este implementat printr-o serie de interfețe și clase concrete care delegă responsabilitatea de creare a documentelor către subclase. Astfel, se obține un cod mai flexibil și extensibil.

Interfața „IDocumentFactory"

```
namespace InvoiceJet.Infrastructure.Factories
{
    public interface IDocumentFactory
    {
        IDocument CreateDocument(DocumentRequestDto
invoiceData);
    }
}
```

Interfața IDocumentFactory definește contractul pentru crearea documentelor. Aceasta include o singură metodă CreateDocument, care primește un obiect de tip DocumentRequestDto și returnează un obiect de tip IDocument.

Funcția GetDocumentFactory din cadrul clasei DocumentFactoryProvider are rolul de a returna clasa factory corectă în funcție de ID-ul identificator al tipului de document furnizat. (figura 2.5)

Această metodă este folosită ulterior pentru a salva instanța documentului dorit în interfața IDocument, care poate fi una dintre cele trei tipuri:

- InvoiceDocument (Factură)
- ProformaInvoice (Factură Proformă)
- StornoInvoice (Factură Storno)

În final, se folosește metoda comună din IDocument pentru a genera documentul în format PDF.

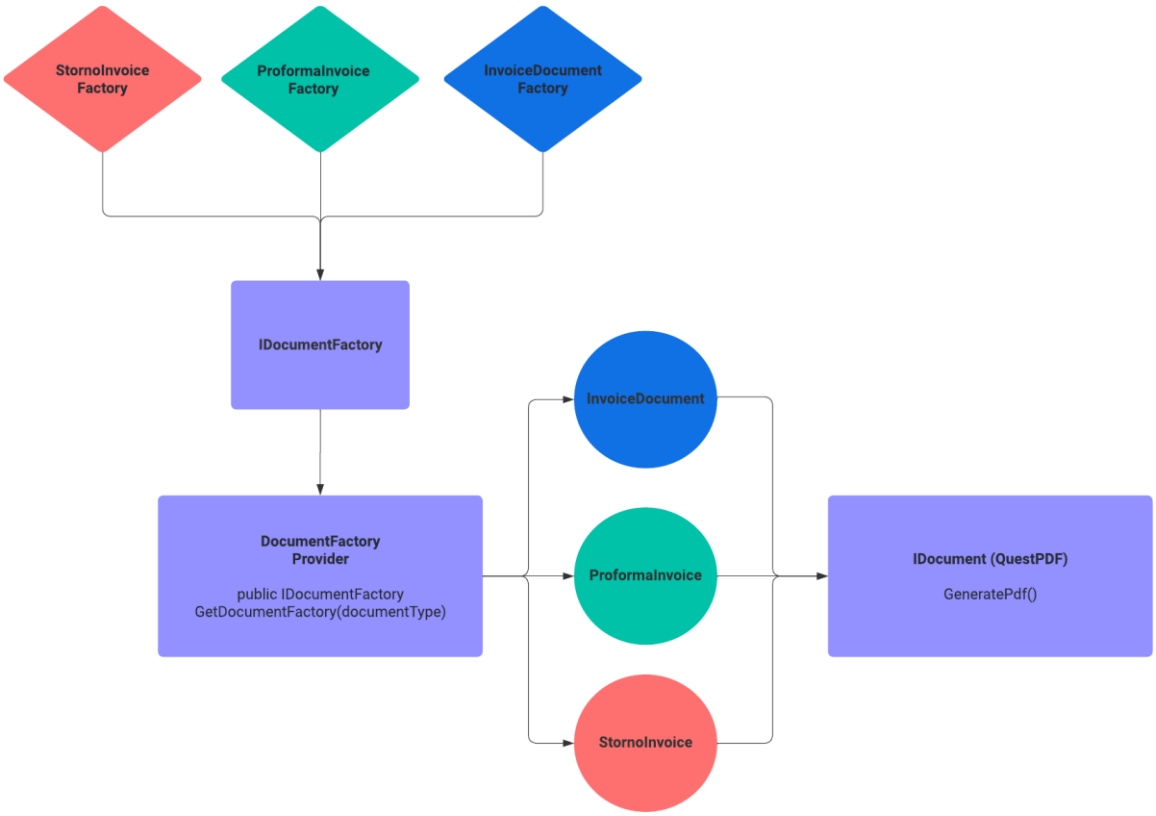


Figura 2.5: Exemplu factory pattern

Factory Pattern poate aduce numeroase beneficii, precum:

- **Separarea codului de instanțiere de cel de utilizare:** Crearea obiectelor este separată de logica principală a aplicației, facilitând întreținerea și înțelegerea codului.

