



UNIVERSITATEA DIN BUCUREȘTI



**FACULTATEA
DE
MATEMATICĂ ȘI INFORMATICĂ**

SPECIALIZAREA INGINERIE SOFTWARE

Lucrare de disertație

Dezvoltarea unei aplicații web pentru gestionarea activității unei firme folosind o arhitectură bazată pe microservicii

Absolvent

Badea Bogdan-Andrei

Coordonator științific

Lect.dr. Iulia Banu Demergian

București, septembrie 2023

Rezumat

Lucrarea are drept scop prezentarea procesului de dezvoltare a unei aplicații web pentru gestionarea activității unei firme mici. Aceasta poate fi folosită pentru a facilita comunicarea dintre angajați, cât și comunicarea dintre angajați și clienți. Funcționalitățile de bază ale aplicației le reprezintă componenta de autentificare, cea de programare a ședințelor și cea gestionare a ticket-elor.

Există trei tipuri de utilizatori care se pot autentifica în aplicație:

- Administrator: este cel care are rolul cel mai complex al aplicației și poate gestiona întreaga activitate. Acesta creează utilizatori noi, programează ședințe și gestionează ticket-e;
- Angajat: reprezintă persoanele care lucrează în firmă. Aceștia au acces la ședințele la care sunt invitați și rezolvă ticket-e;
- Client: reprezintă angajații ai firmelor client. Aceștia creează ticket-ele.

Aplicația a fost dezvoltată folosind o arhitectură client-server, bazată pe microservicii. Această abordare a fost aleasă datorită popularității în ascensiune a acestui concept. Aplicația este formată din trei servicii back-end, în Spring Boot, și unul front-end, în Angular. Acestea sunt pornite prin intermediul unor containere Docker. Aplicația se conectează atât la o bază de date relațională, cât și una NoSQL.

Abstract

The scope of the paper is to present the development process of a web application for managing the activity of a small company. It can have the potential to improve the communication between employees, as well as the communication between employees and clients. The main features of the application are the authentication component, the meeting scheduling component and the ticket management component.

There are three types of users who can log into the application:

- Administrator: is the one who has the most important role and the who can manage the entire activity. He creates new users, schedules meetings and manages tickets;
- Employee: represents the people who work in the company. They have access to the meetings to which they are invited and solve tickets;
- Client: represents the employees of the client companies. They create tickets.

The application was developed using a client-server architecture, based on microservices. This approach was chosen because of the rising popularity of this concept. The application consists of three backend service, in Spring Boot, and one frontend service, in Angular. They are started using Docker containers. The application connects to both a relational database and a NoSQL one.

Cuprins

1. Introducere	6
2. Preliminarii.....	8
2.1. Scopul aplicației și aria de acoperire	8
2.2. Interacțiunea cu aplicația	8
2.3. Studiul pieței.....	9
2.4. Tehnologii folosite.....	10
2.4.1. MongoDB.....	10
2.4.2. Docker	12
2.4.3. Java	15
2.4.4. Spring Boot.....	15
2.4.5. HTML.....	16
2.4.6. CSS.....	17
2.4.7. JavaScript	17
2.4.8. Typescript	18
2.4.9. Angular	18
3. Implementare.....	20
3.1. Structura aplicației.....	20
3.1. Baza de date.....	21
3.2. Containere Docker.....	23
3.3. Aplicația Server	24
3.3.1. Maven.....	24
3.3.2. IoC și DI	25
3.3.3. Container IoC	27
3.3.4. Bean.....	28
3.3.5. Scopuri Bean	30
3.3.6. Aspecte	34

3.3.7. Persistența datelor.....	36
3.3.7.1. Data Access Object.....	36
3.3.7.2. Java Database Connectivity.....	36
3.3.7.3. Object Realtional Mapping.....	37
3.3.7.4. Hibernate	37
3.3.7.5. Java Persistence API.....	38
3.3.7.6. Spring Data JPA.....	40
3.3.8. Spring MVC	42
3.3.9. Servicii web	43
3.3.10. REST API	44
3.3.11. Controller REST	45
3.3.12. Tratarea excepțiilor.....	46
3.3.13. WebClient	48
3.3.14. Spring Security	49
3.3.15. Arhitectura pe Microservicii.....	52
3.4. Aplicația Client.....	54
3.4.1. Componente.....	54
3.4.1.1. Data biding	56
3.4.2. Servicii.....	57
3.4.3. Observabile.....	58
3.4.4. Module.....	59
3.4.5. Routare	60
4. Concluzii	63

