

GESTIUNEA PORTOFOLIILOR DE PRACTICĂ ALE STUDENȚILOR

Candidat: Christian-Alexandru Drăgoi

Coordonator științific: Conf.dr.ing. Ciprian-Bogdan Chirilă

Sesiunea: Iunie 2025

REZUMAT

Lucrarea propune realizarea unei aplicații web pentru gestionarea portofoliilor de practică ale studenților din cadrul unei instituții de învățământ superior. Aplicația este dezvoltată folosind tehnologii precum Java, Spring Boot, Thymeleaf pentru integrarea cu partea de interfață, și MySQL pentru gestionarea datelor. Pentru generarea documentelor PDF semnate este utilizată biblioteca open-source OpenPDF.

Aplicația oferă funcționalități adaptate fiecărui tip de utilizator: prodecan (administrator), student, tutore și cadru didactic. Prodecanul are acces la funcționalități avansate precum importul studenților din fișiere CSV și generarea automată de parole pentru conturile lor, salvează într-un alt fișier CSV, urmând ca aceștia să poată schimba parola ulterior. De asemenea, acesta poate adăuga și șterge manual conturile oricărui altui utilizator și poate semna digital documentele generate. Studenții pot completa un formular electronic pentru generarea portofoliului de practică, îl pot semna și descărca în format PDF, iar prin introducerea email-ului tutorelui, acesta este automat înregistrat în sistem. Tutorii și cadrele didactice pot vizualiza și semna portofoliile aferente, fără a le putea modifica.

La nivel tehnic, aplicația este organizată pe o arhitectură de tip MVC și oferă o interfață web intuitivă realizată cu ajutorul Thymeleaf. Parolele sunt generate automat pentru utilizatori și gestionate securizat, prin Spring Security.

Prin această soluție, procesul administrativ de colectare, semnare și arhivare a portofoliilor de practică este semnificativ simplificat și digitalizat, oferind o experiență eficientă și modernă pentru toți actorii implicați.

ABSTRACT

This paper presents the development of a web application for managing student internship portfolios within a higher education institution. The application is built using technologies such as Java, Spring Boot, Thymeleaf for frontend integration, and MySQL for data management. For generating signed PDF documents, the open-source OpenPDF library is used.

The application provides role-based functionalities for each type of user: vice-dean (administrator), student, tutor, and faculty member. The vice-dean has access to advanced features such as importing students from CSV files and automatically generating passwords for their accounts, which are saved in another CSV file. Students can later change their passwords. Additionally, the administrator can manually add or delete any user account and digitally sign the generated documents. Students can fill out an electronic form to generate their internship portfolio, digitally sign it, and download it as a PDF. When a student enters the tutor's email address, the tutor is automatically registered in the system. Tutors and faculty members can view and sign the portfolios assigned to them, without the ability to modify them.

From a technical perspective, the application is based on an MVC architecture and provides an intuitive web interface implemented using Thymeleaf. User passwords are automatically generated and securely managed via Spring Security.

This solution significantly simplifies and digitizes the administrative process of collecting, signing, and archiving internship portfolios, offering an efficient and modern experience for all users involved.

CUPRINS

1. INTRODUCERE	5
1.1. MOTIVAȚIE	5
1.2. STRUCTURA LUCRĂRII	5
2. TEHNOLOGII FOLOSITE	6
2.1. LIMBAJUL JAVA	6
2.2. BIBLIOTECA SPRING BOOT	7
2.3. UTILITARUL APACHE MAVEN	9
2.4. BIBLIOTECA THYMELEAF	10
2.5. MOTORUL DE BAZE DE DATE MYSQL	11
2.6. BIBLIOTECA JPA	12
2.7. TEHNOLOGII FOLOSITE PENTRU GESTIONAREA FIȘIERELOR DE DATE ȘI PREZENTARE	13
3. SPECIFICAȚIILE PROIECTULUI	15
3.1. SCOPUL APLICAȚIEI	15
3.2. INTERFAȚA PRODECANULUI(ADMINISTRATOR)	15
3.3. INTERFAȚA STUDENTULUI	16
3.4. INTERFAȚA CADRULUI DIDACTIC SUPERVIZOR	16
3.5. INTERFAȚA TUTORELUI DE PRACTICĂ	16
4. PROIECTAREA APLICAȚIEI WEB	17
4.1. ARHITECTURA GENERALĂ A APLICAȚIEI	17
4.2. PROIECTAREA BAZEI DE DATE	21
4.3. PROIECTAREA CLASELOR JAVA	23
5. IMPLEMENTAREA APLICAȚIEI WEB	25
5.1. STRUCTURA DIRECTOARELOR	25
5.2. PARTEA DE BACKEND	27
5.3. PARTEA DE FRONTEND	45
6. TESTARE	57
7. CONCLUZII	58
BIBLIOGRAFIE	59

1. INTRODUCERE

1.1 MOTIVAȚIE

Digitalizarea proceselor educaționale reprezintă o direcție esențială pentru modernizarea învățământului superior. În contextul în care tot mai multe activități academice și administrative se desfășoară online, gestionarea portofoliilor de practică ale studenților rămâne adesea un proces manual și consumator de timp. Transmiterea documentelor prin email sau fizic, semnarea fizică și lipsa unei baze de date centralizate îngreunează monitorizarea eficientă a practicii obligatorii a studenților și colaborarea dintre părțile implicate (studenți, tutori coordonatori din firme, cadre didactice).

Această lucrare își propune dezvoltarea unei aplicații web care să răspundă acestor nevoi, oferind un cadru pentru completarea, semnarea și vizualizarea portofoliilor de practică.

1.2 STRUCTURA LUCRĂRII

Capitolul 1: INTRODUCERE - Acest capitol conturează contextul general al lucrării, evidențiind problema practică legată de gestionarea portofoliilor de practică ale studenților și motivând necesitatea digitalizării acestui proces printr-o aplicație web dedicată.

Capitolul 2: TEHNOLOGII FOLOSITE - Sunt prezentate principalele tehnologii utilizate în dezvoltarea aplicației, atât pentru partea de frontend, cât și pentru backend, împreună cu justificarea alegerii acestora.

Capitolul 3: SPECIFICAȚIILE PROIECTULUI - Capitolul descrie cerințele funcționale și nefuncționale ale aplicației, scenariile de utilizare și structura entităților implicate în sistem, precum și modelul conceptual al bazei de date.

Capitolul 4: PROIECTAREA APLICAȚIEI WEB - Este detaliată arhitectura aplicației, structura modulelor, componentele frontend și backend, precum și diagrama UML a aplicației.

Capitolul 5: IMPLEMENTAREA APLICAȚIEI WEB - Sunt descrise etapele concrete de implementare a aplicației, principalele funcționalități dezvoltate și interacțiunea dintre utilizatori și sistem.

Capitolul 6: TESTARE - Include scenariile de testare manuală, eventualele erori identificate și modul în care acestea au fost remediate.

Capitolul 7: CONCLUZII - Reunește concluziile rezultate din dezvoltarea proiectului, impactul aplicației și posibile direcții de extindere sau îmbunătățiri viitoare.

BIBLIOGRAFIE

2. TEHNOLOGII FOLOSITE

2.1 LIMBAJUL JAVA

Java este un limbaj de programare orientat pe obiecte, flexibil și portabil, care a fost introdus pentru prima dată de Sun Microsystems în 1995 și menținut acum de Oracle.[8] Este unul dintre cele mai populare limbaje de programare datorită portabilității sale și ecosistemului extins de biblioteci și framework-uri.

Java este un limbaj de programare de nivel înalt, cu o sintaxă inspirată de limbajul C++, însă conceput pentru a oferi o mai mare portabilitate și simplitate.[8] Spre deosebire de limbajele compilate direct în cod mașină, Java utilizează un model de compilare intermediar: codul sursă este transformat în bytecode, care este ulterior executat de Java Virtual Machine (JVM), mecanism care permite o independență totală față de platformă, deoarece bytecode-ul poate fi interpretat de JVM pe orice sistem de operare compatibil.

Un aspect esențial al limbajului Java este orientarea sa pe obiecte. Programarea orientată pe obiecte reprezintă o metodă prin care programele sunt organizate ca și colecții de obiecte ce cooperează între ele. Fiecare obiect este o instanță a unei clase, și fiecare clasă este membră a unei ierarhii de clase legate prin relații de moștenire. Java oferă suport complet pentru moștenire, polimorfism, încapsulare și abstractizare.[7]

Moștenirea permite crearea subclaselor (clase noi pe baza altor clase existente numite superclase). Această metodă reduce duplicarea codului.[4][7]

Polimorfismul permite utilizarea aceleiași interfețe sau metode în contexte diferite, în funcție de tipul concret al obiectului la momentul execuției. Strâns legat de moștenire, polimorfismul este esențial pentru override și legare dinamică. Override reprezintă metoda de suprascriere a unei metode dintr-o superclasă într-o subclasă. Metodele se pot suprascrie doar dacă au semnătură (tip, nume și parametrii) identică cu cele din superclasă. Legarea dinamică permite apelarea metodelor overriden în funcție de felul concret al obiectului referit la acel moment, stabilite la rularea programului, deci dinamic.

Încapsularea reprezintă metoda prin care datele interne ale unui obiect sunt ascunse altor componente. Acestea pot fi accesate doar prin metode publice bine definite (gettere și settere).

Abstractizare presupune definirea caracteristicilor esențiale ale obiectelor și ascunderea detaliilor de implementare. Astfel, fiecare subclasă va putea interpreta oricum metodele comune ale interfeței sau clasei abstracte pe care o moștenește.

Această abordare permite dezvoltarea de aplicații modulare și ușor de întreținut, unde fiecare componentă a aplicației (precum entitățile de date, logica de business și interfețele grafice) este structurată în clase și pachete specifice.[5][6]

Java oferă un sistem robust de gestionare a excepțiilor, esențial pentru crearea unor aplicații stabile și sigure.[1] Mecanismul de tratare a excepțiilor se bazează pe blocuri try-catch-finally și permite separarea clară a codului funcțional de codul de tratare a erorilor. Java distinge între excepții verificate și neverificate, impunând tratarea explicită a celor

verificate la momentul compilării, ceea ce contribuie la o mai bună controlare a fluxului aplicației. Cuvântul cheie `throw` este utilizat pentru a arunca o excepție în mod explicit, în timp ce `throws` se folosește în semnătura unei metode pentru a indica faptul că metoda respectivă poate propaga o excepție către apelant.

Începând cu Java 8, limbajul oferă suport pentru programarea funcțională prin introducerea expresiilor `lambda`, interfețelor funcționale și API-urilor orientate pe fluxuri (`streams`).[1] Aceste funcționalități permit scrierea unui cod mai concis, expresiv și mai ușor de întreținut, în special în operațiuni ce implică procesări repetitive sau colecții de date.

Java este limbajul de bază al platformei Spring Boot, permițând dezvoltarea rapidă a aplicațiilor web scalabile și sigure. În cadrul proiectului realizat, Java a fost utilizat pentru a implementa toate componentele fundamentale din backend-ul aplicației: modelarea entităților și a relațiilor dintre acestea (prin intermediul JPA și Hibernate prin care se realizează și interacțiunea directă cu baza de date MySQL), gestionarea și manipularea datelor pentru a satisface cerințele proiectului (ex. crearea documentelor, setarea datei semnării, tratarea excepțiilor etc.).

Așadar, Java reprezintă o alegere bună care oferă aplicației flexibilitate și scalabilitate pentru extinderea ulterioară a funcționalităților.

2.2 BIBLIOTECA SPRING BOOT

Spring Boot este o extensie a framework-ului Spring, concepută pentru a simplifica procesul de creare a aplicațiilor Java moderne.[9][10] Un framework reprezintă un ansamblu de biblioteci, instrumente și convenții predefinite care facilitează dezvoltarea aplicațiilor software, oferind o structură clară și un mod standardizat de a organiza codul. Spre deosebire de bibliotecile obișnuite, care furnizează doar un set de funcții sau clase, un framework stabilește „scheletul” aplicației și fluxul principal de control. Astfel, framework-ul permite dezvoltatorilor să se concentreze pe logica specifică a aplicației, fără a fi nevoiți să configureze manual fiecare detaliu al infrastructurii. Spring Boot are ca scop principal reducerea configurărilor manuale necesare într-o aplicație Spring tradițională, oferind o structură predefinită și un sistem de auto-configurare care permite dezvoltarea rapidă a aplicațiilor web, bazată pe arhitectura Model-View-Controller (MVC), cu autentificare și autorizare, acces la baze de date și expunere la interfețe REST.[3][9]

Arhitectura MVC este un model clasic de organizare a aplicațiilor, separând clar responsabilitățile între logica de afaceri, prezentarea datelor și manipularea cererilor utilizatorilor.[3] Modelul reprezintă datele aplicației și regulile care definesc organizarea acestora. Astfel, datele sunt implementate sub forma unor clase cu adnotarea „`@Entity`”. Relațiile dintre componente sunt definite prin adnotări precum „`@OneToMany`”, „`@ManyToOne`”, „`@JoinColumn`”. Aceste adnotări permit maparea obiectelor Java cu tabelele din baza de date prin JPA și Hibernate.

Din punct de vedere tehnic, Spring Boot se bazează pe mai multe concept fundamentale, precum Inversion of Control (crearea și gestionarea componentelor, programatorul gestionând doar modul prin care acestea comunică), Dependency

Injection(injectarea automată a dependențelor necesare între clase – ex. servicii sau repository) și Aspect-Oriented Programing(permite aplicarea funcționalităților transversale precum logarea fără a mai modifica direct logica principală a aplicației).[9][10]

Framework-ul Spring Boot beneficiază de componente intermediare(@Controller, @Service, @Repository).[9][10] Ele sunt gestionate de containerul IoC al Spring Boot, care se ocupă de crearea și injectarea automată a instanțelor necesare (Dependency Injection). Controller-ul se ocupă cu cererile de la utilizator, cum ar fi cereri de tip GET sau POST. După ce le procesează, stabilește ce view va fi folosit sau ce date vor fi transmise. În fiecare clasă de acest tip se folosește adnotarea „@Controller”, iar fiecare metodă are ca adnotare „@GetMapping” sau „@PostMapping”, în funcție de tipul cererii. Service-ul și Repository-ul(cu adnotările „@Service”, respectiv „@Repository”) contribuie la separarea clară a responsabilităților. Clasa Service se ocupă cu logica principală(validarea, manipularea datelor), iar clasa Repository implementează JpaRepository și permite interacțiunea directă cu baza de date facilitând operațiile CRUD(create, read, update, delete). Astfel, într-o aplicație Spring Boot MVC, atunci când un utilizator accesează o pagină web, se execută următorul flux: cererea HTTP este preluată de un Controller, care poate apela un Service pentru procesarea logicii de afaceri, iar la rândul său acesta poate folosi un Repository pentru accesul la baza de date. În final, Controllerul returnează o pagină Thymeleaf sau un răspuns JSON, în funcție de context.[5][6] Această abordare duce la un cod modular, ușor de întreținut și testabil.

Structura MVC în Spring Boot aduce mai multe avantaje, cum ar fi separarea clară a activităților(logica de afaceri, interfața și persistența datelor sunt bine delimitate), ușurința testării și extinderii aplicației(fiecare componentă poate fi dezvoltată și testată independent), reducerea duplicării codului și creșterea reutilizabilității prin serviciile partajate, integrare nativă cu bazele de date relaționale și framework-uri de frontend (MySQL, Thymeleaf).[5][6]

De asemenea, Spring Boot pune la dispoziția dezvoltatorilor un sistem de configurație centralizat (application.properties sau application.yml). Astfel, pot fi definite rapid setările legate de portul serverului, baza de date, politicile de securitate și altele.

Unul dintre cele mai importante avantaje ale Spring Boot este faptul că include un server web încorporat (precum Tomcat), permițând rularea aplicației direct, fără a mai fi necesară instalarea și configurarea unui server extern. De asemenea, integrează ușor alte module din ecosistemul Spring, precum Spring Data JPA pentru lucrul cu baze de date, Spring Security pentru autentificare și autorizare, și Spring Web pentru expunerea de interfețe web. Unul dintre cele mai importante module este Spring Security, care oferă un cadru standardizat pentru autentificare (login) și autorizare (roluri și permisiuni). Include suport pentru criptarea parolilor (ex: BCrypt) și controlul accesului pe URL-uri (hasRole, hasAuthority). De asemenea, Spring Boot oferă acces la mai multe module, cum ar fi suport integrat pentru testarea unitară și de integrare prin spring-boot-starter-test, care conține JUnit, Mockito și alte biblioteci. Testarea este esențială în orice proiect serios, iar Spring facilitează testarea controlerelor, a serviciilor și a integrării cu baza de date (prin @DataJpaTest). Framework-ul permite și configurarea aplicației pe „profiluri” (ex: development, testing, production). Astfel, se pot include setări diferite (ex: baza de date, nivelul de logare) în fișierele application-dev.properties sau application-prod.properties. La

rule, se specifică care profil este activ (`spring.profiles.active=dev`), iar Spring Boot aplică automat configurările potrivite. Aceste module predefinite adaugă automat toate bibliotecile și configurațiile necesare funcționalităților respective.[10]

Datorită simplității în utilizare, flexibilității și suportului oferit de comunitate, Spring Boot a devenit una dintre cele mai populare opțiuni pentru dezvoltarea aplicațiilor web Java, fiind utilizat împreună cu unele module enumerate mai sus și în cadrul acestei lucrări pentru a implementa funcționalitățile propuse. Prin adoptarea arhitecturii MVC și integrarea completă a acesteia în Spring Boot, aplicația dobândește o structură clară și scalabilă.

2.3 UTILITARUL APACHE MAVEN

Apache Maven este unul dintre cele mai utilizate instrumente pentru automatizarea procesului de construire (build) în ecosistemul Java. Scopul principal al acestui tool este acela de a furniza un sistem complet și inteligent pentru gestionarea proiectelor software.

Maven funcționează pe baza unui concept central numit Project Object Model (POM), definit în fișierul `pom.xml`. [11] Acest fișier descrie toate aspectele importante ale unui proiect: meta datele aplicației (nume, versiune, autor), structura, dependențele necesare, plugin-urile, comenzile de build și configurațiile asociate. Spre deosebire de alte sisteme de build tradiționale care presupun un control granular al pașilor de compilare și rule, Maven pune accent pe convenții în locul configurațiilor, reducând semnificativ complexitatea proiectelor.

Unul dintre cele mai valoroase aspecte ale lui Maven este capacitatea sa de gestionare automată a dependențelor. În loc să descarce manual biblioteci externe și să le includă în proiect, dezvoltatorii pot specifica în `pom.xml` ce biblioteci sunt necesare, iar Maven va descărca automat versiunile corecte dintr-un repository central (cum ar fi Maven Central). Acest lucru permite o actualizare facilă a bibliotecilor și elimină problemele de compatibilitate sau duplicare a codului.

În plus, Maven oferă un set standard de cicluri de viață ale proiectului (build lifecycle) - cum ar fi `validate`, `compile`, `test`, `package`, `verify`, `install` și `deploy` - care pot fi executate cu comenzi simple, de exemplu `mvn package` pentru a crea un fișier `.jar` sau `.war`. Această standardizare permite dezvoltatorilor să colaboreze eficient, indiferent de experiență sau de mediul de dezvoltare utilizat.