- **Flexibilitate și extensibilitate:** Adăugarea de noi tipuri de obiecte devine simplă, fără a modifica codul existent, ci doar adăugând noi clase factory.

- **Encapsularea logicii de creare:** Toată logica de creare a obiectelor este centralizată într-o singură clasă, ceea ce facilitează modificările și întreținerea.

3.5.3. Implementare front-end

Module

În Angular, App Module (sau modulul principal al aplicației) este elementul central care definește structura și configurarea principală a aplicației. Este implementat ca o clasă decorată cu @NgModule, care organizează și grupează modulele, componentele, directivele și serviciile necesare pentru funcționarea aplicației.

În aplicația InvoiceJet, sunt utilizate mai multe module.

În modulul implicit, App Module, sunt declarate toate componentele aplicației (declarations), toate modulele externe (imports), clasele de tip interceptor, librăria JwtHelperService și setările globale referitoare la Angular Material.

```
@NgModule({
  declarations: [...
  ],
  imports: [...
  ],
  providers: [
    provideAnimations(),
    { provide: JWT_OPTIONS, useValue: JWT_OPTIONS },
    {
      provide: MAT_FORM_FIELD_DEFAULT_OPTIONS,
      useValue: { appearance: "outline" },
    },
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AuthInterceptor,
      multi: true,
    },
    {
      provide: HTTP_INTERCEPTORS,
      useClass: ErrorInterceptor,
      multi: true,
    },
    JwtHelperService,
  ],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Figura 2.6: Exemplu App Module în Angular

Pe lângă acesta, există și modulul material.module.ts, unde sunt declarate doar componentele provenite din librăria Angular Material, pentru o mai bună organizare și diferențiere a componentelor.

Rutare

Un alt modul este si fisierul de rutare: App Routing Module, care este un modul specializat utilizat pentru a configurarea și gestionarea navigării în cadrul unei aplicații. Acesta definește rutele pentru diferitele componente ale aplicației, permițând utilizatorilor să navigheze între pagini fără a reîncărca întreaga aplicație (SPA).

Structura App Routing Module:

- **Importuri**: Se importă toate modulele și componentele necesare pentru configurarea rutelor.

- **Definirea rutelor**: Se creează o constantă care definește toate rutele aplicației, specificând calea și componenta asociată fiecărei rute.

- **NgModule**: Se configurează decoratoarele @NgModule pentru a importa RouterModule și pentru a configura rutele.

După cum se poate observa și în figura 2.7, o rută are mai multe proprietăți, printre cele mai importante se numără:

- **path**: Specifică calea URL-ului pentru rută.

- **component**: Componenta care va fi încărcată atunci când calea respectivă este accesată.

- **canActivate**: Un array de gărzi (guards) care controlează accesul la ruta specificată în funcție de regulile definite în fiecare guard în parte

Rutele care nu au guard-uri implicit nu au accesul restricționat; prin urmare, acestea sunt accesibile public de către orice utilizator prin accesarea rutei adecvate (de exemplu, rutele login și register din figura 2.7).

Ruta "**" capturează orice URL care nu se potrivește cu niciuna dintre rutele definite în modulul App Routing și redirecționează utilizatorul către DashboardComponent, care este protejat de AuthGuard.

```
const routes: Routes = [
  { path: "login", component: LoginComponent },
  { path: "register", component: RegisterComponent },
  {
    path: "dashboard",
    component: DashboardComponent,
    canActivate: [AuthGuard],
  },
  //alte rute
  { path: "**", component: DashboardComponent, canActivate: [AuthGuard] },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Figura 2.7: Exemplu App Routing Module în Angular

Guards

În Angular, guard-urile (gărzi) sunt utilizate pentru a proteja rutele și pentru a controla accesul la diferite secțiuni ale aplicației. Ele sunt implementate ca servicii care decid dacă o anumită rută poate fi activată sau dezactivată, în funcție de condițiile specificate.