1. Introducere

În ziua de azi, tehnologia are un rol mai important ca niciodată în viața cotidiană, iar procesul de digitalizare este mai prezent decât oricând. În secolul vitezei, tot se întâmplă mai repede de oricând, iar accesul la informației este aproape instant. Vedem astăzi cum fiecare dintre noi interacționăm zilnic cu un computer, mai mult sau mai puțin, fie că este vorba despre un laptop sau un telefon inteligent. Aproape că nu mai putem concepe viața noastră fără calculator.

O dată cu apariția internetului în anii 1990, se poate spune că a avut loc o „A Patra Revoluție Industrială”. Deși conceptul de PC (Personal Computer) exista deja de ani buni, a fost nevoie de internet pentru ca acesta să joace un rol cu adevărat important în viețile noastre de zi cu zi. Comunicarea are acum un cu totul alt mod de desfășurare în ziua de azi, fiind poate aspectul cel mai mult modelat de această evoluție tehnologică.

În decursul anilor care s-au scurs de la începuturile WWW (World Wide Web), soluțiile software oferite pe piață au evoluat semnificativ, acestea devenind din ce în ce mai robuste și mai accesibile. Dacă la început, aplicațiile web erau destinate doar persoanelor cu deprinderi tehnologice avansate, în ziua de azi există soluții web pentru toate categoriile de persoane și pentru o arie largă de scenarii de utilizare. Această arie de acoperire din ce în ce mai largă, a necesita și ca proiectele software să devină din ce în ce mai complexe, pentru a răspunde nevoilor societății. Astfel, s-a ajuns la nevoie de flexibilitate a arhitecturilor software, pentru a gestiona volumul ridicat de lucru.

Aplicațiile web, bazate pe arhitecturi client-server pe microservicii au devenit din ce în ce mai populare în ultima perioadă. Principala caracteristică a acestei tipologii de dezvoltare a proiectelor, îl reprezintă împărțirea aplicațiilor în servicii web independente, fiecare cu scopuri bine definite. Astfel, sarcinile de lucru pot fi împărțite către unei echipe care să lucreze la implementarea acestora. Totodată, disponibilitatea aplicației, cât și securitatea acesteia, sunt îmbunătățite semnificativ, prin împărțirea resurselor în module mai mici.

În această lucrare am decis să prezint o aplicație de gestionare a activității unei firme mici, cu un număr limitat de angajați și cu o activitate restrânsă. Consider că astfel de operatori economici trebuie sprijiniți cât mai mult, deoarece ei sunt principalul motor al economiei, iar din rândurile lor se nasc firmele mari. Am ales ca exemplu pentru această lucrare o firmă care se ocupă cu mentenanța componentelor hardware și software pentru alte firme de pe plan local.

În trecut am avut ocazia să lucrez atât într-o astfel de firmă, cât și într-o multinațională. Prima dată a fost vorba despre un stagiu de practică, iar a doua oară despre un internship. Deși nu pot spune că am acumulat o experiență semnificativă pe piața muncii în domeniul IT, am reușit să îmi formez o oarecare opinie (simplistă, ce este drept) despre modul de organizare al activității într-o firmă din acest domeniu. În diferite situații mai confruntat cu o lipsă de organizare a sarcinilor de lucru primite, lucru pe care l-am observat și la alți colegi la început de lucru. Toate cerințele care vin pot părea copleșitoare pentru o persoană fără experiență. Astfel, mi-a venit ideea de a dezvolta o soluție software pentru a încerca să rezolv această problemă.

Aplicația prezentată în lucrare are scopul de a gestiona, activitatea de bază a unei firme mici, așa cum am precizat că am ales drept exemplu pentru această lucrare. Principalele două necesități pe care le-am considerat necesare pentru o astfel de aplicație au fost cea de gestionare a ședințelor din firmă și cea de gestionare a cerințelor clienților.

