UNIVERSITATEA "POLITEHNICA" din BUCUREȘTI

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Dezvoltarea unei aplicații web folosind tehnologia Spring Boot

Data predare: 15.12.2025

Student: Grecu Victor

Grupa: 433C

București 2025

CUPRINS

CAPITOLUL 1
INTRODUCERE

1.1. Context și motivație

Dezvoltarea aplicațiilor web moderne impune soluții software scalabile, sigure și ușor de mentinut, capabile să se adapteze rapid cerințelor pieței [1]. În acest peisaj tehnologic, Spring Boot (lansat în 2014) s-a impus ca un standard în ecosistemul Java, simplificând radical procesul de dezvoltare prin paradigma *"convention over configuration"* și eliminând necesitatea configurărilor XML laborioase [2].

Popularitatea framework-ului în mediul *enterprise* se bazează pe trei piloni fundamentali care reduc timpul de livrare a produsului software:

- Auto-configurare: Detectarea și configurarea automată a componentelor pe baza dependențelor din proiect [3];
- Servere Embedded: Integrarea nativă a serverelor web (Tomcat, Jetty) direct în executabil, eliminând complexitatea procesului de *deployment* extern [4];
- Spring Boot Starters: Gestionarea eficientă a dependențelor prin pachete pre-configurate [5].

Această arhitectură permite dezvoltatorilor să se concentreze preponderent asupra logicii de business, delegând responsabilitățile de infrastructură către framework.

1.2. Obiectivele lucrării

Lucrarea de față își propune dezvoltarea unei aplicații web complete pentru gestionarea datelor academice, demonstrând eficiența Spring Boot în implementarea unei arhitecturi stratificate și a operațiilor CRUD.

Obiectivele specifice urmărite sunt:

1. Analiza teoretică a ecosistemului Spring Boot și a avantajelor acestuia;
2. Proiectarea unei baze de date relaționale optimizate pentru entități academice (studenți, discipline);
3. Implementarea arhitecturii MVC (Model-View-Controller) pentru o separare strictă a responsabilităților [6];
4. Realizarea persistenței datelor prin standardul JPA, utilizând Spring Data JPA și Hibernate [7];
5. Dezvoltarea interfeței utilizator cu ajutorul motorului de template-uri Thymeleaf [8];
6. Validarea funcționalităților sistemului prin teste unitare și de integrare [9].

CAPITOLUL 2
FUNDAMENTELE TEORETICE ALE TEHNOLOGIEI SPRING BOOT

2.1. Framework-ul Spring Boot

Spring Boot revoluționează dezvoltarea în ecosistemul Java prin filozofia sa *"opinionated"*, oferind o platformă gata de producție cu configurări minime. Spre deosebire de abordarea tradițională, care implică configurări XML extinse, Spring Boot utilizează **Auto-configuration**—un mecanism care analizează *classpath*-ul și instanțiază automat bean-urile necesare (ex: configurarea automată a unui EntityManager la detectarea dependenței JPA) [10].

Arhitectura sa se bazează pe componente cheie:

- **Spring Boot Starters:** Descriptori de dependențe care agregează biblioteci compatibile pentru funcționalități specifice (de exemplu, spring-boot-starter-web include Tomcat, Spring MVC și Jackson) [11].
- **Actuator:** Modul dedicat monitorizării în producție, expunând metrici despre starea aplicației ("health checks") și performanță prin endpoint-uri HTTP.

2.2. Arhitectura Model View Controller (MVC)

Pattern-ul arhitectural MVC separă aplicația în trei componente distincte, facilitând mentenanța și testarea modulară [12]. În contextul Spring Boot, implementarea acestui pattern (Spring Web MVC) funcționează astfel:

1. **Model:** Reprezentat de clasele @Entity (structura datelor) și @Service (logica de business).
2. **View:** Stratul de prezentare, gestionat în acest proiect de Thymeleaf, responsabil de randarea interfeței utilizator.
3. **Controller:** Clasele adnotate cu @RestController sau @Controller care interceptează cererile HTTP, apelează serviciile și determină răspunsul [13].