De asemenea, Maven este extensibil și modular. Se pot adăuga plugin-uri care extind funcționalitățile implicite — cum ar fi `spring-boot-maven-plugin` pentru aplicații Spring Boot, `maven-compiler-plugin` pentru configurarea compilatorului Java, sau `maven-surefire-plugin` pentru testare automată. Integrarea cu diverse medii de dezvoltare (Eclipse, IntelliJ IDEA, NetBeans) este nativă, iar multe CI/CD pipelines moderne (ex. Jenkins, GitHub Actions) acceptă direct proiecte Maven.

În proiectele mari, Maven permite și definirea de proiecte multi-modul, care împart aplicația în componente reutilizabile. Fiecare modul are propriul `pom.xml`, dar poate moșteni

setări comune de la un POM părinte. Acest lucru oferă o flexibilitate sporită și o separare clară a responsabilităților în proiecte complexe.

În cadrul acestei aplicații web, Maven a fost utilizat pentru a gestiona toate dependențele esențiale (Spring Boot, Hibernate, Thymeleaf, Spring Security, MySQL Connector etc.), precum și pentru a rula aplicația în mediu local. Configurația completă a proiectului este centralizată în fișierul pom.xml, ceea ce face ca proiectul să fie ușor de clonat, rulat și extins de către alți dezvoltatori.[11]

Mai mult, datorită plugin-ului spring-boot-maven-plugin, rularea aplicației se poate face cu o singură comandă (mvn spring-boot:run), fără a fi necesară configurarea manuală a unui server de aplicații. În acest mod, Maven a contribuit atât la simplificarea procesului de dezvoltare, cât și la creșterea portabilității și predictibilității aplicației pe diverse medii.

2.4 BIBLIOTECA THYMELEAF

Thymeleaf este un motor de template modern pentru Java, open-source, utilizat în principal în aplicațiile web Spring pentru a genera pagini HTML dinamice, pornind de la șabloane predefinite și completându-le cu datele furnizate de backend.[12] Este proiectat să funcționeze atât în modul de rulare pe server (server-side rendering), cât și în modul de editare statică, ceea ce permite vizualizarea paginilor direct în browser fără a rula aplicația. Astfel, dezvoltatorii și designerii pot edita paginile în mod direct, fără a rula efectiv aplicația.

Unul dintre avantajele majore ale Thymeleaf este integrarea sa nativă cu framework-ul Spring Boot, permițând legătura directă între datele trimise de controlere și afișarea acestora în interfața utilizator, datorită integrării sale perfecte cu arhitectura MVC. Acesta oferă un set de attribute speciale (ex. th:text, th:if, th:each) care permit inserarea de date, condiții și iterații direct în codul HTML, într-un mod lizibil. Aceste attribute pot fi:

1. th:text – inserează un text dinamic, înlocuind conținutul unei etichete.
2. th:href și th:src – generează URL-uri dinamice pentru linkuri și imagini.
3. th:if și th:unless – evaluează condiții pentru a afișa sau ascunde anumite secțiuni.
4. th:each – permite iterația prin liste de obiecte, fiind esențial pentru afișarea de tabele sau liste.

Aceste attribute transformă șablonul HTML într-o interfață dinamică, în care datele transmise de backend pot fi manipulate și prezentate în funcție de logica aplicației.

Thymeleaf încurajează reutilizarea componentelor vizuale prin mecanisme de fragmentare, cu ajutorul th:replace și th:include. Se pot defini secțiuni de pagină (ex: header, footer, navbar) într-un fișier separat, care este inclus automat în toate paginile aplicației. Această abordare nu doar că reduce duplicarea codului HTML.[12]

În cadrul acestui proiect, Thymeleaf este utilizat pentru a construi toate interfețele aplicației: autentificare, completare de portofolii, vizualizare și semnare documente și așa mai departe. El face legătura între logica aplicației implementată în Java și experiența vizuală a utilizatorului, deoarece creează interfețe clare, coerente și ușor de utilizat. Folosind

th:each, datele despre utilizatori și portofolii sunt afișate în mod dinamic, iar cu ajutorul th:if, interfața este personalizată în funcție de rolurile utilizatorului logat. Fragmentarea interfeței asigură consistența vizuală și o experiență de utilizare clară și intuitivă.

Datorită suportului său extins pentru expresii dinamice, a integrării cu Spring Boot și a ușurinței de utilizare pentru dezvoltatori, Thymeleaf contribuie la realizarea unei interfețe web moderne, eficiente și adaptate nevoilor proiectului. Alegerea acestui motor de template asigură nu doar flexibilitate și scalabilitate, ci și o structură clară a interfeței, esențială pentru un proiect robust.

2.5 MOTORUL DE BAZE DE DATE MYSQL

MySQL este un sistem de gestiune a bazelor de date relaționale (RDBMS) open-source, dezvoltat inițial de compania suedeză MySQL AB și în prezent întreținut de Oracle Corporation. Este unul dintre cele mai utilizate sisteme de baze de date în aplicații web, datorită performanței, fiabilității și ușurinței de utilizare.[2]

Limbajul SQL (Structured Query Language) este standardul utilizat pentru interacțiunea cu bazele de date relaționale.[2] Prin SQL, aplicațiile pot efectua operații precum interogarea (SELECT), inserarea (INSERT), actualizarea (UPDATE) sau ștergerea (DELETE) datelor. De asemenea, SQL permite definirea structurii bazei de date (CREATE TABLE, ALTER TABLE), stabilirea relațiilor și aplicarea constrângerilor de integritate.[13]

Modelul relațional presupune organizarea datelor în tabele formate din rânduri și coloane, fiecare rând reprezentând o înregistrare, iar fiecare coloană un atribut (un câmp). Relațiile între entități se exprimă prin chei primare (identificatori unici ai fiecărui rând) și chei externe (care fac referire la chei primare din alte tabele). Acest model oferă o structură logică clară și facilitează organizarea datelor într-un mod coerent și scalabil.[2]

Într-o bază de date MySQL, datele sunt organizate în tabele relaționale, iar accesul la acestea se face prin limbajul SQL (Structured Query Language). MySQL oferă suport pentru tranzacții, relații între tabele (chei primare și externe), constrângeri de integritate, precum și mecanisme de securitate prin conturi de utilizatori și drepturi de acces.[2]

În cadrul aplicației dezvoltate în această lucrare, MySQL este utilizat pentru a stoca informații despre utilizatori (studenți, tutori, cadre didactice, prodecani), portofolii de practică și relațiile dintre aceștia. Comunicarea dintre aplicație și baza de date se face prin intermediul Spring Data JPA, care permite maparea obiectelor Java la tabelele din baza de date (prin tehnologia ORM – Object Relational Mapping). Astfel, persistarea și regăsirea datelor sunt realizate eficient și într-un mod transparent pentru dezvoltator.

Comparativ cu alte soluții de baze de date, cum ar fi PostgreSQL sau SQLite, MySQL se remarcă printr-un echilibru foarte bun între performanță și ușurință de configurare, fiind alegerea naturală în cadrul aplicațiilor web Java cu volum mediu de date și nevoi de scalabilitate moderată.[2][14]

Pentru vizualizarea și manipularea tabelelor în timpul dezvoltării aplicației a fost utilizată interfața grafică phpMyAdmin. Aceasta oferă posibilitatea de a vizualiza tabelele, a rula interogări SQL și a verifica relațiile dintre entități. De asemenea, pentru rularea aplicației

web local, a fost folosit XAMPP, un pachet software care conține serverul Apache, serverul MySQL și phpMyAdmin menționat mai sus.[2]

MySQL rămâne o soluție robustă și eficientă pentru gestionarea datelor în aplicații Java moderne. În contextul proiectului de față, el asigură nu doar stocarea persistentă a datelor despre utilizatori și portofolii, ci și integritatea și securitatea acestora, în strânsă legătură cu restul componentelor din ecosistemul Spring.

2.6 BIBLIOTECA JPA

Pentru gestionarea datelor în cadrul aplicației propuse, a fost utilizată o soluție bazată pe JPA (Java Persistence API), în combinație cu modulul Spring Data JPA din cadrul ecosistemului Spring. Această abordare permite maparea directă a obiectelor din aplicație la tabele din baza de date relațională, folosind un mecanism de tip ORM (Object-Relational Mapping).[15]

Fiecare entitate principală, precum Student, Portofoliu, Tutore sau CadruDidactic, este definită ca o clasă Java adnotată cu `@Entity`, iar câmpurile acestor clase sunt mapate la coloane din tabelele corespunzătoare din baza de date MySQL. Relațiile între entități sunt exprimate prin adnotări specifice, cum ar fi `@OneToMany`, `@ManyToOne` și `@JoinColumn`; ele reflectă corect legăturile logice și funcționale dintre obiecte (ex: un tutore verifică mai multe portofolii, cel puțin un portofoliu aparține unui student). De asemenea, adnotarea `@Id` este utilizată pentru a marca un câmp ca fiind cheia primară a entității.[17] Aceasta este obligatorie pentru orice clasă mapată la o tabelă din baza de date, deoarece fiecare entitate trebuie să fie identificată într-un mod unic. Pentru generarea automată a valorii acestei chei primare, se folosește în combinație adnotarea `@GeneratedValue`, care specifică strategia de generare. De exemplu, `strategy = GenerationType.IDENTITY` indică faptul că valoarea cheii este generată de baza de date, în general printr-un mecanism de auto-increment.

Spring Data JPA este un modul din cadrul Spring care extinde funcționalitatea specificației JPA și utilizează în mod implicit implementarea Hibernate pentru maparea obiect-relațională. Interfețele Repository din cadrul aplicației extind `JpaRepository`, oferind implicit metode pentru salvarea, actualizarea, ștergerea și regăsirea entităților (`save`, `findById`, `deleteById`, `findAll` etc.). În plus, Spring Data permite definirea de metode personalizate pe baza denumirii acestora, facilitând interogări complexe fără a scrie manual cod SQL.[15]

Configurarea conexiunii la baza de date MySQL este realizată centralizat în fișierul `application.properties`, unde sunt definite detalii precum URL-ul bazei de date, utilizatorul și parola, dialectul SQL utilizat (`hibernate.dialect`) și strategia de actualizare a schemei bazei de date (`spring.jpa.hibernate.ddl-auto=update`).[13]

În Java Persistence API (JPA), moștenirea între entități poate fi mapată în baza de date relațională prin intermediul adnotării `@Inheritance`, care definește strategia de stocare a datelor pentru clasele aflate într-o relație de moștenire. Există trei strategii principale de mapare: `SINGLE_TABLE`, `TABLE_PER_CLASS` și `JOINED`. Fiecare au propriile avantaje și compromisuri.[17]

Strategia `InheritanceType.SINGLE_TABLE` presupune stocarea tuturor entităților din ierarhia de moștenire într-un singur tabel. Acest tabel va conține toate coloanele necesare pentru toate subclasele, iar diferențierea tipurilor de entități se realizează cu ajutorul unei coloane discriminatorii (definită cu `@DiscriminatorColumn`). Această strategie oferă performanță ridicată la interogare deoarece nu necesită alăturări (JOIN), însă poate duce la redundanță de date și tabele dificil de întreținut, mai ales când ierarhia devine complexă sau când multe coloane sunt neutilizate (populate cu valori NULL).

Strategia `InheritanceType.TABLE_PER_CLASS` creează câte un tabel separat pentru fiecare clasă din ierarhie, inclusiv pentru clasa de bază. Fiecare tabel conține doar coloanele corespunzătoare entității respective, fără relații explicite între ele. Această abordare asigură o izolare clară a datelor, dar interogările asupra clasei de bază devin costisitoare, deoarece se realizează prin UNION peste toate tabelele fiu. Totodată, acest tip de moștenire nu este întotdeauna bine susținut în toate sistemele de baze de date, precum MySQL.

Cea mai echilibrată și des utilizată strategie este `InheritanceType.JOINED`. Aceasta implică existența unei tabele pentru clasa de bază, care conține atributele comune tuturor subclaselor, și câte o tabelă separată pentru fiecare subclasă, conținând doar atributele specifice. Tabelele sunt legate prin cheia primară comună.[16] Această abordare reflectă în mod fidel principiile modelării relaționale și contribuie la evitarea redundanței. Totuși, ea implică JOIN-uri multiple în cadrul interogărilor, ceea ce poate afecta performanța în cazul unor volume mari de date.

Alegerea strategiei de moștenire trebuie să țină cont de complexitatea aplicației, de nevoile de performanță și de modul în care se dorește gestionarea datelor în baza de date. Strategia JOINED este adesea preferată în aplicațiile cu structuri clare și normalizate, în timp ce SINGLE_TABLE este potrivită pentru ierarhii simple, iar TABLE_PER_CLASS se potrivește atunci când entitățile din ierarhie nu au nevoie de relații comune sau interogări frecvente la nivelul clasei de bază. În aplicația pe baza acestei lucrări s-a folosit strategia JOIN în urma modificării structurii datelor utilizatorilor. Toți au câmpuri comune precum email și parolă, care sunt necesare pentru autentificare, iar restul datelor și rolurile rămân separate. Administratorul a fost și el legat de entitatea CadruDidactic prin strategia JOIN, deoarece este și el un cadru didactic și poate semna documentele, dar are și alte responsabilități separate.[16]

Așadar, Spring Data JPA asigură o interacțiune eficientă și sigură cu baza de date, contribuind la o structură clară a aplicației și la o logică de persistare coerentă, scalabilă și ușor de întreținut.

2.7 TEHNOLOGII PENTRU GESTIONAREA FISIERELOR DE DATE ȘI PREZENTARE

În cadrul aplicației dezvoltate, gestionarea fișierelor joacă un rol esențial în realizarea unor funcționalități-cheie, precum importul utilizatorilor, generarea de documente PDF și salvarea de resurse multimedia.

Pentru importul listelor de studenți și exportul datelor aferente (emailuri, parole), aplicația utilizează biblioteca Apache Commons CSV, o soluție open-source care oferă suport extensiv pentru lucrul cu fișiere delimitate (CSV, TSV etc.).[18] Importul se realizează prin parsarea fișierului încărcat, fiecare rând fiind tratat ca un obiect de tip CSVRecord. Datele extrase sunt validate și salvate ulterior în baza de date.

Exportul datelor în format CSV se realizează utilizând CSVFormat, care permite definirea structurii fișierului (separatori, antete, ghilimele). Această funcționalitate este folosită pentru generarea automată a unui fișier CSV care conține datele de autentificare ale studenților nou înregistrați (nume, prenume, email, parola temporară).[18]

Pentru manipularea fișierelor încărcate din interfața web (ex: fișiere CSV sau imagini pentru semnături), aplicația utilizează interfața MultipartFile din cadrul Spring.[19] Aceasta permite trimiterea și preluarea fișierelor prin cereri HTTP de tip multipart/form-data.

Fișierele încărcate sunt procesate în backend, validate și salvate temporar pe disc, sau prelucrate direct în memorie. Această abordare este folosită atât pentru importul de date în format CSV, cât și pentru gestionarea imaginilor de semnătură (în format .png) atașate documentelor PDF.

Pentru generarea automată a portofoliilor de practică în format PDF, aplicația folosește biblioteca OpenPDF, o soluție open-source derivată din vechiul iText 2.1.7.[20] Aceasta permite compunerea dinamică de documente PDF prin adăugarea de texte, tabele, imagini și semnături.

În cadrul aplicației, PDF-ul este generat în urma completării unui formular de către student și include automat datele introduse, împreună cu semnăturile tutorelui de practică din firmă, cadrului didactic și prodecanului, dacă sunt disponibile. Semnăturile sunt imagini în format .png, inserate în documentul PDF. Fișierul rezultat poate fi descărcat sau vizualizat de către toți utilizatorii.

3. SPECIFICAȚIILE PROIECTULUI

3.1 SCOPUL APLICAȚIEI

Aplicația are ca scop digitalizarea procesului de gestionare a portofoliilor de practică ale studenților din cadrul unei instituții de învățământ superior. Aceasta permite completarea, semnarea, exportarea și validarea portofoliilor, precum și administrarea utilizatorilor implicați (studenți, tutori, cadre didactice și prodecani).

Aplicația este realizată ca aplicație web și este construită folosind următoarele tehnologii: Java, Spring Boot, Thymeleaf, MySQL și biblioteca OpenPDF.

Pe lângă funcționalitățile principale ale aplicației, implementarea proiectului respectă o serie de cerințe non-funcționale esențiale pentru asigurarea unei experiențe de utilizare optimă, a siguranței datelor, precum:

5. Ușurința în utilizare: interfața aplicației este intuitivă și organizată în funcție de rolul fiecărui utilizator, astfel încât navigarea să fie naturală și accesul la funcționalități să fie rapid. Designul interfeței este simplu și accesibil, chiar și pentru utilizatori fără cunoștințe tehnice.

6. Securitatea datelor: datele personale ale utilizatorilor sunt protejate prin mecanisme standard de securitate. Parolele sunt criptate folosind algoritmul BCrypt, oferind protecție împotriva accesului neautorizat. Autentificarea se face în funcție de rol, cu drepturi diferite pentru fiecare categorie de utilizator.

7. Performanță și timp de răspuns: aplicația este construită astfel încât să răspundă rapid la acțiunile utilizatorilor. Generarea de PDF-uri, încărcarea fișierelor CSV și accesarea datelor din baza de date se realizează în timp real.

8. Fiabilitate și consistență: sistemul gestionează corect scenariile frecvente de utilizare, asigurând consistența datelor introduse, chiar și în cazul unor acțiuni repetate sau parțiale (ex: salvarea unui formular incomplet).

9. Extensibilitate: aplicația este structurată modular, ceea ce permite adăugarea ulterioară de funcționalități (ex: notificări, autentificare cu doi factori, rapoarte de practică) fără a afecta componentele existente.

10. Compatibilitate și standardizare: aplicația respectă standardele moderne ale aplicațiilor web și este compatibilă cu majoritatea browser-elor actuale (Chrome, Firefox, Edge).

3.2 INTERFAȚA PRODECANULUI(ADMINISTRATOR)

Prodecanul beneficiază de o interfață complexă, cu funcționalități de administrare. Aceasta îi permite să importe studenții din fișiere CSV și să genereze automat parole pentru conturile acestora, exportate apoi într-un fișier separat. Prodecanul poate adăuga sau șterge manual conturile altor utilizatori (studenți, tutori, cadre didactice) și poate edita propriul profil. De asemenea, interfața oferă acces la lista completă a studenților și a portofoliilor acestora,

inclusiv posibilitatea de a semna digital portofoliile prin inserarea unei semnături în documentele PDF generate. În momentul în care acesta semnează un portofoliu atribuit altui cadru didactic, documentul i se atribuie automat prodecanului. Exemplu de flux pentru rolul de administrator este prezentat în Figura 3.

3.3 INTERFAȚA STUDENTULUI

Studentii au la dispoziție o interfață care facilitează completarea și gestionarea portofoliilor lor de practică. Aceștia pot introduce email-ul tutorelui iar aplicația îl înregistrează automat și generează o parolă temporară pentru acesta. Studentul poate completa un formular de practică, genera și descărca documentul PDF, semna digital portofoliul și îl poate edita ulterior, dacă este necesar. De asemenea, interfața le permite studenților să își gestioneze propriul profil. Fiecare student poate avea unul sau mai multe portofolii de practică, reprezentând mai multe stagii de practică din diferite firme. Obiectivul studentului este de a îndeplini numărul de ore impus de instituția de învățământ superior în programul său de practică, indiferent de numărul stagiilor de practică și de firmele afiliate.