2. Preliminarii

2.1. Scopul aplicației și aria de acoperire

Aplicația are drept scop răspunderea la nevoia de o mai bună gestionare a activității unei firme care se ocupă cu mentenanța echipamentelor hardware și software pentru alte firme. Acest tip de afacere este destul de întâlnit în țara noastră, majoritatea operatorilor economici având nevoie de echipamente de calcul pentru a își desfășura activitatea.

Deși majoritatea angajații firmei care se ocupă cu mentenanța au deprinderi tehnice peste medie, nu același lucru se poate zice și despre angajații firmelor client. De multe ori, aceștia abia au reușit să dobândească abilități de lucru digitale, iar interacțiunea cu o aplicație nouă se poate dovedi de multe ori o provocare. Astfel, aplicația prezentată a fost gândită să aibă o interfață cât mai simplă și mai intuitivă pentru utilizatori, pentru a acoperi o plajă cât mai largă de clienți.

2.2. Interacțiunea cu aplicația

Există trei tipuri de utilizatori ai aplicației: administrator, angajat și client.

- **Administratorul:** reprezintă angajații companiei cu rol în coordonarea activității acesteia. Aceștia sunt responsabili în mare parte de toate datele prezente în aplicație. Administratorii au rolul principal de a crea alți utilizatori pentru aplicație, cum ar fi angajații sau clienții. În plus, ei sunt cei care programează ședințele, la care iau parte angajații specificați de aceștia;
- **Angajatul:** reprezintă pionul principal al firmei. Acesta are posibilitatea de a vedea toate ședințele la care trebuie să participe, cât și task-urile pe care trebuie să le rezolve. În mare parte, aplicația este gândită ca activitatea pe care angajații o au de făcut să fie accesibilă în mod cât mai ușor pentru aceștia;
- **Clientul:** este în general reprezentat de către angajați ai firmelor client. Atunci când au o problemă cu echipamentul de lucru, aceștia deschid un ticket în aplicație. Pe baza detaliilor furnizate de clienți, un angajat al firmei va rezolva ticket-ul respectiv.

2.3. Studiul pieței

Deși există mai multe soluții software pe piață pentru rezolvarea problemelor adresate de aplicație, am considerat că totuși acestora le lipsește ceva. Majoritatea acestor aplicații sunt create de companii mari, care încearcă să ofere experiențe cât mai robuste, dar consider că aceste scenarii nu sunt mereu potrivite și pentru firmele mici.

La primul loc de muncă, am avut ocazia de a vedea cum se lucrează cu Autotask PSA, de la compania Datto. Aplicația oferea toate caracteristicile necesare pentru gestionarea activității unei firme și în special a task-urilor. Totuși aplicația pare mult prea complexă pentru multe scenarii de utilizare, iar clienților le poate fi dificil să se obișnuiască cu interfața. De multe ori am sesizat cum aceștia nu mai foloseau aplicația de ticket-e, preferând să utilizeze servicii de mail sau de telefonie mobilă.

La cel de-al doilea loc de muncă, mediul de lucru a fost mult mai organizat, specific unei multinaționale. Toate proiectele erau dezvoltate după metodologia SCRUM, iar organizarea task-urilor era realizată folosind Jira. Ședințele aveau loc pe platforma Microsoft Teams. Deși soluțiile software adoptate de companie erau populare și relativ ușor de utilizat, modul în care acestea erau implementate se concentra exclusiv pe proiecte. Pentru realizarea altor sarcini de lucru, care nu țineau de proiect, nu mai exista o soluție centralizată, folosindu-se în mod special aplicația de e-mail pentru comunicare.

Astfel, am ales să dezvolt această aplicație care, după părerea mea, poate reuși să acopere un gol lăsat în piață. Dacă majoritatea companiilor obligă clienții să folosească întreaga lor suită de servicii, aplicația prezentată în lucrare funcționează în mod independent, utilizatorii având nevoie doar de un cont de e-mail valid pentru a o accesa. În rest, nu sunt necesare alte soluții software complementare. Mai mult, nu există costuri de utilizare a aplicației, aceasta fiind open-source.