2.3. Spring Data JPA și Hibernate

Persistența datelor este asigurată prin **JPA (Java Persistence API)**, specificația standard pentru *Object-Relational Mapping* (ORM). Aceasta permite manipularea datelor folosind obiecte Java (POJO) în loc de interogări SQL brute [14]. **Hibernate** este utilizat ca implementare a acestei specificații, oferind funcționalități avansate precum *caching*, *lazy loading* și generarea automată a schemelor de bază de date.

Spring Data JPA simplifică accesul la date prin abstractizarea **Repository Pattern**. Prin extinderea interfeței JpaRepository<T, ID>, framework-ul generează automat la runtime implementările pentru operațiile CRUD standard, eliminând necesitatea scrierii codului repetitiv [15].

De asemenea, permite derivarea interogărilor direct din numele metodelor (ex: findByEmail(String email) este tradus automat în SQL: SELECT * FROM ... WHERE email = ?).

2.4. Thymeleaf - Motor de template-uri

Thymeleaf este un motor de template-uri server-side modern, care se distinge prin conceptul de **"Natural Templates"**. Spre deosebire de JSP, fișierele Thymeleaf sunt documente HTML valide care pot fi vizualizate static în browser, facilitând colaborarea dintre developeri și designeri [16]. Integrarea cu Spring se realizează prin atribute speciale (th:text, th:each, th:if), permițând legarea dinamică a datelor din Model direct în elementele DOM.

CAPITOLUL 3
PROIECTAREA ȘI IMPLEMENTAREA APLICAȚIEI

3.1. Descrierea aplicației

Sistemul dezvoltat este o soluție de management academic care facilitează administrarea studenților, a disciplinelor și a procesului de înscriere la cursuri. Arhitectura se bazează pe **Spring Boot**, alegere care elimină complexitatea configurării unui server de aplicații extern datorită containerului Tomcat *embedded* [18]. Tehnologiile adiacente includ **MySQL** pentru persistența datelor, **Thymeleaf** pentru randarea dinamică a paginilor HTML și **Bootstrap** pentru o interfață utilizator *responsive*.

3.2. Cerințe funcționale

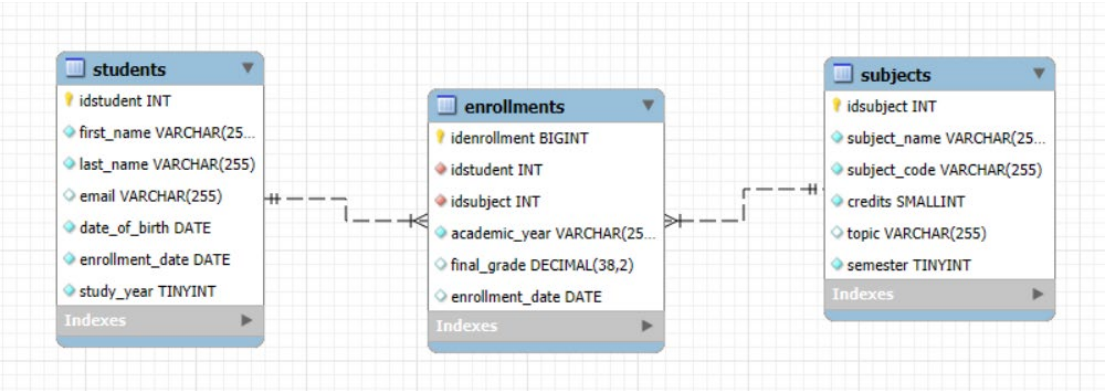
Aplicația implementează un set complet de operații CRUD (Create, Read, Update, Delete), asigurând:

- Gestionarea profilurilor studenților (înregistrare, editare, ștergere).
- Validarea datelor de intrare (input validation) pentru menținerea consistenței informațiilor [19].
- Vizualizarea listelor de entități și a relațiilor dintre acestea.