3.4 INTERFAȚA CADRULUI DIDACTIC SUPERVIZOR

Interfața cadrului didactic oferă acces exclusiv la vizualizarea și semnarea portofoliilor de practică asignate lor. Ei nu pot vizualiza portofoliile atribuite altor cadre didactice. Parola de acces este generată automat, dar poate fi editată ulterior de către utilizator, precum și restul datelor(ex. nume, prenume, funcție, specializare etc.). Această interfață asigură implicarea cadrului didactic în validarea și aprobarea portofoliilor, fără a permite modificarea conținutului acestora. Fluxul detaliat al semnării este prezentat în Figura 4.

3.5 INTERFAȚA TUTORELUI DE PRACTICĂ

Similar cu cea a cadrelor didactice, interfața tutorilor le permite să vizualizeze și să semneze portofoliile studenților coordonați. Fiecare tutore poate vizualiza și semna exclusiv portofolii atribuite de către student. Această atribuire a portofoliilor se face prin completarea unui câmp cu email-ul tutorelui. În cazul în care nu există un utilizator cu acel email, aplicația generează un cont nou cu o parolă unică. Apoi asignează portofoliul către acest utilizator sau către utilizatorul care are deja cont cu acea adresă de email, în cazul în care un cont a fost deja generat. Parola poate fi ulterior modificată, împreună cu celelalte date personale. Pentru utilizatorii noi generați, care nu au încă completate datele personale(nume, funcție, telefon etc.), după prima logare aplicația redirecționează tutorele direct pe pagina de editare a profilului, urmând ca la următoarele logări să fie redirecționat către pagina principală. Tutorii nu pot modifica conținutul portofoliilor, ci doar să își ofere semnătura digitală.

4. PROIECTAREA APLICAȚIEI WEB

4.1 ARHITECTURA GENERALĂ A APLICAȚIEI

Aplicația este construită pe baza arhitecturii MVC (Model-View-Controller), respectând principiile de separare a responsabilităților între modelul de date, interfața cu utilizatorul și logica de control, așa cum este ilustrat în Figura 1. În același timp, aceasta urmează o arhitectură monolitică, întrucât toate funcționalitățile sunt dezvoltate și rulate în cadrul unei singure aplicații Spring Boot, cu o bază de cod comună și o bază de date unificată.[3]

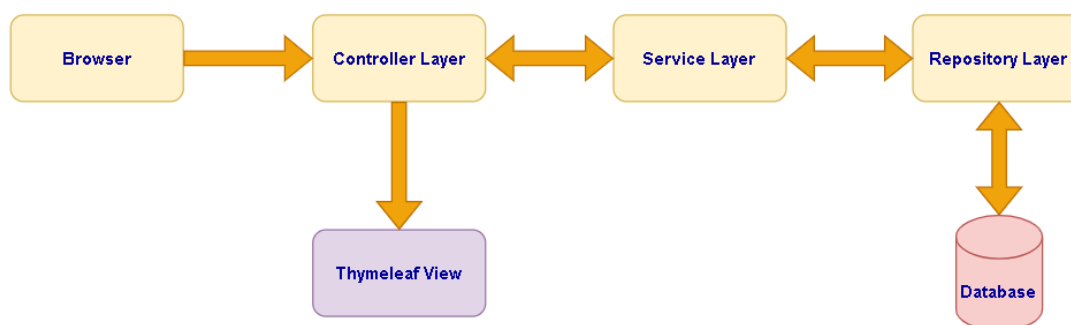


Figura 1 – Arhitectura MVC

În continuare sunt descrise și prezentate diagrame care descriu câteva funcționalități cheie bazate pe arhitectura MVC. Aceste diagrame au un rol pur ilustrativ, fiind utilizate pentru proiectarea aplicației și pentru evidențierea modului în care este structurată arhitectura MVC, fără a acoperi în detaliu toate scenariile de utilizare sau implementările efective ale mesajelor HTTP, procesarea datelor respectiv interogărilor SQL.

Procesul de creare a unui portofoliu este descris în Figura 2, care evidențiază pașii principali. În Figura 2 este reprezentată o diagramă care ilustrează fluxul de creare a unui portofoliu nou de către un utilizator cu rolul de student. Procesul începe cu trimiterea formularului de completare a portofoliului, care este recepționat de PortofoliuController. Controllerul validează datele introduse; în cazul în care acestea sunt invalide, utilizatorul este redirecționat către aceeași pagină cu un mesaj de eroare. Dacă datele sunt corecte, se continuă cu atribuirea cadrului didactic la portofoliu și căutarea tutorului după adresa de email. Dacă tutorul este găsit în baza de date, acesta este asociat direct portofoliului. În caz contrar, este creat automat un cont de tutor prin intermediul TutorService, iar noul tutor este salvat în baza de date. După ce toate entitățile sunt asociate corect (Student, CadruDidactic, Tutor), portofoliul este salvat în baza de date și studentul este redirecționat către pagina principală. De menționat este faptul că toate semnăturile au valoarea false la crearea fiecărui portofoliu, ceea ce implică verificarea și validarea documentelor de către toți actorii.

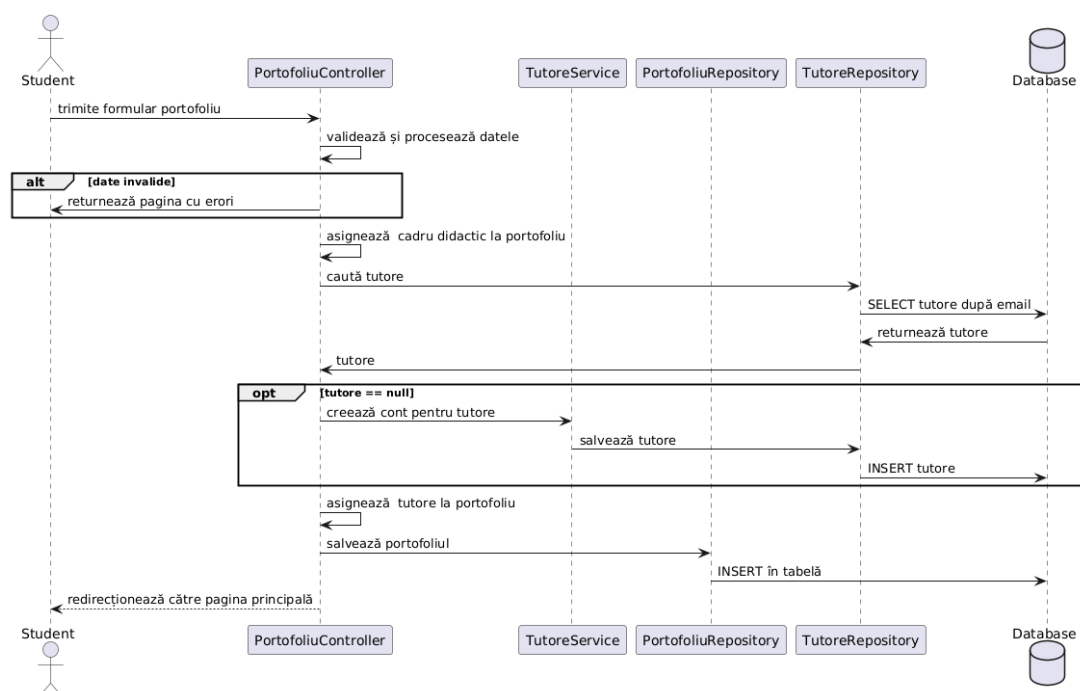


Figura 2 – Diagramă UML pentru crearea unui portofoliu nou

Fluxul de import al studenților de către administrator prin fișier CSV este ilustrat în Figura 3. Diagrama din Figura 3 ilustrează interacțiunea dintre administrator și sistem în momentul adăugării studenților folosind un fișier CSV care conține informațiile acestora. Procesul începe cu trimiterea fișierului de către Admin către AdminController, care trimite mai departe fișierul către serviciul AdminService. AdminService procesează datele parcurgând fiecare rând din fișier. Astfel, extrage datele studenților și generează automat parole aleatorii. Pe baza acestor informații, creează conturi noi pentru fiecare student, salvându-le în baza de date și într-un fișier CSV separat, utilizat ulterior pentru distribuirea credențialelor. Dacă fișierul este încărcat cu succes, sistemul returnează un mesaj de succes și actualizează lista studenților. În caz contrar, este trimis un mesaj de eroare către administrator.

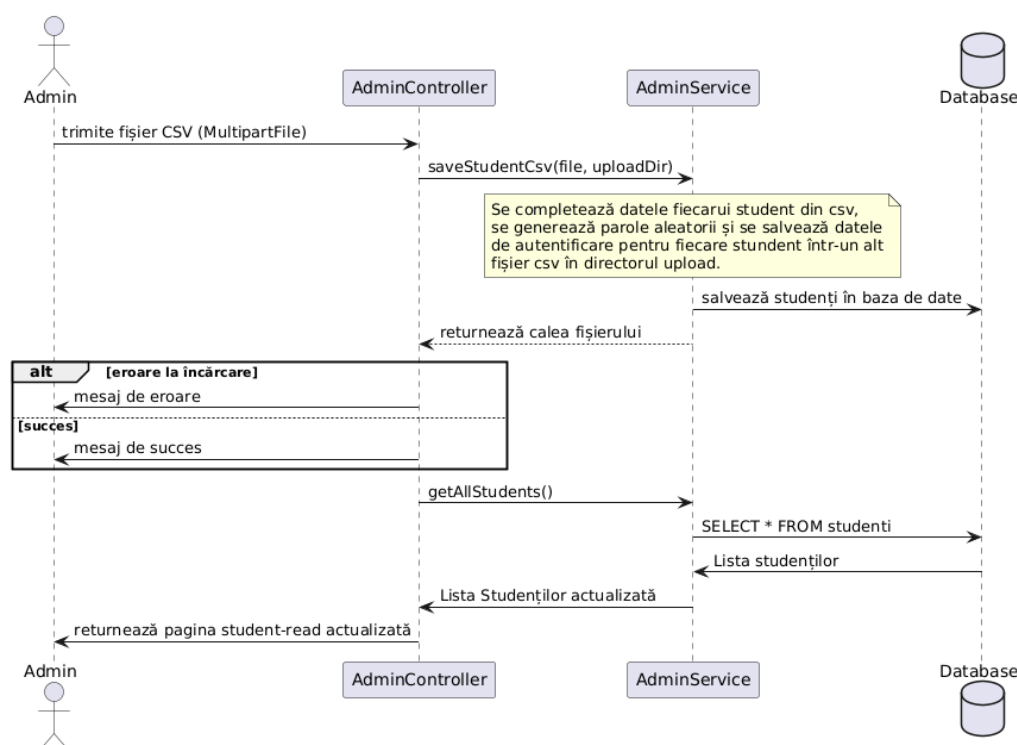


Figura 3 – Diagrama UML pentru adăugarea studenților via CSV

Procesul de semnare electronică a unui portofoliu de către un cadru didactic este prezentat în Figura 4. Diagrama din Figura 4 prezintă procesul prin care un utilizator cu rol de cadru didactic semnează un portofoliu de practică. Fluxul începe cu acțiunea de semnare, declanșată din interfața utilizatorului, care este preluată de CadruDidacticController. Controllerul caută portofoliul în baza de date, folosind PortofoliuRepository și se generează o eroare corespunzătoare dacă portofoliul nu este găsit (portofoliu == null). În cazul în care portofoliul este valid, se verifică existența semnăturii cadrului didactic. Dacă semnătura este absentă, se semnalează o eroare internă de tip signCadruError. În caz contrar, semnătura cadrului didactic este setată pe true, iar portofoliul este actualizat în baza de date. La final, utilizatorul este redirecționat către pagina de vizualizare a portofoliului actualizat.

Asemănător se întâmplă și pentru semnarea portofoliului de către un tutor de practică, mai puțin setarea datei semnării. Data semnării se generează odată cu semnarea de către ultimul actor, adică de către cadrul didactic.

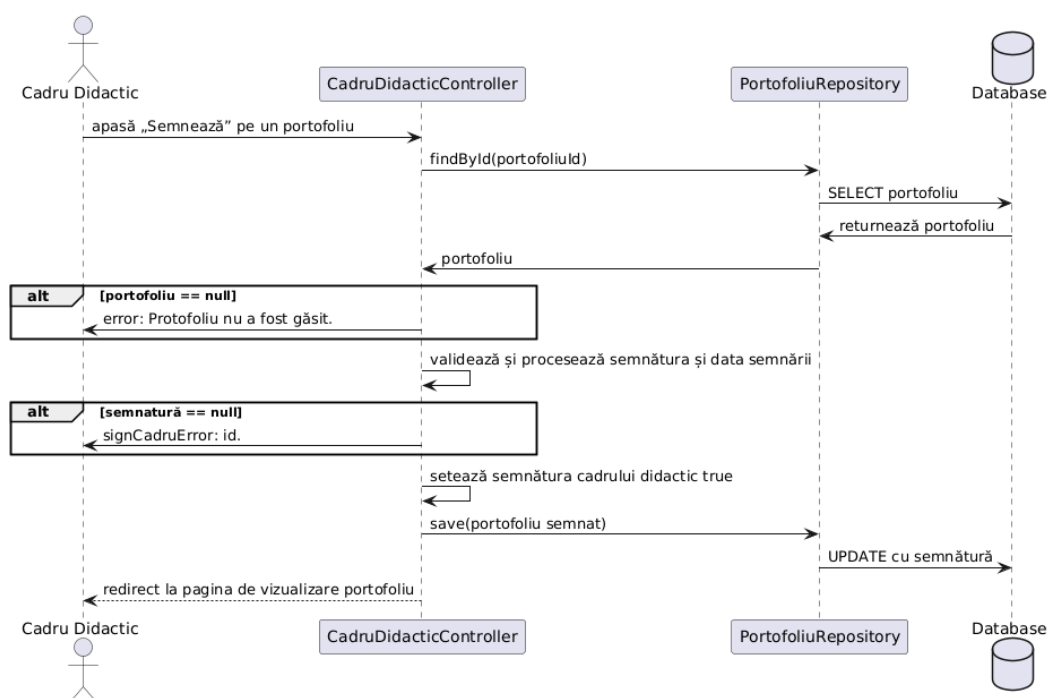


Figura 4 – Diagramă UML pentru semnarea unui portofoliu de către un cadru didactic

Figura 5 evidențiază actualizarea datelor personale ale unui tutor. În Figura 5 este ilustrată diagrama prin care se descriu acțiunile pe care un utilizator cu rol de tutor le poate realiza pentru actualizarea propriului profil în cadrul aplicației. Sunt reprezentate trei situații posibile: modificarea datelor personale, actualizarea semnăturii și schimbarea parolei.

În cazul actualizării datelor personale, tutorii completează un formular cu informații precum nume, funcție sau număr de telefon. Aceste date sunt transmise către controller, care le validează și le trimite mai departe către serviciul aplicației, apoi se salvează în baza de date. După procesare, utilizatorul este redirecționat către pagina principală.

Pentru actualizarea semnăturii, utilizatorul încarcă un fișier de tip .png care conține semnătura digitală. Fișierul este procesat și validat, iar semnătura este înregistrată în baza de date. Sistemul oferă apoi un mesaj de succes și revine la pagina inițială. În cazul în care fișierul nu e validat, utilizatorul primește un mesaj de eroare.

În cazul schimbării parolei, tutorii solicită actualizarea acesteia și e redirecționat către o altă pagină. Parola nouă este criptată și salvată în mod securizat în baza de date. După finalizarea modificării, utilizatorul este delogat automat pentru a se reconecta cu noile date de autentificare.

Fluxul de actualizare a profilului este similar pentru toți actorii din aplicație, însă fiecare rol beneficiază de un formular adaptat care conține doar câmpurile relevante pentru acel tip de utilizator.

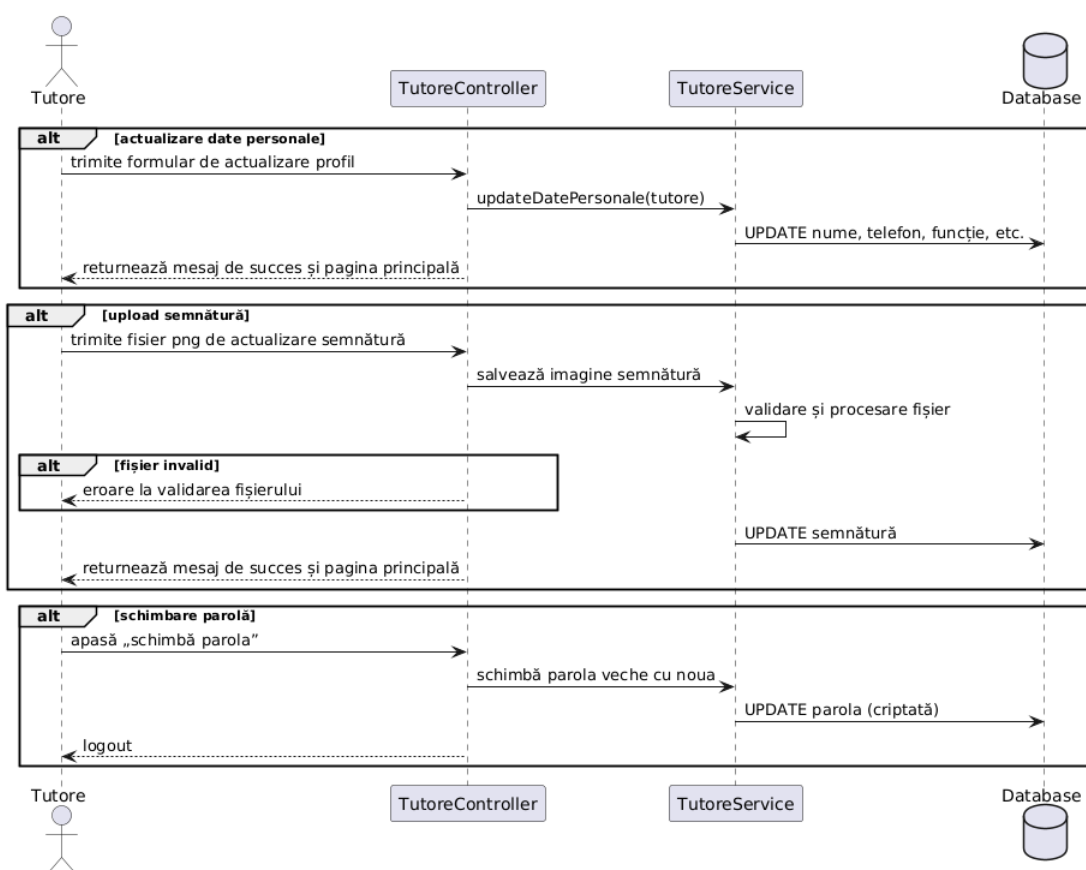


Figura 5 – Diagramă UML pentru modificarea profilului unui tutore

4.2 PROIECTAREA BAZEI DE DATE

Aplicația utilizează Hibernate (JPA) pentru generarea automată a bazei de date. Configurarea se face în application.properties, iar entitățile Java sunt mapate în tabele SQL.