2.4. Tehnologii folosite

2.4.1. MongoDB

Bazele de date SQL, numite și baze de date relaționale, reprezintă colecții de tabele, în care fiecare entitate este stocată sub forma unui rând, fiecare coloană conținând același tip de informație despre entități. Pentru manipularea datelor din aceste tabele sunt folosite limbaje structurate de interogare (SQL). Dintre cele mai populare limbaje SQL se pot enumera: MySQL, Oracle, SQL Server etc.

Serverele care operează baze de date SQL stochează și organizează datele în tabele. În cadrul de management al sistemelor bazelor de date relaționale, acestea sunt fundamentale și sunt proiectate pentru a stoca date sub formă de rânduri, pe mai multe coloane. În timp ce rândurile reprezintă câte o entitate, coloanele definesc attributele acestora. De exemplu, într-o tabelă care stochează utilizatorii unei aplicații, fiecare rând ar reprezenta câte un utilizator în parte, în timp ce fiecare coloană ar conține o anumită informație despre aceștia, precum id-urile, numele utilizatorilor și altele.

Bazele de date SQL reprezintă piatra de temelie a unei vaste majorități de aplicații și servicii, esențiale pentru un număr mare de industrii. Companiile și instituțiile se bazează pentru stocarea și prelucrarea datelor pe servere de baze de date SQL, datorită capacităților lor operaționale, precum procesarea tranzacțiilor, analiza datelor etc.

Față de bazele de date SQL, care salvează datele în tabele relaționale, bazele de date NoSQL rețin datele prin intermediul documentelor. Astfel, acestea pot fi clasificate ca fiind baze de date „nu numai SQL” și pot fi împărțite într-o gamă largă de modele cât mai flexibile. Printre cele mai populare tipuri de baze NoSQL se numără cele de tip document, bazele cheie-valoare, bazele wide-column și bazele de tip graf. Bazele de date NoSQL au fost proiectate pentru a stoca și procesa date la scară largă, ajungând astfel să se regăsească în cadrul tot mai multor soluții de business moderne.

Tehnologia pe care o folosesc bazele de date NoSQL stochează informații în documente de tip JSON, față de coloane și rânduri ca în tabelele relaționale. Aceste baze se mai numesc și „nu numai SQL”, deoarece manipulează date fără a folosi un limbaj SQL. Totuși, se pot combina flexibilitatea JSON cu standardizarea SQL pentru a putea folosi caracteristicile cele mai avantajoase din ambele părți. Bazele de date NoSQL au fost gândite pentru a fi cât mai

flexibile, scalabile și rapide, atunci când răspund la cerințelor tot mai solicitante ale aplicațiilor moderne.

Printre cele mai des întâlnite tipuri de baze de date NoSQL, se numără patru care sunt cele mai des utilizate. Fiecare dintre ele folosesc modele de date diferite, astfel că fiecare sunt utilizate în scenarii de lucru diverse: [1]