3.3. Schema relațională

Persistența este asigurată de baza de date "university", structurată pe trei tabele normalizate care modelează fluxul academic:

1. **students**: Tabelul principal, identificat prin cheia primară idstudent (AUTO_INCREMENT). Stochează datele personale și academice (anul de studiu, data înscrierii).
2. **subjects**: Definește catalogul de materii, incluzând numărul de credite și semestrul aferent.
3. **enrollments**: Tabel de tip *junction* care rezolvă relația **Many-to-Many** dintre studenți și materii. Integritatea referențială este garantată prin chei străine (student_id, subject_id) [20].



[Figura 1: Diagrama bazei de date - Relațiile dintre tabele]

3.4. Maparea Object-Relational (ORM)

Maparea dintre modelul orientat pe obiecte și baza de date relațională este asigurată prin adnotarea claselor Java cu @Entity. Spring Data JPA utilizează aceste metadate pentru a traduce automat operațiile asupra obiectelor în interogări SQL, simplificând persistența datelor.

3.5. Arhitectura aplicației

Aplicația implementează o arhitectură stratificată (*Layered Architecture*) pentru a garanta separarea responsabilităților (*Separation of Concerns*) și modularitatea sistemului.

Structura cuprinde patru niveluri interconectate:

1. Stratul de Persistență (Entity): Modelul de date reprezentat prin clase POJO.
2. Stratul de Acces la Date (Repository): Interfețe ce extind JpaRepository, eliminând codul redundant prin generarea automată a implementărilor SQL la *runtime*.
3. Stratul de Servicii (Service): Componente adnotate cu @Service ce încapsulează logica de business și gestionează dependențele prin @Autowired.
4. Stratul de Prezentare (Controller): Gestionează cererile HTTP (@GetMapping, @PostMapping) și transmite datele către *view-urile* Thymeleaf.

CAPITOLUL 4
IMPLEMENTAREA FUNCȚIONALITĂȚILOR

4.1 Conexiunea la baza de date

Gestionarea conexiunilor este optimizată prin utilizarea implicită a pool-ului HikariCP. Configurarea parametrilor JDBC și a dialectului Hibernate se realizează centralizat în fișierul de proprietăți:

```
spring.datasource.url=jdbc:mysql://localhost:3306/university?useSSL=fals
e

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDiale
ct
```

4.2 Implementarea operațiilor CRUD

Operațiile CRUD (Create, Read, Update, Delete) reprezintă cele patru funcționalități fundamentale necesare pentru persistența datelor în orice aplicație care gestionează informații. Aceste operații corespund operațiilor SQL de bază: INSERT (Create), SELECT (Read), UPDATE (Update) și DELETE (Delete). În această secțiune vom detalia implementarea fiecărei operațiuni CRUD pentru entitatea Students, demonstrând fluxul complet de la nivelul Controller-ului până la interacțiunea cu baza de date.

4.2.1. Operația CREATE - Adăugarea unui student

Operația CREATE permite adăugarea de noi înregistrări în baza de date. În aplicația noastră, aceasta este implementată printr-un formular HTML care colectează datele studentului și o metodă Controller care procesează trimiterea formularului.

```
@GetMapping("/students/new")
public String createStudentForm(Model model) {
    Students student = new Students();
    model.addAttribute("student", student);
    return "create_student";
}

@PostMapping("/students")
public String saveStudent(@ModelAttribute("student") Students
```

```
student) {
    studentsService.saveStudent(student);
    return "redirect:/students";
}
```

Prima metodă, marcată cu @GetMapping("/students/new"), gestionează cererea GET pentru afișarea formularului de adăugare. Aceasta creează un obiect Students gol, îl adaugă în Model pentru a fi disponibil în view, și returnează numele template-ului Thymeleaf "create_student" care va fi randat.

A doua metodă tratează cererea de tip POST venită din formularul de adăugare. Adnotarea @PostMapping("/students") mapează metoda la URL-ul /students pentru cererile POST. Spring Boot preia automat valorile introduse de utilizator în câmpurile HTML (nume, prenume, email, etc.) și le mapează la proprietățile obiectului student prin data binding. Acest proces este facilitat de adnotarea @ModelAttribute care instruiește Spring MVC să populeze obiectul cu datele din formular.