În urma analizei cerințelor și specificațiilor, am ajuns la concluzia că este nevoie de patru entități: portofoliu, student, cadru didactic, tutore de practică. Atributele acestora sunt enumerate în Tabelul 1.

Tabelul 1. Atributele entităților în urma analizei cerințelor

PORTOFOLIU	STUDENT	CADRU DIDACTIC	TUTORE
-loculDesfășurării	-nume	-nume	-nume
-durataPracticii	-prenume	-prenume	-prenume
-dataÎnceput	-cnp	-functie	-functie
-dataSfârșit	-dataNasterii	-specializare	-telefon
-dataSemnării	-loculNasterii	-telefon	-semnatura
-orar	-cetătenie	-semnatura	-email
-LocațiiExtra	-serieCi	-email	-password
-competenteNecesare	-numarCi	-password	
	-adresa		

-complementare InvatamantPractica -tematicaSiSarcini -competenteDobandite -modDePregatire -activitatiPlanificate -observatii	-anUniversitar -facultate -specializare -anDeStudiu -telefon -semnatura -email -password		
--	---	--	--

Atributele portofoliilor reprezintă fiecare câmp ce trebuie completat de student și ceilalți actori în document. Atributele studentului reprezintă datele cu care acesta se poate autentifica și semnătura, respectiv datele cu care e salvat în baza de date a universității. Astfel, prodecanul poate încărca direct un fișier CSV cu datele studenților direct din baza de date a facultății. Cadrul didactic și tutorele au ca și atribute date de contact, de autentificare, semnături și alte câmpuri necesare completării documentului(ex. funcție, specializare).

Relațiile dintre entități se reprezintă prin chei străine generate automat de Hibernate prin adnotările corespunzătoare(@OneToMany, @ManyToOne etc.). Aceste relații sunt următoarele:

11. Un student are mai multe portofolii.
12. Un cadru didactic verifică mai multe portofolii
13. Un tutore verifică mai multe portofolii

Modelul entitate-relație al aplicației este redat în Figura 6, unde sunt vizibile principalele entități și relațiile dintre ele.

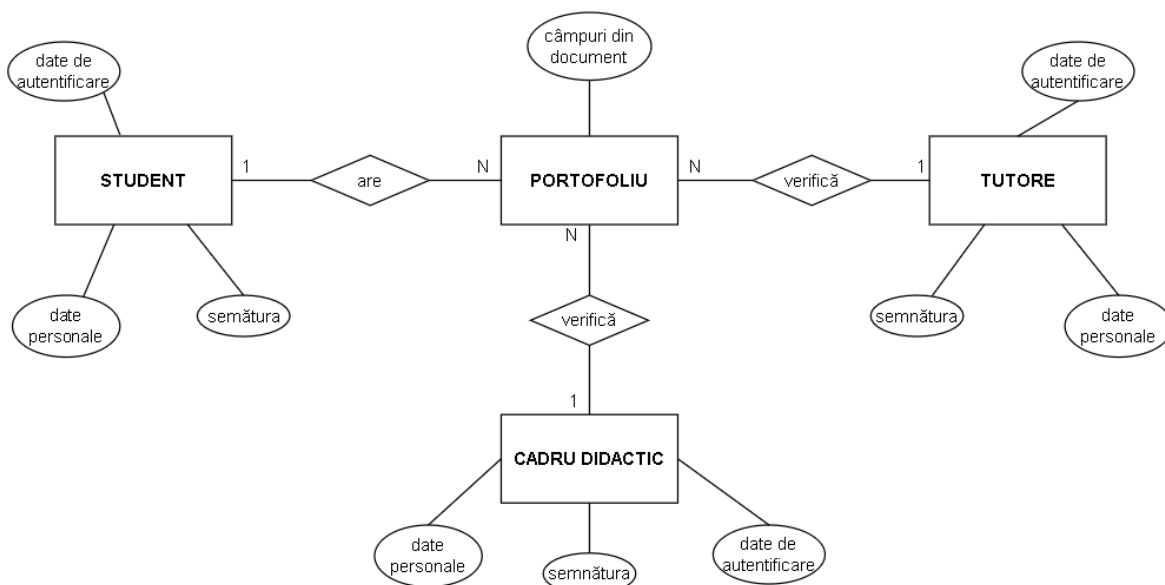


Figura 6 – Diagrama entitate relație simplificată

4.3 PROIECTAREA CLASELOR JAVA

Clasa portofoliu, precum celelalte clase, au adnotarea `@Entity` pentru a fi mapată în tabelele bazei de date. De asemenea, conține câmpuri de tip `String`, `int` și `Date` pentru datele necesare completării documentului, echivalente cu atributele entității din baza de date evidențiate mai sus și un câmp `int` cu adnotările `@Id` și `@GeneratedValue(strategy = GenerationType.IDENTITY)` pentru generarea automată de Hibernate a identificatorului unic. De asemenea, am introdus 3 câmpuri noi de tip `boolean` care evidențiază dacă portofoliul a fost semnat de fiecare dintre rolurile actorilor.

Una dintre modificările făcute pentru modularitatea programului este de a folosi moștenirea. Astfel, se folosește o clasă abstractă `User` care conține un câmp `id` de tip `int`, un `String` `email` și un `String` `password`, precum și un `enum` de tip `Role`. Această soluție ajută și pentru autentificarea utilizatorilor prin `Spring Security`. `Enum`-ul `Role` reprezintă rolul fiecărui actor, rol care ajută pentru autentificare și alte acțiuni. Aceste roluri sunt: `ADMIN`, `STUDENT`, `TUTORE`, `CADRU_DIDACTIC`. Alegerea unei clase abstracte, în detrimentul uneia concrete, reflectă faptul că în cadrul aplicației nu există instanțe de utilizator generice, ci doar tipuri specializate, fiecare cu atribute și comportamente specifice. Clasa `User` definește un set de atribute comune (precum `email`, `password`, `telefon`, `semnatura`) și oferă o structură coerentă pentru moștenire. Fiind abstractă, clasa nu poate fi instanțiată direct, ceea ce împiedică apariția unor entități incomplete sau irelevante din punct de vedere funcțional.

Restul entităților moștenesc clasa `User`, prin expresia „`extends`”. Fiecare clasă conține atribute echivalente cu cele ale entităților din baza de date. O altă modificare este adăugarea clasei `Admin`, care moștenește la rândul ei clasa `CadruDidactic`. Diferența dintre cele două este rolul.

Figura 7 oferă o privire de ansamblu asupra claselor Java și relațiilor dintre acestea. În diagrama de clase prezentată, relațiile dintre entități reflectă modul în care componentele aplicației colaborează în cadrul procesului de gestionare a portofoliilor de practică. Aceste relații sunt esențiale pentru înțelegerea logicii aplicației, iar ele includ moștenirea, agregarea și compoziția.

În primul rând, moștenirea este exprimată prin legătura dintre clasa abstractă `User` și cele patru clase derivate: `Student`, `CadruDidactic`, `Tutore` și `Admin`, cel din urmă fiind legat de `CadruDidactic`.

Compoziția este evidențiată în relația dintre `Student` și `Portofoliu`. Aici, portofoliul este o parte esențială a studentului – fără student, portofoliul nu poate exista, ceea ce este specific unei relații de compoziție. Compoziția oferă o flexibilitate mai mare decât moștenirea, permițând schimbarea comportamentului la runtime.[4]

Agregarea este utilizată pentru a reflecta relațiile dintre `Portofoliu` și celelalte entități implicate în validarea sa, adică `Tutore` și `CadruDidactic`. Astfel, un `Tutore` sau un `CadruDidactic` poate fi asociat cu mai multe portofolii și poate exista în sistem și în absența unui anumit portofoliu.

Prin aceste relații, diagrama subliniază un model de date robust, care reflectă fidel constrângerile reale ale domeniului – atât în ceea ce privește dependențele logice, cât și

gradul de autonomie al fiecărei entități. Acest tip de modelare asigură consistență, claritate și scalabilitate în implementarea aplicației.

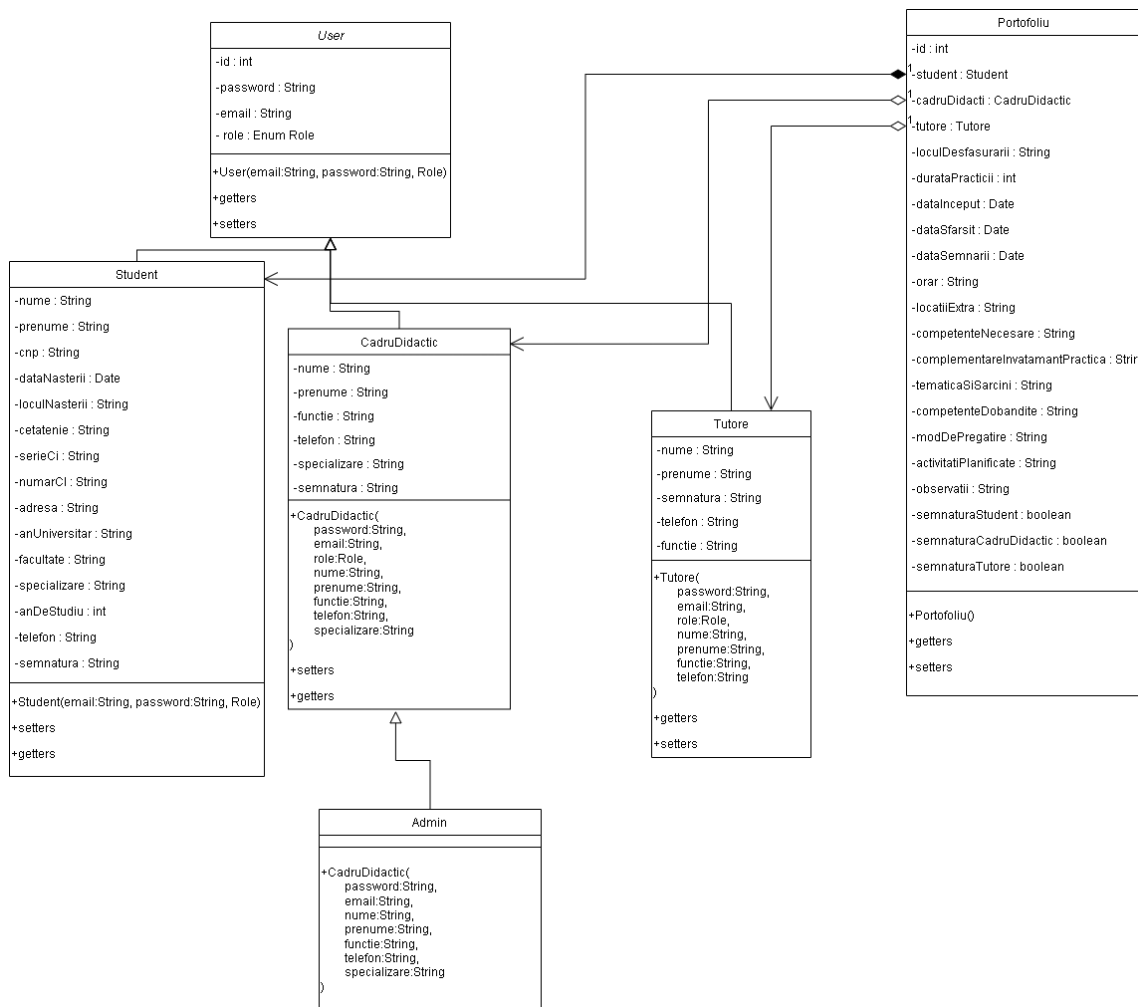


Figura 7 – Diagrama claselor Java

5. IMPLEMENTAREA APLICAȚIEI WEB

5.1 STRUCTURA DIRECTOARELOR

Structura directoarelor unei aplicații software este esențială pentru asigurarea unei organizări coerente a codului sursă, pentru facilitarea întreținerii și pentru promovarea unei dezvoltări modulare. În cazul de față, proiectul este organizat conform convențiilor impuse de Apache Maven și de arhitectura specifică aplicațiilor Spring Boot.

Directorul principal este Portofolii_Practică_UPT. Acesta reprezintă rădăcina proiectului și conține fișiere de configurare specifice mediului de dezvoltare (cum ar fi .classpath, .project, .gitignore), precum și fișierul pom.xml, care gestionează toate dependențele și plugin-urile utilizate prin Maven.

Directorul src/main/ conține codul sursă al aplicației. Acesta este împărțit în două directoare, java/(pentru backend) respectiv resources/(pentru frontend și fișierul application.properties).

Backendul aplicației este implementat în Java, utilizând cadrul Spring Boot. Codul este organizat modular, pe funcționalități, fiecare funcționalitate având propriul pachet dedicat. Nu se folosește o structură tradițională controller/service/repository globală, ci o structurare pe roluri și domenii, cum ar fi:

14. admin/ - conține clasele Admin(pentru entitatea Admin), AdminService(serviciul care se ocupă cu validarea și procesarea datelor pentru acțiunile administratorului) și AdminController(pentru preluarea cererilor HTTP).

15. student/ - conține clasele Student(pentru entitatea Student), StudentService(serviciul pentru validarea și procesarea datelor pentru anumite acțiuni legate de student), StudentController(pentru preluarea mesajelor HTTP și validarea unor date unde e cazul) și StudentRepository(pentru operații cu entitatea Student din baza de date).

16. portofoliu/ - conține clasele Portofoliu(pentru entitatea Portofoliu), PortofoliuController(pentru preluarea mesajelor HTTP și validarea respectiv procesarea datelor) și PortofoliuRepository(pentru operații cu entitatea Portofoliu din baza de date).

17. tutore/ - conține clasele Tutore(pentru entitatea Tutore), TutoreService(serviciul pentru validarea și prpcesarea datelor pentru anumite acțiuni legate de tutore), TutoreController(pentru preluarea mesajelor HTTP și validarea unor date unde e cazul) și TutoreRepository(pentru operații cu entitatea Tutore din baza de date).

18. cadruDidactic/ - conține clasele CadruDidactic(pentru entitatea CadruDidactic), CadruDidacticService(serviciul pentru validarea și prpcesarea datelor pentru anumite acțiuni legate de student), CadruDidacticController(pentru preluarea mesajelor HTTP și validarea unor date unde e cazul) și CadruDidacticRepository(pentru operații cu entitatea CadruDidactic din baza de date).

Pachetul security/ este dedicat gestionării autentificării și autorizării utilizatorilor în cadrul aplicației, prin integrarea componentelor oferite de Spring Security. Clasa centrală a

acestui pachet este SecurityConfig, unde sunt configurate politicile de securitate: ce rute pot fi accesate în funcție de rol (ex. ROLE_ADMIN, ROLE_STUDENT), metodele de autentificare permise și criptarea parolelor. Clasa CustomAuthenticationSuccessHandler este responsabilă pentru comportamentul aplicației imediat după autentificare, redirecționând utilizatorii către pagina corespunzătoare rolului pe care îl dețin. Autentificarea se bazează pe încărcarea utilizatorului din baza de date prin intermediul clasei UserDetailsServiceImpl, care implementează interfața UserDetailsService și extrage datele utilizatorului pe baza adresei de e-mail. În plus, clasa Role definește în mod clar toate rolurile existente în sistem, fiind utilizate atât la nivel de interfețe, cât și pentru restricționarea accesului la anumite acțiuni. Acest pachet contribuie la securizarea eficientă a aplicației și la personalizarea accesului în funcție de responsabilitățile fiecărui actor. În completarea acestor componente, clasa abstractă User definește structura de bază a tuturor tipurilor de utilizatori din sistem, oferind câmpuri comune precum id, email, password și role, și fiind mapată ca entitate JPA prin strategia de moștenire JOINED, ceea ce permite extinderea sa în entitățile specifice (ex. Student, Tutore, CadruDidactic). Modelul de date al aplicației este reprezentat atât la nivel logic, cât și la nivel fizic. Figura 6 prezintă diagrama entităților (ERD), evidențiind relațiile dintre entitățile persistente, în timp ce Figura 7 oferă o vedere de ansamblu asupra implementării acestora în codul Java, utilizând moștenirea. De asemenea, LoginController gestionează afișarea paginii de autentificare și procesează resetarea parolelor pentru utilizatori, adaptând comportamentul în funcție de rolul acestora. Aceste două clase contribuie la organizarea clară și reutilizarea eficientă a logicii de autentificare în întregul sistem.

Pachetul utils/ are rolul de a grupa clasele auxiliare care oferă funcționalități suport. Acesta include clasa DataInitializer, utilizată pentru popularea bazei de date cu date inițiale necesare rulării aplicației în mediul de dezvoltare, precum conturi de test sau roluri implicite. Clasa PasswordGenerator este responsabilă pentru generarea automată de parole random sigure, fiind utilizată în procesul de adăugare a studenților în baza de date. Tot în acest pachet se află și PdfGenerator, o componentă care generează documente PDF pe baza datelor introduse de utilizatori, contribuind astfel la automatizarea și standardizarea portofoliilor de practică. Prin aceste componente, utils/ asigură o separare clară a logicii auxiliare, îmbunătățind modularitatea și întreținerea codului.

Clasele Application(cu adnotarea @SpringBootApplication) și MvcConfig(cu adnotarea @Configuration) care implementează interfața WebMvcConfigurer aparțin directorului java/. Ele definesc configurația de bază a aplicației Spring Boot: clasa Application inițializează contextul aplicației și pornește serverul, în timp ce MvcConfig configurează comportamentul Web MVC, precum rutele statice, localizarea sau mapping-urile personalizate pentru view-uri.

Directorul resources/, situat în cadrul structurii src/main/, este dedicat gestionării fișierelor statice, configurațiilor și resurselor necesare la rularea aplicației. În contextul acestui proiect, el îndeplinește un dublu rol: pe de o parte, găzduiește fișierele de configurare ale aplicației backend(application.properties), iar pe de altă parte, conține interfața grafică a aplicației (frontend-ul), sub forma fișierelor HTML, CSS.

În mod specific, subdirectoarele `templates/` și `static/` sunt esențiale. Directorul `templates/` conține toate fișierele `.html` folosite de Spring Boot împreună cu motorul de template-uri Thymeleaf pentru a genera pagini dinamice pe baza datelor transmise din controller. Aici sunt organizate pagini specifice fiecărui rol din aplicație(ex: `student-index.html`, `login.html`, `tutore-portofoliu-read.html` etc.). Directorul `static/` include resursele statice precum stiluri CSS. În plus, subdirectorul `dejavu-sans/` conține mai multe fișiere `.ttf` care reprezintă fonturi din familia DejaVu Sans.

Astfel, directorul `resources/` joacă un rol central în implementarea interfeței aplicației și în configurarea sa.

În afara codului sursă, aplicația generează automat două directoare esențiale: `semnaturi/` și `upload/`.

Directorul `semnaturi/` este utilizat pentru stocarea imaginilor cu semnături digitale ale utilizatorilor. Acesta conține sub-foldere pentru fiecare categorie de utilizator: `studenti/`, `cadreDidactice/` și `tutori/`, fiecare dintre ele organizate suplimentar în funcție de ID-ul contului (exemplu: `student7/`, `cadruDidactic3/`, `tutore9/`). În interiorul acestora se salvează fișiere `.png` cu semnăturile utilizatorilor atunci când aceștia le încarcă. Această organizare permite aplicației să acceseze rapid semnătura aferentă fiecărui actor implicat în completarea și semnarea portofoliilor de practică. Astfel, semnăturile utilizatorilor se încarcă în baza de date ca niște șiruri de caractere care reprezintă exact calea către semnăturile fiecărui utilizator.