- Bazele de date de tip documente: numite și depozite de documente, acestea stochează datele semi-structurate și descrierea lor în documente. Bazele de tip document permit dezvoltatorilor software să facă modificări asupra aplicațiilor, fără a fi nevoie să schimbe schema principală a aplicației. Utilizarea acestora a devenit din ce în ce mai frecventă, o dată cu creșterea în popularitate a framework-urilor care utilizează JavaScript și JSON. Astfel de baze de date sunt utilizate pentru dezvoltarea aplicațiilor web și mobile, precum aplicații de comerț electronic, platforme de blogging etc.
- Bazele de date grafice: utilizează reprezentarea datelor sub formă de noduri, care pot fi văzute drept echivalentul rândurilor din bazele de date relaționale, și conexiuni între noduri, numite margini. Datorită modului în care sunt stocate relațiile dintre noduri, modelele grafice oferă o mai bună reprezentare a relațiilor dintre date. Astfel de modele pot scala ușor în timp, datorită arhitecturii lor flexibile, spre deosebire de modelele relaționale stricte. Bazele de date grafice sunt utilizate în dezvoltarea aplicațiilor care pun accentul pe relațiile dintre entități, precum rețelele de socializare, sau aplicațiile de gestiune a serviciilor de rezervări.
- Bazele de date de tip cheie-valoare: sunt baze de date care lucrează cu modele simple, în cadrul cărora, pentru fiecare valoare este asociată o cheie unică. Datorită simplității acestor modele, aplicațiile care utilizează bazele de date de tip cheie-valoare pot fi scalate ușor și rapid. Acestea sunt folosite în general pentru management-ul sesiunilor sau pentru stocarea memoriei cache a aplicațiilor web. Diferența dintre astfel de baze de date constă în modul în care acestea aleg să folosească resursele, precum RAM sau SSD-uri/hard disk-uri. Printre cele mai des întâlnite scenarii de utilizare în care se folosesc bazele cheie-valoare, se regăsesc gestiunea coșului de cumpărături dintr-o aplicație de comerț online sau gestionarea sesiunilor platformelor multiplayer.
- Bazele de date wide-column: seamănă cel mai mult cu tabelele folosite în bazele de date relaționale, diferența constând în faptul că numele coloanelor și formatarea

acestora pot să difere de la un rând la altul, în interiorul aceluiași tabel. Fiecare coloană este stocată separat, astfel că interogările se fac de asemenea pe coloane, față de tabelele relaționale, unde interogările se fac pe rând. Acest tip de baze de date sunt cel mai des folosite în dezvoltarea motoarelor de căutare și recomandare, cataloage etc.

MongoDB utilizează este o bază de date de tip NoSQL care utilizează documente pentru stocarea datelor, sub formă de perechi de câmpuri și valori. Documentele reprezintă unitatea structurală de bază a MongoDB și sunt asemănătoare cu JSON, dar folosesc BSON. Față de JavaScript Object Notation, avantajul Binary JSON este că suportă o gamă mai largă tipuri de date. Câmpurile din documente pot fi considerate echivalentul coloanelor din bazele de date SQL. Valorile stocate pot conține o gamă largă de tipuri de date, precum alte documente, matrice de documente etc. Documentele au o cheie primară care are rolul de element de identificare unic.

2.4.2. Docker

Virtualizarea este o tehnologie ce permite utilizarea resurselor fizice ale mașinilor de calcul într-un mod cât mai eficient. Aceasta presupune de crearea de versiuni virtuale ale componentelor și sistemelor hardware, precum serverele sau dispozitivele de rețea. Aceste copii virtuale imită comportamentul resurselor hardware pentru a putea rula mai multe instanțe virtuale, folosind aceleași resurse fizice. Astfel, resursele sunt distribuite într-un mod cât mai eficient, iar siguranța echipamentelor fizice crește. Conceptul de virtualizare stă la baza cloud computing-ului. [2]

Virtualizarea oferă firmelor care dezvoltă de aplicații web mai multă flexibilitate în gestionarea resurselor fizice de care aceasta dispune. Echipamentele de care au nevoie produsele software pentru a funcționa sunt scumpe, ocupă spațiu și au nevoie de mentenanță în mod constant. În plus, fiecare aplicație ar avea nevoie de propriul server pentru a funcționa adecvat, în condiții de siguranță.

Principalul concept care stă la baza virtualizării este cel de abstractizare. Resursele hardware sunt separate de cele software prin adăugarea unui layer între acestea. Astfel, fiecare aplicație poate rula în siguranță în propriul ei mediu, dar în același timp pe același server cu alte aplicații, utilizând doar resursele necesare lor.

VM-urile (virtual machines) reprezintă un mediu virtual care simulează funcționarea unei mașini de calcul fizice. O mașină de calcul fizică, numită sistem gazdă (host), poate rula simultan mai multe mașini virtuale, numite guest. Fiecare mașină virtuală are propriul sistem de operare, în funcție de aplicațiile care sunt rulate pe aceasta. Resursele mașinii fizice sunt împărțite între mașinile virtuale, în funcție de nevoi. Astfel, aplicațiile nu mai au acces direct la resursele fizice, fapt ce contribuie pozitiv la eficiența și siguranța serverelor.