Metoda apelează studentsService.saveStudent(student), care la rândul său apelează repository-ul pentru a persista obiectul în baza de date. JPA și Hibernate se ocupă automat de generarea și executarea comenzii SQL INSERT corespunzătoare:

```
INSERT INTO students (firstName, lastName, email, dateOfBirth,
                      enrollmentDate, studyYear)
VALUES (?, ?, ?, ?, ?, ?)
```

După salvarea cu succes, metoda returnează "redirect:/students", ceea ce instruiește browser-ul să realizeze o nouă cerere GET către URL-ul /students, afișând lista actualizată de studenți. Acest pattern se numește Post-Redirect-Get (PRG) și previne problema dublei trimiteri a formularului la refresh-ul paginii.

4.2.2. Operația READ - Afișarea listei de studenți

Operația READ permite extragerea și vizualizarea datelor din baza de date. În aplicație, aceasta este implementată pentru a afișa lista completă a studenților înregistrați:

```
@GetMapping("/students")
public String listStudents(Model model) {
    model.addAttribute("students",
studentsService.getAllStudents());
    return "students";
}
```

Această metodă este apelată când utilizatorul accesează pagina principală a studenților (URL-ul /students). Adnotarea @GetMapping mapează metoda la cererile GET pentru acest URL.

Metoda apelează serviciul pentru a extrage toate înregistrările din baza de date prin getAllStudents(), care execută echivalentul unei interogări SELECT * FROM students. Spring Data JPA generează automat implementarea acestei metode deoarece findAll() este moștenită din interfața JpaRepository.

4.2.3. Operația UPDATE - Modificarea unui student

Operația UPDATE permite modificarea datelor existente în baza de date. Implementarea necesită două metode: una pentru afișarea formularului pre-populat cu datele curente și alta pentru procesarea modificărilor:

```
@GetMapping("/students/edit/{idstudent}")
public String editStudentForm(@PathVariable Integer idstudent,
Model model) {
    Students student = studentsService.getStudentById(idstudent);
    model.addAttribute("student", student);
    return "edit_student";
}

@PostMapping("/students/{idstudent}")
public String updateStudent(@PathVariable Integer idstudent,
                            @ModelAttribute("student") Students
student,
                            Model model) {
    // Get student from database by idstudent field
    Students existingStudent =
studentsService.getStudentById(idstudent);
    existingStudent.setIdstudent(idstudent);
    existingStudent.setFirstName(student.getFirstName());
    existingStudent.setLastName(student.getLastName());
    existingStudent.setEmail(student.getEmail());
    existingStudent.setDateOfBirth(student.getDateOfBirth());

existingStudent.setEnrollmentDate(student.getEnrollmentDate());
    existingStudent.setStudyYear(student.getStudyYear());

    // Save modified student object
    studentsService.updateStudent(existingStudent);
    return "redirect:/students";
}
```

Prima metodă (@GetMapping) extrage studentul din baza de date folosind ID-ul primit din URL prin adnotarea @PathVariable. Acest ID este specificat în șablonul URL-ului ca {idstudent}, iar Spring îl extrage automat și îl injectează ca parametru al metodei. Obiectul student este adăugat în Model și formularul de editare este afișat cu câmpurile pre-completate.

A doua metodă (@PostMapping) realizează actualizarea efectivă a datelor. Mai întâi, se identifică studentul în baza de date folosind ID-ul primit din URL. Apoi, valorile vechi ale obiectului existingStudent sunt înlocuite cu noile valori primite din formularul de editare. Este important să înlocuim valorile unui obiect existent în loc să salvăm direct obiectul primit din formular, pentru a păstra contextul de persistență JPA și a evita crearea de duplicate.

La final, apelul metodei updateStudent (care folosește tot metoda save() din Repository) persistă modificările. Hibernate detectează automat că entitatea există deja în baza de date (prin prezența ID-ului) și generează o comandă UPDATE în loc de INSERT:

```
UPDATE students
SET firstName = ?, lastName = ?, email = ?,
    dateOfBirth = ?, enrollmentDate = ?, studyYear = ?
WHERE idstudent = ?
```

După actualizare, utilizatorul este redirecționat către lista de studenți pentru a vizualiza modificările.