Directorul `upload/` este utilizat în special de către administrator pentru gestionarea fișierelor CSV. Aici se regăsesc fișiere precum `Students_Details.csv`, care conține informațiile originale ale studenților exportate din fișierul CSV inițial încărcat de prodecan, și `Students_Credentials.csv`, generat automat de aplicație și care include datele de autentificare (email și parolă) ale studenților importati.

Aceste directoare sunt create la rularea aplicației dacă nu există deja și contribuie la gestionarea eficientă a documentelor necesare funcționării sistemului.

5.2 PARTEA DE BACKEND

Backendul aplicației a fost realizat în Java, utilizând framework-ul Spring Boot, care oferă o platformă robustă pentru dezvoltarea aplicațiilor web moderne. Codul sursă este organizat modular, conform rolurilor din sistem - student, tutore, cadru didactic și administrator - fiecare având propriile controllere, servicii și entități. Această structurare permite o separare clară a responsabilităților, o întreținere mai ușoară a codului și o dezvoltare scalabilă.

Clasa `SecurityConfig` este responsabilă pentru configurarea politicilor de securitate ale aplicației. Ea este adnotată cu `@Configuration` și `@EnableWebSecurity`, ceea ce o face componenta centrală pentru configurarea Spring Security. Prin definirea unui `SecurityFilterChain`, autentificarea și autorizarea utilizatorilor sunt controlate în funcție de rolurile pe care aceștia le dețin.

Metoda `securityFilterChain(HttpSecurity http)` este responsabilă pentru setarea regulilor de securitate aplicabile cererilor HTTP. În primul rând, protecția CSRF (Cross-Site Request Forgery) este dezactivată, întrucât aplicația nu utilizează token-uri CSRF, iar opțiunile de frame sunt permise prin dezactivarea restricțiilor implicite. Autorizarea se realizează în funcție de rolurile utilizatorilor: de exemplu, doar utilizatorii cu rolul ADMIN și CADRU_DIDACTIC pot accesa rutele care încep cu `/portofoliu/sign-cadru/`, în timp ce STUDENT și TUTORE au acces doar la propriile rute specifice. Orice altă cerere necesită autentificare. Implementarea configurării de securitate este prezentată în Figura 8, unde este detaliată metoda `securityFilterChain`.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf(AbstractHttpConfigurer::disable)
        .headers(HeadersConfigurer<HttpSecurity> headers -> headers.frameOptions(HeadersConfigurer.FrameOptionsConfig::disable))
        .authorizeHttpRequests(AuthorizationManagerRequestMatcherConfigurer::auth -> auth
            .requestMatchers("/portofoliu/sign-cadru/**").hasRole("ADMIN")
            .requestMatchers("/portofoliu/sign-student/**").hasRole("STUDENT")
            .requestMatchers("/portofoliu/sign-tutore/**").hasRole("TUTORE")
            .requestMatchers("/portofoliu/sign-cadru/").hasRole("CADRU_DIDACTIC")
            .anyRequest().authenticated()
        )
        .formLogin(FormLoginConfigurer<HttpSecurity> form -> form
            .loginPage("/login")
            .usernameParameter("username")
            .passwordParameter("password")
            .successHandler(successHandler)
            .permitAll()
        )
        .logout(LogoutConfigurer<HttpSecurity> logout -> logout
            .logoutUrl("/logout")
            .logoutSuccessUrl("/login?logout")
            .permitAll()
        )
        .httpBasic(withDefaults());

    return http.build();
}
```

Figura 8 – Implementarea metodei `SecurityFilterChain` în clasa `SecurityConfig`

Autentificarea se face printr-un formular personalizat, cu pagina de login definită la ruta `/login`. Deși Spring Security utilizează în mod implicit un câmp `username` pentru autentificare, în această aplicație s-a realizat o adaptare astfel încât câmpul `username` să reprezinte, de fapt, adresa de e-mail a utilizatorului. Această funcționalitate a fost implementată explicit în componenta `UserDetailsServiceImpl`, care încarcă utilizatorul din baza de date pe baza adresei de e-mail și nu a unui nume de utilizator standard. În Figura 9 este prezentată logica de redirectionare după autentificare, în funcție de rolul utilizatorului.

```
@Override no usages ± Christi
public void onAuthenticationSuccess(HttpServletRequest request,
                                   HttpServletResponse response,
                                   Authentication authentication) throws IOException {

    String username = authentication.getName();
    User user = userRepository.findByEmail(username)
        .orElseThrow(() -> new UsernameNotFoundException("User not found: " + username));

    if (authentication.getAuthorities().contains(new SimpleGrantedAuthority( role: "ROLE_STUDENT")))) {
        response.sendRedirect( s: "/student/" + user.getId() + "/"index");
    } else if (authentication.getAuthorities().contains(new SimpleGrantedAuthority( role: "ROLE_TUTORE")))) {
        response.sendRedirect( s: "/tutore/" + user.getId() + "/"index");
    } else if (authentication.getAuthorities().contains(new SimpleGrantedAuthority( role: "ROLE_CADRU_DIDACTIC")))) {
        response.sendRedirect( s: "/cadru/" + user.getId() + "/"index");
    } else if (authentication.getAuthorities().contains(new SimpleGrantedAuthority( role: "ROLE_ADMIN")))) {
        response.sendRedirect( s: "/");
    } else {
        response.sendRedirect( s: "/access-denied");
    }
}
```

Figura 9 -Implementarea metodei onAuthenticationSuccess din clasa CustomAuthenticationSuccessHandler

Pentru autentificare, aplicația utilizează un AuthenticationProvider de tip DaoAuthenticationProvider, configurat prin metoda authenticationProvider(). Acesta apelează UserDetailsServiceImpl(care implementează interfața UserDetailsService) pentru a încărca utilizatorii din baza de date și utilizează algoritmul BCrypt pentru criptarea și compararea parolilor, configurat în metoda passwordEncoder(). Astfel, parolele sunt stocate în siguranță și verificate în mod securizat la autentificare. Încărcarea utilizatorului pe baza adresei de email este explicată în Figura 10.

```
@Override no usages ± Christi
public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
    User user = userRepository.findByEmail(email)
        .orElseThrow(() -> new UsernameNotFoundException("User not found: " + email));

    return org.springframework.security.core.userdetails.User
        .withUsername(user.getEmail())
        .password(user.getPassword())
        .roles(user.getRole().name())
        .build();
}
```

Figura 10 – Implementarea metodei loadUserByUsername din clasa UserDetailsServiceImpl

După autentificare, comportamentul aplicației este personalizat prin intermediul clasei CustomAuthenticationSuccessHandler, care redirecționează automat utilizatorii către pagina corespunzătoare rolului lor. Procesul de delogare este gestionat prin ruta /logout, care, la finalizare, redirecționează utilizatorul către pagina de login cu un mesaj de confirmare (/login?logout).

În completare, metoda `authenticationManager()` expune un `AuthenticationManager` global, necesar pentru procesarea autentificărilor și integrarea acestora în mecanismul de securitate al aplicației.

Prin intermediul acestor componente, aplicația implementează un sistem de autentificare și autorizare robust, adaptat fiecărui rol, cu protecții și comportamente bine definite pentru fiecare scenariu posibil.

Clasa `LoginController` este responsabilă pentru gestionarea proceselor legate de autentificare, schimbarea parolei și redirecționarea utilizatorilor către pagina corespunzătoare rolului lor. Aceasta se ocupă cu afișarea paginii de login, cu verificarea utilizatorului curent în funcție de rol, precum și cu oferirea unui formular de schimbare a parolei. Toate metodele sale sunt adnotate cu `@GetMapping` sau `@PostMapping`, ceea ce indică faptul că sunt apelate ca răspuns la cereri HTTP standard.

Metoda `showLoginPage()` este mapată pe ruta `/login` și returnează pagina de autentificare `login.html`. Aceasta este apelată automat atunci când un utilizator neautentificat accesează o resursă protejată, conform configurației definite în `SecurityConfig`.

Funcționalitatea de schimbare a parolei este realizată printr-un mecanism compus din două metode: una pentru afișarea formularului și alta pentru procesarea cererii.

Metoda `showChangePasswordPage(String role, int id, Model model)` este apelată printr-un request de tip GET și se ocupă cu afișarea formularului de schimbare parolă, personalizat în funcție de rolul utilizatorului (student, tutore sau cadru didactic). În funcție de parametrul `{role}`, sunt extrase datele utilizatorului corespunzător din baza de date și adăugate în model, care va fi utilizat pentru a popula dinamic pagina `change-password.html`.

Metoda `changePassword(String role, int id, String newPassword, RedirectAttributes redirectAttributes)` este apelată prin POST, în momentul în care utilizatorul trimite noua parolă prin formular. Aceasta identifică utilizatorul în baza de date, criptează noua parolă folosind `PasswordEncoder` și actualizează înregistrarea în baza de date. După schimbare, utilizatorul este redirecționat automat către logout, fiind forțat să se reconecteze cu parola nouă.

Accesul la interfețele principale ale fiecărui rol este gestionat prin metode dedicate. Metoda `studentIndex(int id, Authentication auth, Model model)` verifică dacă utilizatorul autentificat este același cu cel indicat prin id în URL. Dacă verificarea este validă, sunt extrase datele studentului și încărcate în model, returnând pagina `student-index.html`. În caz contrar, utilizatorul este redirecționat către o pagină de acces restricționat. Similar există metode pentru fiecare rol: `cadruDidacticIndex(int id, Authentication auth, Model model)` și `tutoreIndex(int id, Authentication auth, Model model)`. Redirecționarea studentului către pagina principală este ilustrată în Figura 11.


```
@GetMapping("/student/{id}/index") @Christi
@PreAuthorize("hasRole('STUDENT')")
public String studentIndex(@PathVariable int id, Authentication auth, Model model) {
    String username = auth.getName();
    User user = userRepository.findByEmail(username)
        .orElseThrow(() -> new UsernameNotFoundException("User not found: " + username));

    if (user.getId() != id) {
        return "redirect:/access-denied";
    }

    Student student = studentRepository.findById(id);
    model.addAttribute("student", student);
    return "student-index";
}
```

Figura 11 - Implementarea metodei studentIndex din clasa LoginController

În final, metoda `accessDenied(String error, Model model)` oferă o interfață personalizată pentru afișarea mesajelor de eroare atunci când un utilizator încearcă să acceseze o resursă fără permisiunile corespunzătoare. Poate prelua un mesaj de eroare opțional din query string, care este afișat în pagină pentru claritate.

Prin aceste metode, `LoginController` asigură un control detaliat și sigur asupra fluxurilor de autentificare, autorizare, schimbare parolă și acces în aplicație, contribuind semnificativ la experiența personalizată a fiecărui utilizator și la securitatea întregului sistem.

În cadrul aplicației, datele utilizatorilor și informațiile despre portofolii sunt modelate prin clase de entitate marcate cu adnotarea `@Entity`, specifică JPA (Java Persistence API). Aceste clase sunt persistate automat în baza de date prin Hibernate. Structura aplicată permite o organizare clară și extensibilă a modelului de date, în care toate tipurile de utilizatori moștenesc dintr-o clasă abstractă numită `User`. Această clasă de bază definește câmpurile fundamentale comune tuturor actorilor din sistem, precum identificatorul (`id`), adresa de email, parola și rolul (`Role`), și este mapată cu strategia de moștenire `JOINED`. Prin această strategie, fiecare subclasă – cum sunt `Student`, `CadruDidactic` sau `Tutore` – are propriul său tabel în baza de date, dar toate sunt legate între ele printr-o cheie comună. Clasa de bază pentru toți utilizatorii este implementată conform Figura 12.

```
@Entity 4 inheritors  ± Christi *
@Inheritance(strategy = InheritanceType.JOINED)
@Getter
@Setter
@NoArgsConstructor
public abstract class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String email;

    private String password;

    @Enumerated(EnumType.STRING)
    private Role role;

    public User(String email, String password, Role role) {
        this.email = email;
        this.role = role;
        this.password = password;
    }
}
```

Figura 12 - Implementarea clasei abstracte User

Clasa Admin, spre exemplu, este definită ca o extensie a entității CadruDidactic, fără a adăuga câmpuri suplimentare. Clasa Portofoliu este modelată ca o entitate separată, independentă de User, dar conectată logic la un student, un tutor și un cadru didactic prin relații de tip @ManyToOne, reflectând structura reală a documentelor de practică.

Pentru a evita scrierea repetitivă a metodelor de tip getter, setter sau constructori, proiectul utilizează biblioteca Lombok. Aceasta este folosită în toate clasele de entitate prin adnotări precum @Getter, @Setter și @NoArgsConstructor, care generează automat metodele esențiale necesare manipulării datelor și instanțierii obiectelor. În plus față de simplificarea accesului la câmpuri și a instanțierii claselor, biblioteca Lombok oferă și suport pentru design pattern-uri consacrate. Un exemplu relevant este adnotarea @Builder, care implementează automat pattern-ul Builder.[1][3][4] Acest șablon de proiectare permite construirea pas cu pas a unui obiect complex, oferind o alternativă elegantă la constructorii cu mulți parametri. În loc să fie nevoie de un constructor standard cu o semnătură lungă și greu de urmărit, pattern-ul Builder permite apeluri de forma Student.builder().email("...").nume("...").build(), crescând claritatea codului și reducând riscul erorilor legate de ordinea parametrilor. În contextul aplicației, utilizarea acestui model este deosebit de utilă în etapele de inițializare a bazei de date, dar și în testare sau generarea dinamică a obiectelor în funcție de datele primite din interfață. Astfel, introducerea pattern-ului Builder prin Lombok contribuie la menținerea unui cod clar, extensibil și robust, în conformitate cu bunele practici din programarea orientată pe obiect.[6][21]

Clasele repository din cadrul aplicației sunt responsabile pentru realizarea operațiilor de acces la baza de date, cum ar fi salvarea, căutarea, actualizarea și ștergerea entităților. Acestea extind interfața JpaRepository oferită de Spring Data JPA, care asigură automat implementarea metodelor CRUD de bază. În plus față de metodele implicite, fiecare repository poate defini metode personalizate, denumite conform convențiilor Spring (ex:

findByEmail, findById), care sunt interpretate automat și transpuse în interogări SQL corespunzătoare. Astfel, interacțiunea cu baza de date este simplificată semnificativ, fără a fi necesară scrierea explicită a interogărilor. De exemplu, StudentRepository oferă metode pentru extragerea studenților după ID sau adresă de email, în timp ce PortofoliuRepository permite căutarea portofoliilor asociate cu un anumit student. Această abordare contribuie la separarea clară a logicii de acces la date față de celelalte componente ale aplicației și respectă principiile arhitecturii de tip repository. În Figura 13 este prezentată logica specifică din TutoreRepository

```
@Repository
public interface TutoreRepository extends JpaRepository<Tutore,Integer>
{
    Tutore findById(int id);
    Tutore findByEmail(String email);
}
```

Figura 13 - Implementarea metodelor personalizate din TutoreRepository

Controllerul dedicat studenților are rolul de a gestiona întregul flux de operații asociate entității Student, atât din perspectiva administratorului, cât și din perspectiva studentului autentificat. Acesta este implementat în cadrul clasei StudentController și definește un set extins de rute care acoperă crearea, modificarea, ștergerea și interacțiunea studenților cu portofoliile proprii.

O funcționalitate esențială a controllerului este crearea unui cont de student, care presupune afișarea unui formular (GET /student-create) și procesarea datelor completate în acesta (POST /student-create-save). În cadrul acestei metode, datele sunt validate, iar studentul este salvat în baza de date. Ulterior, se generează automat directorul unde se va introduce semnătura digitală asociată contului, iar datele de autentificare sunt exportate în fișierul CSV dedicat, utilizat ulterior de administrator pentru logarea inițială a utilizatorului. Această metodă de a adăuga studenți este folosită ca o extensie la adăugarea lor prin fișierul CSV, administratorul putând adăuga și manual. Fluxul prin care administratorul adaugă studenți prin fișierul CSV este prezentat în Figura 3.

Controllerul oferă și funcționalități pentru modificarea datelor personale ale unui student, prin metoda GET /student-edit/{id}(pentru afișarea formularului de editare a datelor) urmată de POST /student-update/{id}(pentru modificarea propriu zisă). Pentru adăugarea sau modificarea semnăturii, există o metodă separată, POST /student/{id}/upload-signature, care primește ca parametru un fișier multipart, validează existența directorului studentului pentru semnătură și fișierul propriu zis și salvează semnătura.[19] Metoda de adăugare a semnăturii este descrisă în Figura 14.

```
@PostMapping(value = "/student/{id}/upload-signature", consumes = {"multipart/form-data"})
public String uploadSignature(@PathVariable int id,
                             @RequestParam("signature") MultipartFile file,
                             RedirectAttributes redirectAttributes) {

    try {
        String baseDir = "semnaturi/studenti/";
        Path studentDir = Paths.get(baseDir, ...more: "student" + id);
        if (!Files.exists(studentDir)) {
            System.out.println("Eroare la incarcarea semnaturii");
            redirectAttributes.addFlashAttribute( attributeName: "error", attributeValue: "Folderul studentului nu există!");
            return "redirect:/student/"+id+"/index";
        }

        Path filePath = studentDir.resolve( other: "signature.png");
        Files.write(filePath, file.getBytes());
        System.out.println("Semnătura a fost salvată la: " + filePath.toRealPath());

        redirectAttributes.addFlashAttribute( attributeName: "success", attributeValue: "Semnătura a fost actualizată cu succes!");
    } catch (IOException e) {
        redirectAttributes.addFlashAttribute( attributeName: "error", attributeValue: "Eroare la salvarea semnaturii!");
    }

    return "redirect:/student/"+id+"/index";
}
```

Figura 14 – Implementarea metodei uploadSignature din clasa StudentController

Ștergerea unui student este gestionată prin ruta GET /student-delete/{id}. Înainte de eliminarea efectivă a contului din baza de date, sunt șterse toate portofoliile asociate acestuia prin metoda deleteStudentsPortofolios(int id) din StudentService (care parcurge portofoliile și le șterge pe cele legate cu studentul ce va fi șters, prezentată în Figura 17), iar directorul corespunzător semnăturii este de asemenea eliminat pentru a evita păstrarea de fișiere goale.

Pentru interacțiunea directă cu portofoliile proprii, studentul poate accesa mai multe funcționalități:

19. semnarea unui portofoliu (POST /student/{sid}/sign/portofoliu/{pid})
20. vizualizarea unui portofoliu (GET /student/{sid}/portofoliu-view/{pid})
21. ștergerea acestuia (GET /student/{sid}/portofoliu-delete/{id})

În procesul de semnare, controllerul verifică dacă semnătura digitală există deja în directorul utilizatorului și marchează portofoliul ca fiind semnat doar dacă toate condițiile sunt îndeplinite. În caz contrar, utilizatorul este informat printr-un mesaj de eroare. Metoda este descrisă în Figura 15.