Angular oferă mai multe tipuri de guard-uri pentru gestionarea accesului și navigării, precum CaActivate, CanActivateChild sau CanDeactivate.
În cazul de față este folosit CanActivate, fiind nevoie numai de restricționarea rutei pentru accesul de către persoane neautentificate. Acest lucru este verificat în metoda CheckAuthentication din interiorul clasei AuthGuard, care, la rândul său, folosește AuthService pentru a verifica identitatea utilizatorului și pentru a-l deloga, redirecționându-l în cazul neautentificării sau al expirării tokenului JWT.

```
@Injectable({
  providedIn: "root",
})
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ):
    | boolean
    | UrlTree
    | Observable<boolean | UrlTree>
    | Promise<boolean | UrlTree> {
    return this.checkAuthentication(state.url);
  }

  private checkAuthentication(url: string): boolean | UrlTree {
    if (this.authService.isLoggedIn()) {
      return true;
    } else {
      this.authService.logout();
      this.router.navigate(["/login"]);
      return false;
    }
  }
}
```

Figura 2.8: Exemplu AuthGuard

Interceptor

În acest cod, `AuthInterceptor` este un interceptor HTTP utilizat pentru a intercepta și modifica cererile HTTP trimise din aplicația Angular. Rolul său principal este de a adăuga un token de autentificare la antetul cererilor și de a gestiona răspunsurile de eroare.

Funcționalitatea Interceptorului

- **Autentificare**: Interceptorul verifică dacă există un token de autentificare în localStorage. Dacă există, adaugă acest token la antetul cererii HTTP, permițând serverului să valideze cererea.

- **Gestionarea Erorilor**: Dacă serverul răspunde cu un status 401 Unauthorized[23], interceptorul redirecționează utilizatorul către pagina de login și apelează metoda logout pentru a deloga utilizatorul.

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private router: Router, private authService: AuthService) {}

  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    const token = localStorage.getItem("authToken");
    if (token) {
      req = req.clone({
        setHeaders: {
          Authorization: `Bearer ${token}`,
        },
      });
    }

    return next.handle(req).pipe(
      catchError((error: HttpErrorResponse) => {
        if (error.status === 401) {
          this.router.navigate(["/login"]);
          this.authService.logout();
        }
        return throwError(() => new Error(error.message));
      })
    );
  }
}
```

Figura 2.9: Exemplu AuthInterceptor

[23] Neautorizat

Componente

În Angular, componentele reprezintă elementele fundamentale ale aplicației. Fiecare componentă este un bloc de construcție care definește o parte a interfeței utilizatorului și logica asociată acesteia. Componentele sunt reutilizabile și ajută la organizarea aplicației într-o manieră modulară, ceea ce face codul mai ușor de gestionat și întreținut.

```
@Component({
  selector: "app-add-or-edit-invoice",
  templateUrl: "./add-or-edit-invoice.component.html",
  styleUrls: ["./add-or-edit-invoice.component.scss"],
})
export class AddOrEditInvoiceComponent {
```

Figura 2.10: Exemplu Componentă

Elementele Componentei:

- **Decoratorul @Component:**
  @Component este un decorator Angular care marchează o clasă ca fiind o componentă Angular și furnizează metadata pentru configurarea acesteia.

- **Selector:**
  "app-add-or-edit-invoice" definește selectorul CSS care este folosit pentru a instanția această componentă în template-urile HTML. În acest caz, componenta poate fi utilizată în alte template-uri folosind <app-add-or-edit-invoice></app-add-or-edit-invoice>.

- **TemplateUrl:**
  "./add-or-edit-invoice.component.html" specifică locația fișierului HTML care conține template-ul pentru această componentă. Acest template definește structura vizuală a componentei.

- **StyleUrls:**
  styleUrls: ["./add-or-edit-invoice.component.scss"] specifică locația fișierului (sau fișierelor) CSS/SCSS care conține stilurile pentru această

componentă. Aceste stiluri sunt aplicate numai acestei componente, permițând încapsularea stilurilor și evitarea conflictelor.

- Clasa Componentei
  export class AddOrEditInvoiceComponent {} definește clasa TypeScript a componentei. Această clasă conține logica și datele componentei.

Tipurile de binding

În intermediul aplicației sunt folosite mai multe tipuri de binding pentru realizarea comunicării dintre componente și template-uri. Câteva exemple din binding-urile utilizate sunt următoarele:

- Interpolation (Interpolare)
  Folosită pentru a lega datele din componentă în template. Exemplu:

```
<span>{{ documentNumber }}</span>
```

- Property Binding (Legare de Proprietăți)
  Permite legarea proprietăților elementelor DOM la datele din componentă.