4.2.4. Operația DELETE - Ștergerea unui student

Operația DELETE permite eliminarea înregistrărilor din baza de date. Implementarea în aplicație este realizată printr-o metodă simplă care primește ID-ul studentului ce trebuie șters:

```
@GetMapping("/students/{idstudent}")
public String deleteStudent(@PathVariable Integer idstudent) {
    studentsService.deleteStudentById(idstudent);
    return "redirect:/students";
}
```

Metoda este declanșată când utilizatorul apasă butonul de ștergere din interfață. Adnotarea @GetMapping mapează metoda la un URL care include ID-ul studentului ca parte din path, extras automat prin @PathVariable.

Serviciul apelează repository-ul care execută operația de ștergere. Spring Data JPA oferă metoda deleteById() care generează o interogare de tip DELETE FROM students WHERE idstudent = ?. După ștergere, utilizatorul este redirecționat către lista actualizată a studenților.

Este important de menționat că, în producție, ar trebui implementate mecanisme de confirmare înainte de ștergere pentru a preveni pierderea accidentală a datelor. De

asemenea, în cazul unor relații complexe între entități, trebuie luată în considerare utilizarea strategiilor de ștergere cascadă (CASCADE) sau a ștergerii soft (soft delete) care marchează înregistrările ca fiind șterse fără a le elimina fizic din baza de date.

4.3. Implementarea interfeței utilizator

Interfața utilizator a aplicației este construită folosind Thymeleaf ca motor de template-uri și Bootstrap pentru stilizare. Această combinație permite crearea de interfețe responsive și moderne cu efort minim.

Template-urile Thymeleaf sunt fișiere HTML care conțin atribute speciale prefixate cu th: pentru logica dinamică. De exemplu, th:text setează conținutul textual al unui element, th:each iterează prin colecții, th:if permite afișarea condiționată, iar th:href generează URL-uri folosind sintaxa @{...}.

Bootstrap, cel mai popular framework CSS pentru dezvoltarea web responsive, este utilizat pentru a asigura că interfața se adaptează corect la diferite dimensiuni de ecran. Componentele Bootstrap precum tabele, butoane, formulare și sistem de grid sunt integrate în template-urile Thymeleaf pentru o experiență utilizator consistentă.

Designul interfeței a fost facilitat de utilizarea Bootstrap Studio, un instrument vizual pentru crearea de machete HTML cu Bootstrap. Machetele create în Bootstrap Studio pot fi exportate ca fișiere HTML și integrate în aplicația Spring Boot, accelerând procesul de dezvoltare a interfeței.

CAPITOLUL 5
CONCLUZII ȘI PERSPECTIVE DE DEZVOLTARE

5.1. Concluzii

Lucrarea a validat eficiența framework-ului Spring Boot în dezvoltarea aplicațiilor web moderne, demonstrând practic beneficiile unei arhitecturi stratificate și modulare. Obiectivele propuse au fost îndeplinite integral prin:

- Asimilarea și aplicarea conceptelor din ecosistemul Spring;
- Proiectarea unei baze de date relaționale normalizate, cu respectarea integrității referențiale;
- Implementarea robustă a operațiilor CRUD prin abstractizările oferite de Spring Data JPA și Hibernate;
- Realizarea unei interfețe utilizator *responsive* prin integrarea Thymeleaf și Bootstrap.

Din perspectivă tehnică, alegerea Spring Boot s-a dovedit a fi soluția optimă pentru maximizarea productivității. Prin eliminarea configurărilor infrastructurale manuale — grație mecanismelor de *auto-configuration*, a serverului *embedded* și a gestionării dependențelor prin *Starters* — efortul de dezvoltare s-a putut concentra exclusiv asupra logicii de business, confirmând statutul framework-ului de standard în dezvoltarea aplicațiilor *enterprise* [24].