```
@PostMapping("/{sid}/sign/portofoliu/{pid}")
public String signStudent(@PathVariable int sid, @PathVariable int pid, RedirectAttributes redirectAttributes) {
    Portofoliu portofoliu = portofoliuRepository.findById(pid);
    if (portofoliu == null) {
        redirectAttributes.addFlashAttribute( attributeName: "error", attributeValue: "Portofoliul nu a fost găsit.");
        return "redirect:/portofoliu-read";
    }

    String signaturePath = portofoliu.getStudent().getSemnatura() + "/signature.png";
    File signatureFile = new File(signaturePath);

    if (!signatureFile.exists()) {
        redirectAttributes.addFlashAttribute( attributeName: "signStudentError", pid);
        return "redirect:/student-portofoliu-read/" + sid;
    }

    portofoliu.setSemnaturaStudent(true);
    portofoliuRepository.save(portofoliu);
    redirectAttributes.addFlashAttribute( attributeName: "success", attributeValue: "Semnătura studentului a fost înregistrată.");
    return "redirect:/student-portofoliu-read/" + sid;
}
```

Figura 15 – Implementarea metodei signStudent din clasa StudentController

Ștergerea portofoliului este descrisă în Figura 16. Parametrii @PathVariable reprezintă id-ul studentului(sid) respectiv portofoliului(id). Pentru a evita ștergerea în cascadă a entităților legate de portofoliu, acestea se setează cu null înainte de a fi șters portofoliul propriu-zis.

```
@GetMapping("/{sid}/portofoliu-delete/{id}")
public String portofoliuDelete(@PathVariable("id") int id, @PathVariable("sid") int sid, RedirectAttributes redirectAttributes)
{
    Portofoliu portofoliu = portofoliuRepository.findById(id);
    if (portofoliu == null) {
        redirectAttributes.addFlashAttribute( attributeName: "errorMessage", attributeValue: "Portofoliul nu a fost găsit.");
        return "redirect:/student-portofoliu-read/" + sid;
    }
    portofoliu.setStudent(null);
    portofoliu.setTutore(null);
    portofoliu.setCadruDidactic(null);
    portofoliuRepository.delete(portofoliu);
    return "redirect:/student-portofoliu-read/" + sid;
}
```

Figura 16 – Implementarea metodei portofoliuDelete din clasa StudentController

Controllerul pentru studenți joacă astfel un rol central în funcționarea aplicației, asigurând o interfață coerentă între logica de business și interacțiunea utilizatorilor cu datele proprii. Acesta combină operațiile CRUD tradiționale cu funcționalități specifice domeniului aplicației, precum semnarea electronică și gestionarea portofoliilor digitale.

Figura 17 prezintă implementarea metodei deleteStudentsPortfolios. În aceasta se parcurg toate portofoliile și se șterg atunci când se găsește un student asociat. De aici se poate deduce și comportamentul de compoziție prezentat în Figura 7 - nu poate exista un portofoliu fără un student.

```
public void deleteStudentsPortofolios(int id){  
    List<Portofoliu> portofolios = portofoliuRepository.findAll();  
    for (Portofoliu portofolio : portofolios) {  
        if (portofolio.getStudent().getId() == id) {  
            portofoliuRepository.delete(portofolio);  
        }  
    }  
}
```

Figura 17 – Implementarea metodei deleteStudentsPortofolios din clasa StudentService

Controllerul TutoreController este componenta din backend care gestionează acțiunile specifice utilizatorilor cu rolul de tutor de practică – persoane din cadrul firmelor partenere care supervizează activitatea studenților pe parcursul stagiului. Acest controller oferă atât funcționalități administrative, precum înregistrarea și editarea datelor personale ale tutorilor, cât și acțiuni directe asupra portofoliilor studenților.

Tutorele poate fi adăugat automat, atunci când un student introduce pentru prima dată adresa de email a tutorelui în formularul de creare a portofoliului. Datele sunt validate, iar tutorele este salvat în baza de date. Ulterior, este generat automat un director pentru semnătura digitală.

O funcționalitate importantă expusă de controller este posibilitatea tutorilor de a-și edita propriile date personale, inclusiv numele, funcția și telefonul. Datele sunt validate și actualizate în mod securizat în baza de date. În Figura 5 este prezentat și fluxul prin care un tutore își editează profilul, dar și alte date separate precum parola și semnătura. Pentru afișarea formularului de editare a datelor se folosește o metodă GET /tutore-edit{id} iar pentru salvarea modificărilor se folosește metoda POST /tutore-update/{id}, prezentate în Figura 18.

```
@GetMapping("/{tutore-edit/{id}}")
public String edit(@PathVariable("id") int id, Model model)
{
    Tutore tutore = tutoreRepository.findById(id);

    model.addAttribute("tutore", tutore);
    return "tutore-update";
}

@PostMapping("/{tutore-update/{id}}")
public String update(@PathVariable("id") int id, @Validated Tutore tutore, BindingResult result)
{
    if(result.hasErrors())
    {
        tutore.setId(id);
        return "tutore-update";
    }

    Tutore existingTutore = tutoreRepository.findById(id);

    tutore.setPassword(existingTutore.getPassword());
    tutore.setSemnatura(existingTutore.getSemnatura());
    tutore.setRole(existingTutore.getRole());

    tutoreRepository.save(tutore);
    return "redirect:/tutore/" + tutore.getId() + "/index";
}
```

Figura 18 – Implementarea metodelor edit și update din clasa TutoreController

De asemenea, tutorele poate încărca o semnătură digitală personalizată, sub forma unui fișier imagine (.png). Controllerul validează existența folder-ului dedicat tutorului, salvează fișierul încărcat și confirmă succesul operației printr-un mesaj transmis în interfață. Implementarea este similară pentru fiecare actor și descrisă în Figura 15 pentru student.

În ceea ce privește portofoliile, controllerul oferă tutorilor posibilitatea de a vizualiza portofoliile atribuite și de a le semna digital. Vizualizarea portofoliului este implementată similar pentru toți actorii. Pentru tutore, metoda care se ocupă cu vizualizarea portofoliului (GET tutore/{tid}/portofoliu-view/{pid}) primește ca parametrii @PathVariable id-ul tutorelui (tid) și id-ul portofoliului (pid) și este prezentată în Figura 19. Semnarea este permisă doar dacă fișierul semnăturii este disponibil și valid. După semnare, portofoliul este marcat ca semnat de tutore. Implementarea este similară cu cea din StudentController, descrisă în Figura 15.

```
@GetMapping(value = "/tutore/{tid}/portofoliu-view/{pid}")
public String viewPortofoliu(@PathVariable("tid") int tid, @PathVariable("pid") int pid, Model model) {
    Portofoliu portofoliu = portofoliuRepository.findById(pid);

    if (portofoliu == null) {
        model.addAttribute("errorMessage", "Portofoliul nu a fost găsit.");
        return "redirect:/cadru-portofoliu-read/" + tid;
    }

    Tutore tutore = portofoliu.getTutore();

    model.addAttribute("portofoliuId", pid);
    model.addAttribute("tutore", tutore);

    return "tutore-vizualizare-portofoliu";
}
```

Figura 19 – Implementarea metodei viewPortofoliu din clasa Tutore Controller

Pentru ștergerea tutorelui s-a implementat o metodă GET /tutore-delete/{id} care șterge întâi folder-ul pentru semnătura tutorelui împreună cu ea dacă există. Apoi se apelează metoda removeTutoreFromPortofolios(int id) din TutoreService care setează pe null tutorele care urmează șters (Figura 20). Aceasta elimină legătura tutorelui cu toate portofoliile asociate.

Astfel, TutoreController joacă un rol esențial în fluxul de verificare și aprobare a portofoliilor, permițând tutorilor să participe activ în procesul de validare a stagiilor de practică, într-un mod digitalizat, securizat și eficient.

```
public void removeTutoreFromPortofolios(int id) {
    List<Portofoliu> portofolii = portofoliuRepository.findAll();
    for (Portofoliu portofoliu : portofolii) {
        Tutore tutore = portofoliu.getTutore();
        if (tutore != null && tutore.getId() == id) {
            portofoliu.setTutore(null);
            portofoliuRepository.save(portofoliu);
        }
    }
}
```

Figura 20 – Implementarea metodei removeTutoreFromPortofolios din clasa TutoreService

Componenta CadruDidacticController are rolul de a centraliza și gestiona operațiunile disponibile pentru cadrele didactice implicate în evaluarea și supervizarea stagiilor de practică. Acest controller permite interacțiunea cu datele personale ale cadrului didactic, gestionarea conturilor și, mai ales, coordonarea și validarea portofoliilor atribuite studenților.

Cadrele didactice pot accesa un dashboard personal, de unde pot vizualiza și gestiona portofoliile repartizate. Controllerul permite vizualizarea unui portofoliu, dar și semnarea digitală a acestuia, în cazul în care toate condițiile sunt îndeplinite (inclusiv existența semnăturii salvate anterior). Metoda de vizualizare este asemănătoare cu cele pentru ceilalți actori. În Figura 19 este prezentată metoda echivalentă din TutorController. Și pentru semnarea portofoliului metoda este similară, cu mențiunea că, odată cu semnarea de către cadrul didactic, se setează și data semnării cu data zilei respective. Implementarea metodei POST /cadruDidactic/sign/portofoliu/{id} este prezentată în Figura 21, împreună cu o metodă redirectTo(Integer cid) care se ocupă cu redirecționarea către interfața cadrului didactic sau a prodecanului. Parametrul @RequestParam(value = "cid", required = false) Integer cid este utilizat pentru a prelua opțional identificatorul cadrului didactic. Acesta este necesar atunci când un cadru didactic accesează resursa, însă poate lipsi în cazul în care accesul se face de către un utilizator cu rol de administrator.

```
@PostMapping("@v\"cadruDidactic/sign/portofoliu/{id}\"")
public String signCadru(@PathVariable int id,
                        @RequestParam(value = "cid", required = false) Integer cid,
                        RedirectAttributes redirectAttributes) {
    Portofoliu portofoliu = portofoliuRepository.findById(id);
    if (portofoliu == null) {
        redirectAttributes.addFlashAttribute( attributeName: "error", attributeValue: "Portofoliul nu a fost găsit.");
        return redirectTo(cid);
    }

    String signaturePath = portofoliu.getCadruDidactic().getSemnatura() + "/signature.png";
    File signatureFile = new File(signaturePath);

    if (!signatureFile.exists()) {
        redirectAttributes.addFlashAttribute( attributeName: "signCadruError", id);
        return redirectTo(cid);
    }
    portofoliu.setDataSemnarii(Date.valueOf(LocalDate.now()));
    portofoliu.setSemnaturaCadruDidactic(true);

    portofoliuRepository.save(portofoliu);
    redirectAttributes.addFlashAttribute( attributeName: "success", attributeValue: "Semnătura cadrului didactic a fost înregistrată.");
    return redirectTo(cid);
}

private String redirectTo(Integer cid) {
    if (cid != null) {
        return "redirect:/cadru-portofoliu-read/" + cid;
    }
    return "redirect:/portofoliu-read";
}
```

Figura 21 – Implementarea metodelor signCadru și redirectTo din clasa CadruDidacticController

Cadrele didactice sunt create automat de sistem prin metoda adnotată cu @Beam din clasa DataInitializer, numită initUsers. Un cadru didactic poate fi adăugat și manual de către administrator, similar ca în StudentController. Există două endpoint-uri, GET /cadruDidactic-create (pentru afișarea formularului de creare a cadrului didactic) și POST /cadruDidactic-create-save (pentru salvarea propriu-zisă a cadrului didactic în baza de date). În Figura 22 sunt prezentate aceste metode, iar în Figura 23 este prezentată metoda makeSign din CadruDidacticService. Aceasta este apelată pentru a crea directorul pentru semnătura cadrului didactic și pentru setarea parolei.


```
@GetMapping("/{cadruDidactic-create}")
public String create( Model model)
{
    model.addAttribute( attributeName: "cadruDidactic", new CadruDidactic());
    return "cadruDidactic-create";
}

@PostMapping("/{cadruDidactic-create-save}")
public String createSave(@Validated CadruDidactic cadruDidactic, BindingResult result)
{
    if(result.hasErrors())
    {
        return "cadruDidactic-create";
    }
    cadruDidacticService.makeSign(cadruDidacticRepository.save(cadruDidactic));

    return "redirect:/cadruDidactic-read";
}
```

Figura 22 – Implementarea metodelor create și createSave din clasa CadruDidacticController

```
public void makeSign(CadruDidactic cadruDidactic) {
    String baseDir = "semnatura/cadreDidactice/cadruDidactic" + cadruDidactic.getId();
    File cadruDidacticDir = new File(baseDir);

    if (!cadruDidacticDir.exists()) {
        boolean created = cadruDidacticDir.mkdirs();
        if (created) {
            System.out.println("Directorul 'cadreDidactice/cadruDidactic' + cadruDidactic.getId() + " a fost creat.");
        } else {
            System.out.println("Eroare la crearea directorului 'cadreDidactice/cadruDidactic' + cadruDidactic.getId() + ".");
        }
    }

    cadruDidactic.setSemnatura(baseDir);
    cadruDidactic.setPassword(passwordEncoder.encode( rawPassword: "profesor"+cadruDidactic.getId()));
    cadruDidacticRepository.save(cadruDidactic);
}
```

Figura 23 – Implementarea metodei makeSign din clasa CadruDidacticService

În plus, CadruDidacticController oferă metode pentru editarea datelor de profil, precum numele, titulatura și datele de contact. Aceste date sunt validate și actualizate în siguranță, păstrând informațiile critice, cum ar fi semnătura electronică și rolul în sistem. Implementarea este similară cu cea pentru ceilalți actori, printr-o metodă de vizualizare a formularului de editare a datelor personale și o metodă pentru editarea propriu-zisă a datelor. În Figura 18 sunt prezentate metodele echivalente pentru tutore.

De asemenea, controllerul permite încărcarea unei semnături personalizate, utilizată ulterior pentru aprobarea oficială a portofoliilor. Fișierul încărcat este verificat și salvat într-un director dedicat fiecărui cadru didactic. Implementarea este similară cu cea pentru ceilalți actori, iar Figura 14 prezintă implementarea echivalentă pentru student.

Ștergerea unui cadru didactic se face similar cu ștergerea unui tutore, fiind implementată o metodă în CadruDidacticService, removeProfFromPortfolios(int id), care setează pe null cadrul didactic care urmează eliminat asociat cu toate portofoliile.

Controllerul joacă, astfel, un rol esențial în procesul de validare finală a stagiilor de practică, oferind cadrelor didactice unelte necesare pentru a asigura conformitatea academică și profesională a documentelor încărcate de studenți. Prin aceste funcționalități, platforma digitalizează un segment important din procesul educațional și oferă o soluție coerentă pentru validarea documentației de practică.

În cadrul controllerului administrativ (AdminController) sunt implementate două funcționalități deosebit de importante pentru eficiența și coerența procesului de gestionare a datelor.

Una dintre acestea este încărcarea unui fișier CSV care conține datele studenților, funcționalitate utilă în scenariile în care un număr mare de conturi trebuie create rapid. Prin metoda `uploadStudentCsv`, administratorul poate transmite un fișier în format `.csv`, care este copiat pe disc. Fișierul este apoi procesat de serviciul aferent (`adminService.saveStudentCsv(MultipartFile file, String UPLOAD_DIR)`), iar pentru fiecare înregistrare validă se creează un cont de student, completat cu o parolă și o semnătură digitală. Se generează și fișierul cu datele de autentificare și se salvează pe disc, de unde administratorul le poate distribui fiecărui student în parte. Eventualele erori de parsare sunt afișate în interfață, permițând corectarea rapidă a formatului. Implementarea metodei `POST /upload-students` este prezentată în Figura 24.

```
@PostMapping(value = "/upload-students", consumes = {"multipart/form-data"})
public String uploadStudentCsv(@RequestParam("file") MultipartFile file, Model model) {
    try {
        String filePath = adminService.saveStudentCsv(file, UPLOAD_DIR);
        model.addAttribute("success_message", filePath + "\n");
    } catch (IOException e) {
        model.addAttribute("success_message", "Eroare la încărcare: " + e.getMessage());
    }
    model.addAttribute("studenti", adminService.getAllStudents());
    return "student-read";
}
```

Figura 24 – Implementarea metodei `uploadStudentsCsv` din clasa `AdminController`

A doua funcționalitate notabilă este semnarea administrativă a unui portofoliu, realizată prin metoda `semneazaPortofoliu`. Aceasta permite administratorului să marcheze un portofoliu ca fiind semnat de prodecan, administratorul aplicației dar și cadru didactic în același timp. În cadrul metodei, se setează câmpul `semnaturaCadruDidactic` la valoarea `true`, se asociază prodecanul la portofoliu în cazul în care este asociat cu alt cadru didactic, iar data semnării este înregistrată automat cu ajutorul clasei `LocalDate`. Implementarea metodei se regăsește în Figura 25.

```
@PostMapping("/sign/{portofoliuId}")
public String semneazaPortofoliu(@PathVariable int portofoliuId) {
    Portofoliu portofoliu = portofoliuRepository.findById(portofoliuId);
    if(portofoliu == null){
        return "redirect:/";
    }
    Admin admin = (Admin) cadruDidacticRepository.findById(1);
    if (admin != null) {
        portofoliu.setCadruDidactic(admin);
        portofoliu.setSemnaturaCadruDidactic(true);
        portofoliu.setDataSemnarii(Date.valueOf(LocalDate.now()));
        portofoliuRepository.save(portofoliu);
    }

    return "redirect:/";
}
```

Figura 25 – Implementarea metodei semneazaPortofoliu din clasa AdminController

Una dintre cele mai importante metode din AdminService este responsabilă cu procesarea fișierelor CSV ce conțin date despre studenți. La primirea unui fișier prin interfața de administrare, metoda saveStudentCsv salvează local fișierul, îl parcurge linie cu linie și creează automat conturi de utilizator pe baza informațiilor extrase. Pentru fiecare student generat, este apelată și metoda addStudentCredentialsToCSV, care adaugă datele de autentificare (inclusiv parola temporară) într-un fișier CSV intern, utilizat pentru evidență și control. Metoda addStudentCredentialsToCSV este descrisă în Figura 26.

```
public void addStudentCredentialsToCSV(Student student) throws IOException {
    Path filePath = Paths.get( first: "upload", ...more: "Students_Credentials.csv");

    boolean fileExists = Files.exists(filePath);

    String password = new PasswordGenerator().generateRandomPassword();

    try (BufferedWriter writer = Files.newBufferedWriter(filePath, StandardOpenOption.CREATE, StandardOpenOption.APPEND)) {
        if (!fileExists) {
            writer.write( str: "Nume,Prenume,Email,Password");
            writer.newLine();
        }

        writer.write(String.join( delimiter: ",", student.getNum(), student.getPrenume(), student.getEmail(), password));
        writer.newLine();
    }
}
```

Figura 26 – Implementarea metodei addStudentCredentialsToCSV din clasa AdminService

Pe lângă funcțiile de procesare a datelor, serviciul oferă și operațiuni legate de gestiunea fișierelor și folderelor asociate utilizatorilor. De exemplu, la ștergerea unui cont,

metodele `deleteStudentFolder`, `deleteProfFolder` și `deleteTutoreFolder` elimină în mod recursiv toate fișierele aferente utilizatorului respectiv, respectiv semnătura. Acest lucru asigură curățarea completă a datelor și menținerea unui spațiu de stocare coerent. În Figura 27 este prezentată metoda `deleteTutoreFolder` care se ocupă cu ștergerea directorului pentru un tutore. Similar sunt implementate metode și pentru ceilalți actori.