Mașinile guest sunt separate de mașina host prin introducerea unui layer numit hypervisor. Acesta este un soft care are rolul de gestiona alocarea resurselor fizice pentru mașinile virtuale. Numit și VMM (Virtual Machine Monitor), hypervisor-ul izolează resursele hardware de mașinile gazdă și permite crearea și gestionarea acestora.

Hypervisor-ul alocă fiecărei mașini resursele necesare, care i-au fost alocate în prealabil. Execuția proceselor este făcută tot de componentele fizice ale mașini gazdă, hypervisor-ul asigurând conexiunea cu acestea și ordinea executării proceselor, în cazul rulării mai multor mașini guest în paralel. Mașinile virtuale pot rula chiar sisteme de operare, indiferent de sistemul de operare al mașinii gazdă

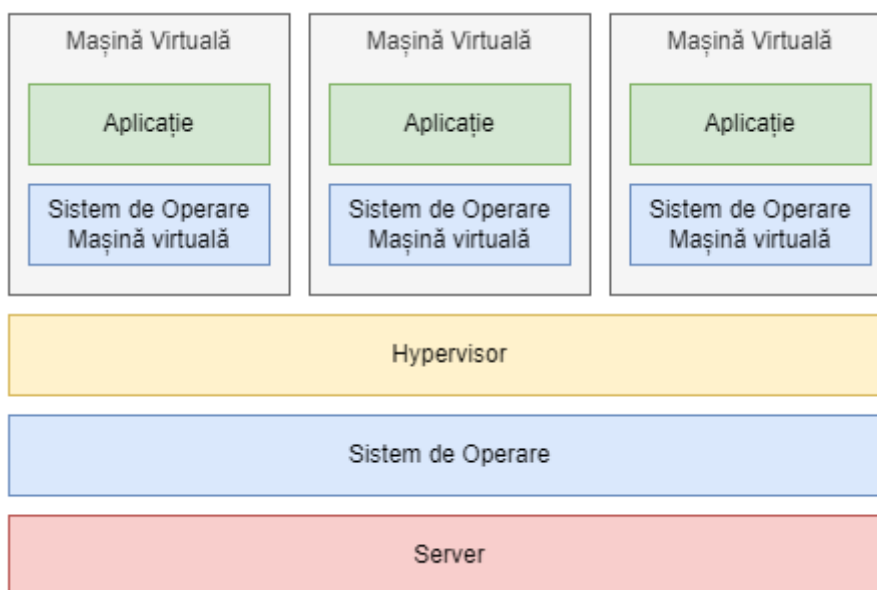


Fig. 1: Arhitectură mașini virtuale

Există două tipuri de hypervisor-i care pot fi folosiți pentru mașinile virtuale:

- Tipul 1: numit și hypervisor nativ, acesta este rulat direct la nivelul componentelor hardware ale mașinii gazdă. Acesta tip nu se mai conectează la sistemul de operare

al mașinii. Exemple de astfel de hypervisor-i sunt KVM (Kernel-based Virtual Machine) pentru kernelul Linux sau Microsoft Hyper-V;

- Tipul 2: numit și hypervisor hosted, este tipul clasic de hypervisor care este rulat pe un sistem de operare ca o aplicație. Acest tip este cel mai des folosit de către utilizatorii comuni. Cel mai popular astfel de hypervisor este Oracle VirtualBox.

Asemenea mașinilor virtuale, containerele reprezintă o tehnologie de virtualizare. Containerele împachetează aplicațiile într-un mediu de rulare cu toate fișierele necesare, izolat de sistemul de operare. Astfel, aplicațiile pot fi mutate fără prea mare efort de pe o mașină pe alta, fără prea multe configurări necesare ale dezvoltatorilor, fără a fi nevoie să se țină cont de sistemul de operare pe care mașinile îl operează. La fel ca la mașinile virtuale, un server poate rula mai multe containere simultan.