```
<input matInput [formControlName]="'client'"
[matAutocomplete]="auto">
```

- Event Binding (Legare de Evenimente)
  Permite gestionarea evenimentelor DOM în componentă.

```
<button mat-raised-button (click)="addProduct()">Add
Product</button>
```

- Two-Way Binding (Legare Bidirecțională)
  Folosită pentru a sincroniza datele între componentă și template în ambele direcții.

```
<input matInput formControlName="name">
```

ReactiveFormsModule

În aplicația InvoiceJet trimiterea formularelor, care conțin date esențiale aplicației și a căror date se doresc a fi salvate, sunt trimise prin intermediul formularului oferit de modulul ReactiveFormsModule din Angular.

Definirea unui FormGroup:

```
invoiceForm: FormGroup = this.fb.group({
  id: 0,
  client: [null, Validators.required],
  issueDate: [new Date(), Validators.required],
  dueDate: null,
  documentSeries: [null],
  documentStatus: [null],
  products: this.fb.array([this.createProductGroup()]),
});
```

Figura 3.1: Exemplu Reactive Forms Module

Definirea în template a formularului:

```
<form [formGroup]="invoiceForm" (ngSubmit)="onSubmit()">
  <input matInput formControlName="client">
  <!– celelalte câmpuri din formular –->
</form>
```

Figura 3.2: Exemplu Reactive Forms Module Template

Formularul de factură (invoiceForm) este utilizat pentru a colecta și valida datele necesare pentru crearea sau editarea unei facturi. Acesta include informații despre client, datele facturii, seria documentului, statusul documentului și produsele incluse în factură. Utilizarea FormGroup, FormControl și FormArray asigură că toate datele sunt gestionate într-un mod structurat și că validările necesare sunt aplicate corect.

Servicii

```
@Injectable({
  providedIn: "root",
})
export class FirmService {
  private baseUrl = environment.apiUrl;

  constructor(private http: HttpClient) {}

  public addFirm(firm: IFirm, isClient: boolean = true) {
    return this.http.post<IFirm>(
      `${this.baseUrl}/Firm/AddFirm/${isClient}`,
      firm
    );
  }

  public editFirm(firm: IFirm, isClient: boolean = true) {
    return this.http.put<IFirm>(
      `${this.baseUrl}/Firm/EditFirm/${isClient}`,
      firm
    );
  }
```

Figura 3.3: Exemplu Serviciu Angular

În Angular, serviciile sunt utilizate pentru a organiza și partaja logica aplicației, precum și pentru a realiza operații cum ar fi manipularea datelor, comunicarea cu API-uri externe și gestionarea stării aplicației. Serviciile sunt clase care pot fi injectate în alte componente sau servicii, oferind un mod centralizat de a gestiona funcționalități comune.

Serviciul din figura 3.1 este responsabil pentru gestionarea operațiilor legate de entitatea Firm (firmă), cum ar fi adăugarea și editarea informațiilor despre firmă.

- **Decoratorul @Injectable**

  Decoratorul Angular care marchează clasa ca fiind un serviciu injectabil.

- **private baseUrl = environment.apiUrl;**

baseUrl: Stochează URL-ul de bază pentru API, preluat din fișierul de mediu (environment.apiUrl).

- **constructor(private http: HttpClient) {}**

  Constructorul injectează HttpClient, care este utilizat pentru a realiza cereri HTTP.

În codul prezentat, sunt utilizate interfețe pentru a tipiza obiectele de firmă IFirm. Interfețele sunt folosite pentru a defini structura obiectelor și pentru a asigura tipul de date în TypeScript.

```
export interface IFirm {
  id: number;
  name: string;
  cui: string;
  regCom?: string | null;
  address: string;
  county: string;
  city: string;
}
```

Figura 3.4 Exemplu Interfață Angular

Bibliografie

[1] Freeman, Adam. Pro Angular: Build Powerful and Dynamic Web Apps. 5th ed. Apress, 2022.
[2] Ingeno, Joseph. Software Architect's Handbook. Packt Publishing, 2018.
[3] Uluca, Doughan. Angular for Enterprise Applications. 3rd ed. 2024.
[4] https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli
[5] Seeman, Mark. Dependency Injection Principles, Practices, and Patterns, 2019
[6] https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure
[7] Clean Architecture: A Craftsman's Guide to Software Structure and Design by Robert C Martin