```
public void deleteTutoreFolder(Tutore t) throws IOException {
    String baseTutoreDir = "src/main/resources/static/tutori";
    String folderName = baseTutoreDir + t.getId();
    Path tutoreDir = Paths.get( first: "tutori", folderName);

    if (Files.exists(tutoreDir) && Files.isDirectory(tutoreDir)) {
        try (DirectoryStream<Path> stream = Files.newDirectoryStream(tutoreDir)) {
            for (Path file : stream) {
                Files.delete(file);
            }
        }
        Files.delete(tutoreDir);
        System.out.println("Folder și conținut șters: " + tutoreDir);
    } else {
        System.out.println("Folderul nu există sau nu este director: " + tutoreDir);
    }
}
```

Figura 27 – Implementarea metodei `deleteTutoreFolder` din clasa `AdminService`

Clasa `PasswordGenerator` din pachetul `utils` este responsabilă cu generarea de parole temporare și aleatorii, utilizate în momentul în care sunt create conturi noi de utilizator (în special conturi de student, atunci când sunt importate din fișiere CSV). Parolele generate respectă o structură minimă de complexitate, fiind compuse din litere mari, litere mici și cifre, pentru a reduce riscul compromiterii conturilor. Se folosește `Stresam API` împreună cu `Random` pentru a produce o parolă aleatorie într-un mod elegant și funcțional. Algoritmul utilizează un set predefinit de caractere valide și construiește șiruri de o lungime specificată (de regulă 8 caractere), prin selecție aleatorie. Acest proces garantează unicitatea și imprevizibilitatea parolelor, fără a fi nevoie de o interacțiune manuală din partea administratorului.

Această componentă este complet independentă de baza de date și poate fi reutilizată în orice moment în cadrul altor procese din aplicație, asigurând consistență și flexibilitate în generarea datelor de autentificare. În Figura 28 este prezentată implementarea acestei clase.

```
public class PasswordGenerator {  
    public String generateRandomPassword() {  
        String chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*";  
        Random random = new Random();  
        return random.ints( streamSize: 8, randomNumberOrigin: 0, chars.length()) IntStream  
            .mapToObj(chars::charAt) Stream<Character>  
            .map(String::valueOf) Stream<String>  
            .collect(Collectors.joining());  
    }  
}
```

Figura 28 – Implementarea clasei PdfGenerator

Clasa PdfGenerator are un rol mult mai amplu și critic, fiind responsabilă de crearea documentelor PDF care conțin portofoliile finale ale studenților. Aceste documente sunt generate automat și includ toate informațiile esențiale pentru conținutul efectiv al portofoliului.

Procesul de generare implică mai mulți pași tehnici. În primul rând, clasa colectează datele necesare din entitățile persistente (Student, Portofoliu, Tutore, CadruDidactic), apoi le formatează într-un document PDF folosind librăria iText. Sunt definite secțiuni clare cu anteturi, text și semnături. Fiecărui document i se adaugă titlul, datele utilizatorilor implicați (inclusiv emailuri și roluri), descrierea stagiului de practică și perioada în care acesta a avut loc.

Un aspect important al acestei clase este și faptul că poate include în mod automat semnăturile digitale ale părților implicate (student, tutore și cadru didactic), în cazul în care acestea au fost deja încărcate în sistem. Aceste semnături sunt preluate sub formă de fișiere imagine și plasate în poziții corespunzătoare în tabelul din documentul final.

Fișierul PDF poate fi descărcat ulterior de pe pagina web unde se generează. El este conceput conform modelului unui portofoliu din cadrul Universității Politehnica din Timișoara.

Prin această clasă, aplicația oferă posibilitatea de a produce documente profesionale într-un format standardizat, direct din platformă, fără a fi necesară nicio intervenție manuală. Această funcționalitate este esențială pentru validarea academică și arhivarea portofoliilor de practică, contribuind la digitalizarea completă a procesului.

Arhitectura backend a aplicației a fost concepută pentru a asigura o gestiune robustă, modulară și clar structurată a portofoliilor de practică ale studenților.[5] Prin utilizarea framework-ului Spring Boot, aplicația beneficiază de o separare clară între logică, acces la date și prezentare, iar funcționalitățile au fost organizate în controllere dedicate fiecărui tip de utilizator (student, tutore, cadru didactic, administrator).

Fiecare controller a fost însoțit de servicii specializate și clase utilitare care automatizează procese precum generarea documentelor PDF, procesarea fișierelor CSV, salvarea semnăturilor și crearea conturilor. Interacțiunea cu baza de date este gestionată eficient prin intermediul Spring Data JPA, iar acțiunile efectuate de utilizatori sunt reflectate în mod coerent în interfața aplicației.

Acest strat backend nu doar că oferă suport funcțional aplicației, dar contribuie esențial la siguranța și coerența datelor, facilitând totodată extinderea viitoare a platformei prin adăugarea de noi funcționalități.

5.3 PARTEA DE FRONTEND

În partea de frontend, aplicația este dezvoltată folosind HTML, CSS și framework-ul de template-uri Thymeleaf, integrat cu Spring Boot. Acest lucru permite o legătură directă între datele din backend și componentele din interfața utilizator, păstrând o arhitectură de tip MVC. Stilizarea paginilor se realizează cu ajutorul fișierelor CSS dedicate.

Interfața utilizator a aplicației este concepută cu o structură unitară, menită să ofere o experiență de navigare intuitivă și coerentă pentru toate rolurile disponibile în sistem. Majoritatea paginilor HTML împărtășesc același stil vizual și includ componente reutilizabile precum header-ul și footer-ul.

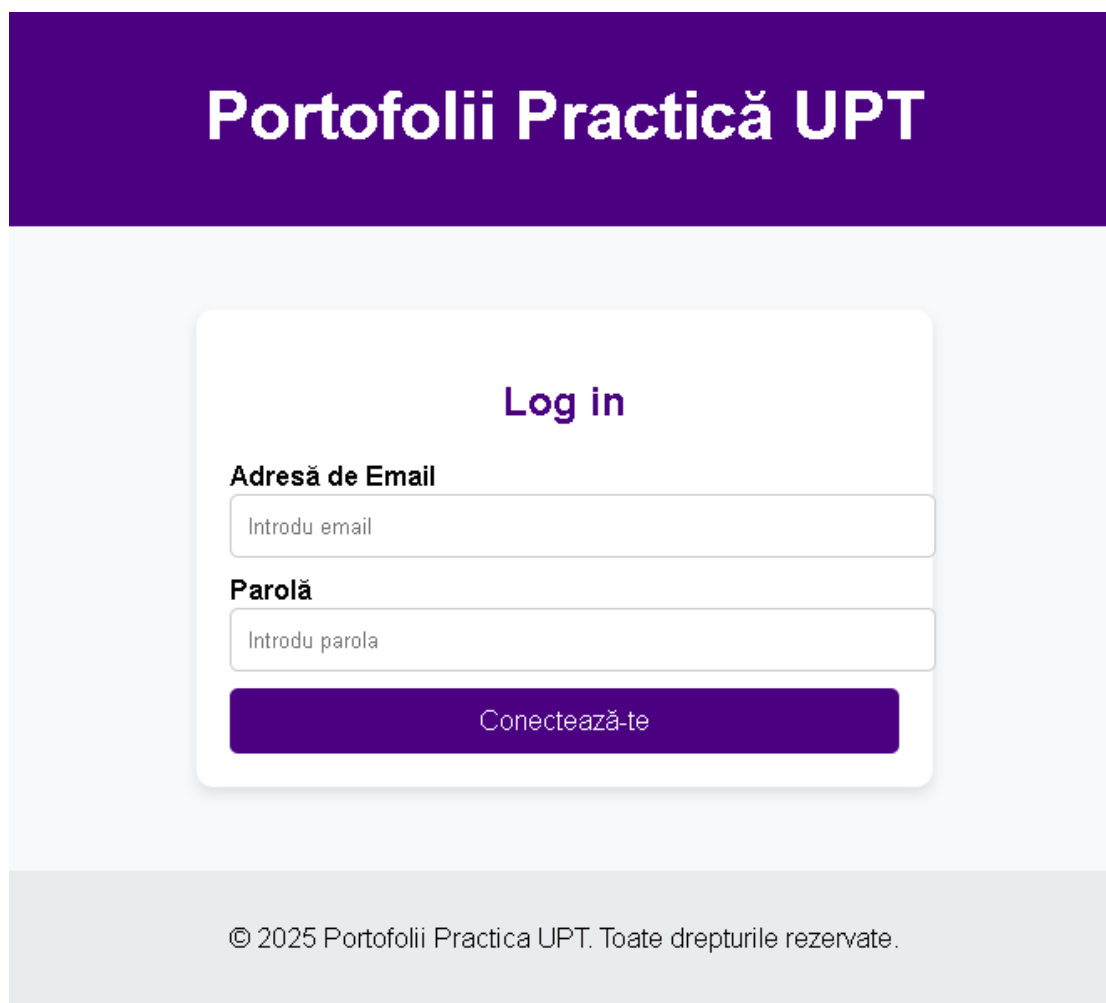
Acest cadru comun este integrat în toate interfețele aplicației, indiferent de rolul utilizatorului (student, tutore, cadru didactic sau administrator), asigurând astfel o consistență vizuală și funcțională.

Interfețele comune, accesibile tuturor categoriilor de utilizatori (student, tutore, cadru didactic și administrator), includ paginile de autentificare și de schimbare a parolei.

Pagina de login reprezintă punctul de acces inițial în aplicație. Aceasta este responsabilă pentru autentificarea utilizatorilor în funcție de adresa de email și parola setată sau primită automat.

Formularul HTML trimite o cerere POST către endpoint-ul /login, gestionat de mecanismul de securitate Spring Security. În cazul unei autentificări reușite, utilizatorul este redirecționat automat către o pagină specifică rolului său (/student/{id}/index, /tutore/{id}/index, etc.). În caz contrar, aplicația afișează un mesaj de eroare relevant.

Interfața este minimalistă și intuitivă, permițând utilizatorilor să se conecteze rapid la sistem. Ea este prezentată în Figura 29.



Portofolii Practică UPT

Log in

Adresă de Email

Parolă

Conectează-te

© 2025 Portofolii Practica UPT. Toate drepturile rezervate.

Figura 29 – Interfața paginii de login.html

După autentificare, utilizatorii au opțiunea de a-și schimba parola din interfața dedicată. Această pagină conține un formular cu un câmp pentru a introduce parola nouă. După trimiterea formularului, aplicația validează parola și actualizează parola criptat în baza de date.

Pentru securitate, actualizarea parolei este condiționată de existența unei sesiuni valide și este efectuată printr-un POST către un endpoint protejat, gestionat de backend. În cazul în care parola este schimbată cu succes, utilizatorul este notificat vizual printr-un mesaj de confirmare.

Header-ul conține de regulă logo-ul aplicației, numele platformei, butoane de navigare către secțiuni relevante, precum și o pictogramă de profil care permite accesul rapid la opțiuni precum „Schimbă parola” sau „Logout”. Footer-ul este simplu și include informații legate de aplicație sau universitate, menținând un design curat și aerisit. Atât footer-ul cât și header-ul sunt implementate ca în Figura 30 respectiv Figura 31.

```
<header class="py-3">
  <div class="container d-flex justify-content-between align-items-center">
    <h1 class="h3">Portofolii Practică UPT</h1>
    <a href="/logout" class="logout-link">Logout</a>
  </div>
</header>
```

Figura 30 – Implementarea header-ului comun

```
<footer class="text-center py-4 mt-5 bg-light">
  <p>© 2025 Portofolii Practică UPT. Toate drepturile rezervate.</p>
</footer>
```

Figura 31 – Implementarea footer-ului comun

Figura 32 – Interfața paginii change-password.html

După autentificare, utilizatorii cu rol de administrator sunt redirecționați către o interfață centrală, implementată în pagina index.html. Această interfață funcționează ca un dashboard administrativ de unde pot fi accesate toate resursele gestionabile din sistem.

Din punct de vedere vizual, pagina păstrează consistența stilistică a aplicației, utilizând același antet (header) și subsol (footer) ca și celelalte pagini. Antetul conține titlul aplicației și un buton de delogare plasat în colțul din dreapta sus. Pagina index.html este prezentată în Figura 33.

Figura 33 – Interfața paginii index.html

Conținutul principal constă într-un set de patru butoane organizate sub forma unei liste (list-group Bootstrap), fiecare dintre acestea oferind acces către o secțiune distinctă:

22. Studenți – pentru vizualizarea și gestionarea conturilor de student;
23. Cadre didactice – pentru accesul la datele profesorilor coordonatori;
24. Tutori – pentru gestionarea utilizatorilor din mediul economic;
25. Portofolii – pentru afișarea și modificarea documentelor asociate stagiilor

de practică.

Structura paginii este clară și minimalistă, oferind o navigare rapidă și intuitivă către funcționalitățile administrative esențiale. Această abordare reflectă rolul administratorului de a avea o vedere de ansamblu asupra întregului sistem și de a putea interveni asupra tuturor entităților principale.

Pagina student-read.html este accesibilă exclusiv administratorului și are rolul de a afișa lista completă a studenților existenți în baza de date. Interfața este concepută pentru a permite atât vizualizarea rapidă a utilizatorilor, cât și adăugarea acestora, fie manual, fie printr-un fișier CSV.

În cazul în care baza de date nu conține niciun student, aplicația afișează un mesaj informativ central, evidențiat vizual prin culoarea roșie: „Nu există studenți în baza de date!”. Sub acest mesaj este disponibil un formular care permite încărcarea unui fișier .csv, ce conține datele studenților ce urmează a fi înregistrați. Formularul este trimis prin metoda POST către endpoint-ul /admin/upload-students și suportă fișiere de tip multipart/form-data, cu validare automată în backend. Interfața este prezentată în Figura 34. După inserarea lor via CSV, administratorul poate doar șterge studenții sau să îi adauge manual. Interfața este prezentată în Figura 35.

Figura 34 – Interfața paginii student-read.html înainte de a insera studenții

Portofolii Practică UPTLogout

[Acasă](#) / [Studenti](#)

Lista Studenților

#	Acțiuni	Nume	Prenume	CNP	Data nașterii	Locul nașterii	Cetățenie	Serie CI	Număr CI	Adresă	An universitar	Facultate	Specializare
10	Șterge	Dragoi	Christian	1900505010203	2002-09-13	Caransebes	romana	KS	112233	Strada Mures 115, Timisoara	2024-2025	AC	cti-rc
11	Șterge	Bunea	Sergiu	1900505010203	1990-05-05	Caransebes	romana	TZ	112233	Calea Lipovei 10, Timisoara	2023-2024	AC	cti-er

File Studenti.csv was copied to upload\Students_Details.csv and generated credentials in upload\Students_Credentials.csv

◀ ▶

Adaugă un student nou

© 2025 Portofolii Practică UPT. Toate drepturile rezervate.

Figura 35 – Interfața paginii student-read.html după inserarea studenților prin fișierul CSV

Tot din această pagină, administratorul poate opta pentru adăugarea manuală a unui student, prin intermediul unui buton vizibil denumit „*Adaugă un student nou*”, care redirecționează către formularul de creare. În Figura 36 este prezentat formularul pentru inserarea manuală a unui student. La inserarea manuală, fișierul cu datele de autentificare este actualizat cu datele noului student.

Portofolii Practică UPTLogout

[Acasă](#) / Creare student

Formular Adăugare Student

Nume:

Prenume:

Data nașterii:

Email:

Telefon:

Serie CI:

Număr CI:

CNP:

Locul nașterii:

Cetățenie:

Adresa:

Figura 36 – Interfața paginii student-create.html

În cadrul interfeței de afișare a studenților, s-a utilizat mecanismul de control condițional oferit de motorul de template-uri Thymeleaf, prin expresia `th:if`, pentru a adapta conținutul afișat în funcție de starea datelor (această tehnică permite o interfață adaptivă și intuitivă, evitând afișarea unui tabel gol și oferind utilizatorului posibilitatea de a acționa imediat pentru a popula baza de date). Această tehnică permite o interfață adaptivă și intuitivă, evitând afișarea unui tabel gol și oferind utilizatorului posibilitatea de a acționa imediat pentru a popula baza de date. Implementarea mecanismului este prezentat în Figura 37.