Avantajul principal al containerelor constă în portabilitate. Acestea folosesc același kernel cu sistemul de operare, astfel că nu au nevoie de un sistem de operare propriu, ca în cazul mașinilor virtuale. Kernel-ul este componenta cea mai importantă a sistemelor de operare, care acționează ca un layer între aplicații și operațiile efectuate de componentele hardware. Totodată, containerelor nu li se mai alocă resurse în prealabil, managementul resurselor fiind realizat de către un motor. Cel mai popular astfel de motor este Docker, care permite crearea de containere Linux.

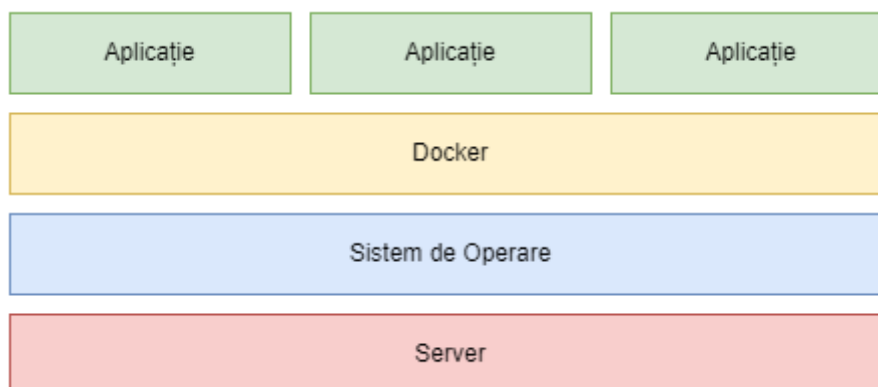


Fig. 2: Arhitectură Containere Docker

Un al doilea avantaj al containerelor îl reprezintă capacitatea acestora de comunicare. Containerele pot forma rețele prin care acestea pot schimba informații în mod sigur, fără interacțiuni din exterior. Acest aspect este esențial în deployment-ul aplicațiilor ce se bazează pe o arhitectură pe microservicii.

2.4.3. Java

Java este un limbaj de programare orientat pe obiecte și se bazează pe clase. A fost dezvoltat pe baza principiului WORA („Write once, run anywhere”), aplicațiile Java putând fi rulate indiferent de platformă, atât timp cât aceasta care suportă limbajul. Java este similar cu C și C++. Conform Github, Java este al treilea cel mai des utilizat limbaj de programare, după JavaScript și Python. În general, limbajul este folosit pentru implementarea aplicațiilor web de tip client-server.

2.4.4. Spring Boot

Spring este un framework open-source bazate pe limbajul de programare Java, lansat în 2003. Cu ajutorul unor extensii, acesta permite dezvoltarea de aplicații web bazate pe platforma Java EE (Enterprise Edition).

Termenul „Spring” poate avea mai multe semnificații, în funcție de context, precum framework-ul Spring în sine. În timp, alte tehnologii au fost dezvoltate pornind de la framework-ul de bază, iar termenul mai sus menționat a ajuns să fie asimilat cu totalitatea acestor proiecte. [4]

Spring este o tehnologie robustă și populară ce permite dezvoltarea de aplicații de business în Java. Versiunea de Spring folosită pentru realizarea aplicației este 5.3.6, iar cea de Spring Boot este 2.4.5. Pentru JDK (Java Development Kit) a fost aleasă versiunea 17.0.5. JDK asigură mediul de lucru și instrumentele necesare pentru executarea și compilarea de programe Java. JDK include JRE (Java Runtime Environment), o suită de componente software utilizate de aplicațiile Java. La rândul său, JRE include JVM (Java Virtual Machine). JVM este o implementare a unei mașini virtuale ce execută programul Java.

Spring Boot este un proiect din cadrul familiei de proiecte a framework-ului Spring. Este unul dintre cele mai populare astfel de proiecte, iar scopul său a fost simplificarea semnificativă a dezvoltării de aplicații folosind Spring.