BIBLIOGRAFIE/SITOGRAFIE

[1] Spring.io, *"Why Spring? - Microservices, Reactive, Cloud"*. [Online]. Disponibil: https://spring.io/why-spring

[2] Baeldung, *"Introduction to Spring Boot"*, 2024. [Online]. Disponibil: https://www.baeldung.com/spring-boot-intro

[3] VMware, *"Spring Boot Reference Documentation - Auto-configuration"*, Spring.io. [Online]. Disponibil: https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.auto-configuration

[4] Oracle, *"Modern Web Development with Java and Spring Boot"*, Java Magazine. [Online]. Disponibil: https://blogs.oracle.com/javamagazine/post/modern-web-development-with-java-and-spring-boot

[5] Spring.io, *"Spring Boot Starters"*, Reference Documentation. [Online]. Disponibil: https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.build-systems.starters

[6] GeeksforGeeks, *"Spring MVC Architecture"*, 2024. [Online]. Disponibil: https://www.geeksforgeeks.org/spring-mvc-architecture/

[7] Spring Projects, *"Spring Data JPA - Reference Documentation"*. [Online]. Disponibil: https://spring.io/projects/spring-data-jpa

[8] Thymeleaf, *"Thymeleaf + Spring"*, Tutorial. [Online]. Disponibil: https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html

[9] Spring.io, *"Testing in Spring Boot"*, Reference Documentation. [Online]. Disponibil: https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.testing

[10] Spring.io, *"Spring Boot - Core Features"*, Reference Documentation. [Online]. Disponibil: https://docs.spring.io/spring-boot/docs/current/reference/html/features.html

[11] Baeldung, *"Spring Boot Starters: How They Work"*, 2024. [Online]. Disponibil: https://www.baeldung.com/spring-boot-starters

[12] Martin Fowler, *"GUI Architectures - MVC"*, martinfowler.com. [Online]. Disponibil: https://martinfowler.com/eaaDev/uiArchs.html

[13] GeeksforGeeks, *"Spring MVC Framework - Controllers"*, 2023. [Online]. Disponibil: https://www.geeksforgeeks.org/spring-mvc-controller/

[14] Vlad Mihalcea, *"The Best Way to Map Entities with JPA and Hibernate"*, 2023. [Online]. Disponibil: https://vladmihalcea.com/jpa-hibernate-entity-mapping/

[15] Spring Projects, *"Working with Spring Data Repositories"*, Spring Data JPA Docs. [Online]. Disponibil: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories

[16] Thymeleaf, *"Thymeleaf: The Natural Template Engine"*, Official Documentation. [Online]. Disponibil: https://www.thymeleaf.org/doc/articles/standarddialect5minutes.html

[17] Oracle, *"MySQL 8.0 Reference Manual - InnoDB Storage Engine"*. [Online]. Disponibil: https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html

[18] Spring.io, *"Embedded Web Servers in Spring Boot"*, Reference Docs. [Online]. Disponibil: https://docs.spring.io/spring-boot/docs/current/reference/html/web.html#web.servlet.embedded-container

[19] Baeldung, *"Validation in Spring Boot"*, 2024. [Online]. Disponibil: https://www.baeldung.com/spring-boot-bean-validation

[20] MySQL, *"MySQL 8.0 Reference Manual - Many-to-Many Relationships"*. [Online]. Disponibil: https://dev.mysql.com/doc/refman/8.0/en/create-table-foreign-keys.html

[21] Spring Projects, *"Spring Data JPA - Query Methods"*, Reference Documentation. [Online]. Disponibil: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods

[22] GeeksforGeeks, *"Spring MVC - Model Interface"*, 2023. [Online]. Disponibil: https://www.geeksforgeeks.org/spring-mvc-model-interface/

[23] Vlad Mihalcea, *"Why you should strictly avoid hibernate.ddl-auto in production"*, 2023. [Online]. Disponibil: https://vladmihalcea.com/hibernate-hbm2ddl-auto-schema-generation/

[24] VMware Tanzu, *"The State of Spring 2024 Report"*, Spring.io. [Online]. Disponibil: https://spring.io/state-of-spring