```
<h2 class="mb-4">Lista Studenților</h2>
<div class="card shadow-sm p-4 border-0 rounded-3">
  <div class="card-body">
    <div th:if="{#lists.isEmpty(student)}">
      <h3 class="text-danger text-center">Nu există studenți în baza de date!</h3>
    <div class="card shadow-sm p-3">
      <div class="card-body">
        <h4>Încarcă fișier CSV pentru studenți</h4>
        <form action="/admin/upload-students" method="post" enctype="multipart/form-data" class="mt-3">
          <div class="mb-3">
            <label for="csvFile" class="form-label">Selectează fișier CSV:</label>
            <input type="file" id="csvFile" name="file" accept=".csv" class="form-control" required>
          </div>
          <button type="submit" class="btn btn-primary">Trimite</button>
        </form>
      </div>
    </div>
  </div>
  <div th:if="{not #lists.isEmpty(student)}">
    <div class="table-responsive">
      <table class="table table-bordered table-striped text-center align-middle">
        <thead class="table-light">
          <tr>
            <th>#</th>
            <th>Acțiuni</th>
            <th>Nume</th>
            <th>Prenume</th>
            <th>CNP</th>
            <th>Data nașterii</th>
            <th>Locul nașterii</th>
            <th>Cetățenie</th>
            <th>Serie CI</th>
            <th>Număr CI</th>
            <th>Adresă</th>
          </tr>
        </thead>
      </table>
    </div>
  </div>
</div>
```

Figura 37 – Implementarea logicii pentru vizualizarea listei studenților în pagina student-read.html

Vizual, pagina păstrează consistența cu celelalte secțiuni ale aplicației, utilizând componenta header cu numele aplicației și butonul de „Logout”, precum și breadcrumb pentru a evidenția locația curentă în sistemul de navigație: *Acasă > Studenți*.

Această interfață joacă un rol central în procesul de inițializare și mentenanță a conturilor de student, fiind prima etapă în gestionarea portofoliilor de practică.

Pagina destinată afișării portofoliilor de practică oferă administratorului o perspectivă structurată asupra documentelor asociate studenților. Interfața este implementată sub forma unei tabele responsive, în care sunt prezentate informații relevante pentru fiecare portofoliu: identitatea studentului, tutorele, cadrul didactic, perioada de desfășurare a stagiului și statusul semnării etc.

Fiecare rând din tabel este asociat unui portofoliu și include acțiuni precum vizualizarea detaliilor sau, în funcție de drepturile utilizatorului, ștergerea acestuia. Tabelul este generat dinamic cu ajutorul motorului de template-uri Thymeleaf, fiind alimentat cu date preluate din baza de date prin controllerul specific. Administratorul poate vedea dacă ceilalți actori au semnat sau nu. Când un cadru didactic a semnat deja documentul, butonul „Semnează” este înlocuit, ca în Figura 39. De asemenea, în Figura 38 sunt prezentate acțiunile de ștergere și vizualizare a portofoliului. Similar se întâmplă și pentru ceilalți actori, doar că studentul poate și edita ulterior portofoliul, iar tutorele poate doar vizualiza documentul, neavând dreptul de a-l șterge sau edita.

Interfața urmărește principiile de claritate și eficiență vizuală, păstrând un design unitar cu restul aplicației și permițând utilizatorului să navigheze ușor printre documentele existente. În absența portofoliilor, se afișează un mesaj informativ corespunzător.

Portofolii Practică UPT Logout

[Acasă](#) / Lista Portofoliilor

Lista Portofoliilor

ID	Acțiuni	Nume Student	Tutore	Profesor Coordonator	Locul Desfășurării	Durata Practicii	Data Început	Data Sfârșit	Data Semnării	Orar
2	Vizualizare Ștergere	Pop Ana	null null	Nanu Sorin	Conti	140	2025-06-04	2025-06-27		de luni pana vine
3	Vizualizare Ștergere	Dragoi Christian	Petrescu Alina	Todinca Doru	Nokia	240	2025-06-03	2025-06-28		de luni pana vine
4	Vizualizare Ștergere	Pop Ana	Petrescu Alina	Nanu Sorin	Nokia	100	2025-06-01	2025-07-05		de luni pana vine

© 2025 Portofolii Practică UPT. Toate drepturile rezervate.

Figura 38 – Interfața portofoliu-read.html

Portofolii Practică UPT Logout

[Acasă](#) / Lista Portofoliilor

Lista Portofoliilor

	Competențe Dobândite	Mod de Pregătire	Activități Planificate	Observații	Semnătura Student	Semnătura Tutore	Semnătura Cadru Didactic
	Invatarea implementarii unui softare complex.	Training-uri	Training-uri si task-uri	fara observatii	X	X	Semnează
are	Invatarea tehnicilor de testare manuala si automata	Stagiu de practica in domeniu	Training-uri	observatii	X	X	Semnează
are	Invatarea tehnicilor de testare manuala si automata	Stagiu de practica in domeniu	Training-uri	fara observatii	✓	X	Semnează

© 2025 Portofolii Practică UPT. Toate drepturile rezervate.



Figura 39 – Interfața portofoliu-read.html cu acțiunea de semnare

Interfața pentru vizualizarea tutorilor este prezentată în Figura 40. Această interfață oferă administratorului o viziune de ansamblu asupra tuturor tutorilor existenți în platformă. Informațiile sunt afișate sub formă tabelară, fiecare rând reprezentând un utilizator de tip tutore și incluzând datele esențiale: nume, prenume, funcție, telefon, adresă de email și semnătura asociată.

Portofolii Practică UPTLogout

[Acasă](#) / Tutori

Lista Tutorilor

#	Acțiuni	Nume	Prenume	Funcție	Telefon	Email	Semnătură
9	Șterge	Petrescu	Alina	Manager	0744444444	alina.petrescu@email.com	
15	Șterge					ion@email.com	

[Adaugă un tutor nou](#)

© 2025 Portofolii Practică UPT. Toate drepturile rezervate.

Figura 40 – Interfața paginii tutor-read.html

Tabelul este generat dinamic, utilizând motorul de template-uri Thymeleaf, și oferă acțiuni contextuale precum ștergerea unui utilizator. Semnăturile sunt afișate în format imagine, fiind preluate automat din fișierele asociate fiecărui tutor. În cazul în care o semnătură nu este disponibilă, aplicația afișează un text pentru informarea utilizatorului.

Interfața respectă stilul vizual unitar al aplicației, cu un antet clar, breadcrumb pentru orientare și un buton dedicat adăugării rapide a unui nou tutor. Prin această pagină, se facilitează atât validarea vizuală a datelor existente, cât și gestionarea rapidă a conturilor.

Pagina dedicată listării cadrelor didactice permite administratorului să vizualizeze și să gestioneze rapid informațiile asociate personalului academic implicat în coordonarea portofoliilor de practică (Figura 41). Interfața este structurată sub forma unui tabel dinamic, în care sunt afișate atributele esențiale ale fiecărui cadru: nume, email, telefon, funcție, specializare și semnătura digitală.

Portofolii Practică UPT

Logout

[Acasă](#) / Cadre didactice

Lista Cadrelor Didactice

#	Acțiuni	Nume	Email	Telefon	Funcție	Specializare	Semnătură
1	Editează Șterge	Chirila Ciprian	prodecan@upt.ro	0700123456	Prodecan	cti-ro	
2	Editează Șterge	Todina Doru	doru.todina@upt.ro	0700123456	conferentiar	cti-ro	
3	Editează Șterge	Cernazanu Cosmin	cosmin.cernazanu@upt.ro	0700123456	conferentiar	cti-en	
4	Editează Șterge	Nanu Sorin	sorin.nanu@upt.ro	0700123456	sef de lucrari	is	
5	Editează Șterge	Szeidert Iosif	iosif.szeidert@upt.ro	0700123456	conferentiar	info-zi	
6	Editează Șterge	Crisan-Vida Mihaela	mihaela.crisan-vida@upt.ro	0700123456	sef de lucrari	info-id	

Adaugă un cadru didactic nou.

© 2025 Portofolii Practică UPT. Toate drepturile rezervate.

Figura 41 – Interfața paginii cadruDidactic-read.html

Fiecare rând din tabel este asociat unui cont activ și include butoane de acțiune pentru editarea datelor sau eliminarea utilizatorului din sistem. În cazul în care semnătura nu este încărcată, aplicația afișează un mesaj informativ („Semnătura cadrului didactic lipsă”), asigurând transparența asupra completitudinii profilului.

Designul vizual este coerent cu restul platformei și include elemente de navigare intuitivă, precum breadcrumb-ul și butonul „Adaugă un cadru didactic nou”. Acest formular duce către o pagină dedicată unde pot fi introduse manual datele unui nou utilizator.

Interfața contribuie la o gestionare eficientă a utilizatorilor academici, oferind în același timp un tablou clar asupra implicării fiecăruia în cadrul stagiilor de practică.

Pagina de editare a informațiilor unui cadru didactic permite actualizarea datelor personale și profesionale într-un mod centralizat și intuitiv. Interfața este împărțită în două secțiuni: formularul de editare propriu-zis și zona dedicată semnăturii electronice.

Formularul conține câmpuri pre completate, care pot fi modificate pentru a actualiza numele, prenumele, funcția deținută, specializarea, numărul de telefon și adresa de email. Similar, pentru fiecare actor, pagina de editare conține câmpurile corespunzătoare. După modificare, utilizatorul poate salva schimbările printr-un buton dedicat sau poate anula operațiunea.

În partea inferioară a paginii se regăsește funcționalitatea de încărcare a unei semnături. Cadrul didactic poate selecta un fișier din sistemul local, iar aplicația îl va salva în directorul corespunzător. Dacă o semnătură a fost deja încărcată anterior, aceasta este

afișată în zona etichetată „Semnătura curentă”. Pagina este implementată similar pentru toți actorii. Figura 42 prezintă pagina pentru editarea profilului unui cadru didactic.

Interfața contribuie la menținerea actualizată a datelor utilizatorilor academici și susține transparența asupra responsabilităților asumate prin semnătură. De asemenea, asigură o experiență de utilizare coerentă cu celelalte secțiuni ale aplicației, respectând aceeași estetică și structură vizuală.

Portofolii Practică UPT Logout

Editează Cadru Didactic

[Change Password](#)

Nume	Prenume
<input type="text" value="Todinca"/>	<input type="text" value="Doru"/>
Funcție	Specializare
<input type="text" value="conferentiar"/>	<input type="text" value="cti-ro"/>
Telefon	Email
<input type="text" value="0700123456"/>	<input type="text" value="doru.todinca@upt.ro"/>

Încarcă semnătura:

No file chosen

Semnătura curentă:

© 2025 Portofolii Practică UPT. Toate drepturile rezervate.

Figura 42 – Interfața paginii cadruDidactic-update.html

În Figura 43 este prezentată pagina de vizualizare a unui document creat de sistem. Această interfață are rolul de a oferi o prezentare completă și structurată a documentului PDF asociat unui portofoliu de practică. Pagina este accesibilă oricărui utilizator.

Portofoliu Practică

Document PDF - Portofoliu

2 / 2 | 80%

a stagiului de practică:
Testarea sistemelor software

13. Definirea competențelor care vor fi dobândite pe perioada stagiului de practică:

Competențe	Competente minime de IT si limbaje de programare(C, Java)
Modulul de pregătire	Stagiu de practica in domeniu
Activități planificate	Training-uri
Observații	observatii

14. Modalități de evaluare a pregătirii profesionale dobândite de practicant pe perioada stagiului de pregătire practică:

Evaluarea practicantului pe perioada stagiului de pregătire practică se va face de către tutore.

	Cadrul Didactic	Tutore	Student
Nume și Prenume	Todinca Doru	Petrescu Alina	Dragoi Christian
Funcția	conferentiar	Manager	Student
Data	2025-06-19	2025-06-19	2025-06-19
Semnătura			

© 2025 Portofolii Practică UPT. Toate drepturile rezervate.

Figura 43 – Interfața paginii portofoliu-vizualizare.html

Din punct de vedere tehnic, documentul PDF este încorporat într-un element iframe, folosind un th:src Thymeleaf pentru a apela dinamic ruta /portofoliu-generate-pdf/{id}. Astfel, se asigură afișarea în timp real a portofoliului actualizat, fără a fi nevoie de descărcarea manuală.

Interfața este organizată clar: antetul paginii include titlul aplicației și opțiunea de logout, iar sub acesta, breadcrumb-ul ghidează utilizatorul în ierarhia navigațională. Zona centrală este ocupată de un card Bootstrap, în care este afișat titlul secțiunii și conținutul PDF-ului, cu opțiuni implicite de derulare, zoom și paginare oferite de viewer-ul browser-ului.

Prin această soluție, aplicația facilitează validarea vizuală a semnăturilor și a completitudinii portofoliului, oferind o experiență intuitivă și profesională fără a expune fișierele local sau a necesita instrumente externe.

Partea de frontend a fost concepută pentru a oferi o interfață intuitivă și accesibilă, adaptată nevoilor diferitelor tipuri de utilizatori (studenți, cadre didactice, tutori și administratori). Utilizarea framework-ului Bootstrap a facilitat structurarea unui design coerent și modern, în timp ce integrarea motorului Thymeleaf a permis afișarea dinamică a conținutului. Rezultatul este o experiență de utilizare fluidă, care simplifică interacțiunea cu portofoliile de practică și contribuie la eficientizarea proceselor academice.

6. TESTARE

Pentru a valida funcționalitatea aplicației, a fost realizată o testare manuală completă a platformei din perspectiva tuturor rolurilor definite: student, tutore, cadru didactic și administrator. Testarea s-a desfășurat într-un mediu local, utilizând browser-ul și interfața grafică a aplicației, fără a folosi instrumente de testare automată.

Obiectivul principal a fost verificarea corectitudinii interacțiunii între componentele frontend și backend, precum și a fluxurilor funcționale esențiale: autentificare, creare/editare entități, încărcare semnături, generare de portofolii PDF și afișarea corectă a acestora în interfață.

Pentru fiecare rol au fost testate cazuri pozitive (funcționalități finalizate cu succes) și negative (erori de validare, lipsa semnăturii sau a câmpurilor obligatorii). De asemenea, a fost urmărită consistența mesajelor afișate.

În cadrul procesului de testare manuală, au fost descoperite o serie de erori care afectau funcționalitatea aplicației în anumite scenarii. Aceste probleme au fost analizate și corectate pentru a asigura comportamentul stabil și robust al platformei.

Una dintre cele mai semnificative probleme a fost observată în procesul de creare manuală a unui cont de student. Deși formularul era completat corect, parola utilizatorului nu era setată, rezultând o valoare null salvată în baza de date. În consecință, studentul nu putea accesa sistemul după creare. Această deficiență a fost remediată prin generarea unei parole randomizate folosind o clasă PasswordGenerator, urmată de criptarea acesteia cu BCryptPasswordEncoder și salvarea valorii hash în baza de date. În paralel, parola necriptată a fost scrisă într-un fișier CSV pentru a putea fi comunicată utilizatorului.

În ceea ce privește afișarea semnăturilor, au fost observate situații în care imaginea aferentă unui utilizator lipsea sau nu era încărcată corect. Acest lucru afecta vizibilitatea semnăturii în paginile de tip „read” și „update”. Pentru rezolvarea acestei probleme, s-au introdus verificări suplimentare în backend, precum și afișarea unui mesaj informativ sau a unui element vizual fallback (alt="Semnătura lipsește") în cazurile în care fișierul nu exista.

De asemenea, au fost testate diverse cazuri de redirectionări incorecte sau accesare de pagini fără permisiuni. Acestea au fost corectate prin revizuirea configurației Spring Security și a metodelor de redirect din controllere.

Corectarea acestor probleme a fost esențială pentru asigurarea unei experiențe de utilizare coerente și pentru garantarea funcționării corecte a platformei în condiții reale de utilizare.

7. CONCLUZII

Lucrarea de față a urmărit dezvoltarea unei aplicații informatice dedicate gestionării portofoliilor de practică ale studenților din cadrul Universității Politehnica Timișoara. Prin implementarea unei soluții full-stack bazate pe tehnologii moderne, s-a reușit acoperirea completă a proceselor implicate în înregistrarea, evaluarea și validarea portofoliilor de practică.

Aplicația dezvoltată oferă o interfață intuitivă și adaptată fiecărui rol implicat (student, tutore, cadru didactic, administrator), gestionând eficient fluxuri precum înscrierea studenților, adăugarea de documente și semnarea digitală a portofoliilor. Pe partea de backend, s-au implementat funcționalități robuste de generare PDF, export CSV, precum și securizare a accesului. În cadrul testării manuale, aplicația s-a dovedit funcțională, iar eventualele erori identificate (ex. inițializarea parolelor, setarea rolului utilizatorului) au fost remediate prompt.

Utilitatea practică a aplicației este evidentă în contextul în care instituțiile de învățământ superior caută soluții digitale pentru simplificarea proceselor birocratice. Platforma propusă reduce considerabil efortul administrativ, minimizează riscul de erori umane și crește gradul de transparență între părți.

Totuși, în actuala formă, aplicația are unele limitări, precum lipsa unui sistem automatizat de notificări sau imposibilitatea recuperării parolei fără intervenție administrativă.

Pentru etapele viitoare, se propune extinderea aplicației prin automatizarea transmiterii emailurilor de confirmare sau notificare, un mecanism prin care prodecanul poate trimite automat email către toți studenții cu datele de autentificare după ce sunt adăugați în baza de date, respectiv automatizarea legării unui portofoliu de un cadru didactic în funcție de specializarea studentului.

În concluzie, aplicația construită răspunde cerințelor fundamentale ale gestionării stagiilor de practică, fiind o bază solidă pentru dezvoltări ulterioare în direcția digitalizării educației universitare.

BIBLIOGRAFIE

- [1] Bloch, J., *Effective Java*, 3rd edition, Boston, Addison-Wesley, 2018.
- [2] DuBois, P., *MySQL*, 5th edition, Boston, Addison-Wesley, 2013.
- [3] Freeman, E., Freeman, E., Bates, B., Sierra, K., *Head First Design Patterns*, Sebastopol CA, O'Reilly Media, 2021.
- [4] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston, Addison-Wesley, 1994.
- [5] Martin, R., *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River NJ, Prentice Hall, 2002.
- [6] Martin, R., *Clean Code: A Handbook of Agile Software Craftsmanship*, Upper Saddle River NJ, Prentice Hall, 2008.
- [7] Riel, A., *Object-Oriented Design Heuristics*, Boston, Addison-Wesley, 1996.
- [8] Schildt, H., *Java: The Complete Reference*, 12th edition, New York, McGraw-Hill, 2022.
- [9] Walls, C., *Spring in Action*, 6th edition, Shelter Island NY, Manning, 2022.
- [10]***<https://docs.spring.io/spring-boot/docs/3.2.5/reference/htmlsingle/> (accesare iunie 2025)
- [11]***<https://maven.apache.org/guides/introduction/introduction-to-the-pom.html> (accesare iunie 2025)
- [12]*** <https://www.thymeleaf.org/documentation.html> (accesare iunie 2025)
- [13]*** <https://dev.mysql.com/doc/> (accesare iunie 2025)
- [14]***<https://www.geeksforgeeks.org/difference-between-mysql-and-postgresql/> (accesare iunie 2025)
- [15]***<https://docs.spring.io/spring-data/jpa/reference/#reference> (accesare iunie 2025)
- [16]***<https://www.baeldung.com/hibernate-inheritance> (accesare iunie 2025)
- [17]***https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html (accesare iunie 2025)
- [18]***<https://commons.apache.org/proper/commons-csv/> (accesare iunie 2025)

[19]*** <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.web.multipart/MultipartFile.html> (accesare iunie 2025)

[20]** <https://github.com/LibrePDF/OpenPDF> (accesare iunie 2025)

[21]*** <https://projectlombok.org/> (accesare iunie 2025)

Anexa 2

**DECLARAȚIE DE AUTENTICITATE A
LUCRĂRII DE FINALIZARE A STUDIILOR***

Subsemnatul DRĂGOI CHRISTIAN-ALEXANDRU
legitimat cu CI seria KS nr. 683806
CNP 5020913350061
autorul lucrării GESTIUNEA PORTOFOLIILOR DE PRACTICĂ ALE STUDENȚILOR

elaborată în vederea susținerii examenului de finalizare a studiilor de
LICENȚĂ organizat de către Facultatea
AUTOMATICĂ ȘI CALCULATOARE din cadrul Universității
Politehnica Timișoara, sesiunea Iunie 2025 a anului universitar
2024-2025, coordonator CONF. DR. ING. CIPRIAN-BOGDAN CHIRILĂ luând în
considerare prevederile Capitolului V - Măsuri privind asigurarea originalității lucrărilor din
*Regulamentul privind organizarea și desfășurarea examenelor de licență/diplomă și disertație
în Universitatea Politehnică Timișoara*, aprobat prin HS nr. .../..... și cunoscând faptul că în
cazul constatării ulterioare a unor încălcări ale normelor de etică/a faptului că diploma a fost
obținută prin mijloace frauduloase, voi suporta sancțiunile legale prevăzute de Legea nr.
199/2023 – Legea Învățământului Superior, declar pe proprie răspundere, că:

- această lucrare este rezultatul propriei activități intelectuale;
- lucrarea nu conține texte, date sau elemente de grafică din alte lucrări sau din alte surse fără ca acestea să nu fie citate, inclusiv situația în care sursa o reprezintă o altă lucrare/alte lucrări ale subsemnatului;
- sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor;
- această lucrare nu a mai fost prezentată în fața unei alte comisii de examen/prezentată public/publicată de licență/diplomă/disertație;
- În elaborarea lucrării am utilizat instrumente specifice inteligenței artificiale (IA) și anume ChatGPT (denumirea) https://chatgpt.com/ (sursa), pe care le-am citat în conținutul lucrării nu am utilizat ~~instrumente specifice inteligenței artificiale (IA)~~¹.

Declar că sunt de acord ca lucrarea să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului său într-o bază de date în acest scop.

Timișoara,

Data
25.06.2025

Semnătura



*Declarația se completează de student, se semnează olograf de acesta și se inserează în lucrarea de finalizare a studiilor, la sfârșitul lucrării, ca parte integrantă.

¹ Se va păstra una dintre variante: 1 - s-a utilizat IA și se menționează sursa 2 – nu s-a utilizat IA