Printre principalele caracteristici introduse de Spring Boot se numără: [7]

- Spring Boot starters: au rolul de a grupa mai multe dependențe cu roluri apropiate într-o singură dependență, care poate fi apoi adăugată în Maven sau Gradle;

- Autoconfiguration: Spring Boot are capacitatea de a configura de care bean-uri are nevoie aplicația pentru a funcționa și le configurează automat. Pentru asta, dezvoltatorii de aplicații trebuie să urmeze convenții standard de implementarea codului;
- Actuator: este o unealtă ce are rolul de a aduna și gestiona date ale aplicației după ce aceasta a fost rulată. Actuator oferă astfel dezvoltatorilor informații despre modul în care se comportă aplicația;
- Executabile JAR (Java Archive): pentru ca aplicația să fie lansată în producție, aceasta este împachetată sub forma de pachet executabil pentru a fi rulată pe server. Astfel, Maven și JDK nu mai trebuie instalate pe alte mașini pentru a rula aplicația, acestea fiind folosite doar în procesul de dezvoltare.

2.4.5. HTML

HTML (Hyper Text Markup Language) este cel mai popular limbaj de dezvoltarea a paginilor Web. Acesta are rolul de a afișa elementele în pagină. Primele pagini web apărute în anii 1990 erau statice, acestea având scopul doar de a afișa informații, iar interacțiunea utilizatorilor cu acestea era minimă. [8]

HTML este un limbaj de tip markup. Față de un limbaj de programare, acesta nu conține caracteristici precum variabile sau metode. HTML este folosit pentru a defini documente web, care apoi sunt interpretate de un browser și astfel sunt create paginile web.

Indiferent de framework-ul folosit pentru dezvoltarea aplicațiilor web, singurul limbaj care este general utilizat de către toate este HTML.

2.4.6. CSS

CSS (Cascading Style Sheet) este un alt limbaj popular folosit în dezvoltarea paginilor Web. Rolul acestuia este de a descrie modul în care elementele HTML vor fi afișate în pagină. CSS a fost gândit pentru a stiliza paginile Web. Elementele HTML sunt simple și inexpresive, astfel, toate site-urile ar arăta la fel și nu ar fi atrăgătoare pentru publicul larg. CSS are scopul de a formata elementele HTML din pagină, astfel încât informația prezentată de acestea să fie afișată într-un mod mai estetic și mai intuitiv. [8]

Asemenea HTML, CSS este un limbaj markup, nu unul de programare. În schimb, CSS sporește nivelul de interacțiune al utilizatorului cu aplicația, prin faptul că elementele își pot schimba aranjamentul în pagină, în funcție de acțiunile utilizatorului.

Față de HTML, care este general utilizat de toate framework-urile, pentru CSS există alte alternative, precum:

- SCSS (Sassy Cascading Style Sheets): un superset al CSS care permite dezvoltatorilor web mai mult control asupra design-ului, oferind noi funcționalități;
- SASS (Syntactically Awesome Style Sheets): un limbaj pe bază de scripturi compilate în CSS înainte de rularea aplicației.

2.4.7. JavaScript

JS (JavaScript) este un limbaj de programare interpretat și ușor. Este cel mai frecvent utilizat limbaj de programare, folosit în dezvoltarea aplicațiilor front-end. Față de alte limbaje de programare, JavaScript este executat în mod direct, fără a fi necesară compilarea acestuia în avans. [8]

Pentru a putea crea componente Web dinamice, care să-și schimbe comportamentul în funcție de interacțiunea cu utilizatorul, aplicațiile web au nevoie de limbaje de programare. Astfel, codul JavaScript este executat de browser, pentru ca utilizatorul să aibă acces la aplicația client.

Popularitatea JavaScript este dată de faptul că poate fi interpretat de orice browser. Astfel, majoritatea framework-urilor de dezvoltare a aplicațiilor web folosesc JavaScript sau limbaje bazate pe acesta, precum TypeScript.

2.4.8. Typescript

TS (TypeScript) este o extensie a limbajului JavaScript. Față de JavaScript, care este un limbaj loosely typed, TypeScript este un limbaj strongly-typed, ceea ce înseamnă că variabilă trebuie definită înainte de a fi utilizată, precizând tipul acesteia. În plus, TypeScript este un limbaj compilat, avantajul fiind că erorile pot fi identificate de la început. Acest lucru reprezintă scopul principal pentru care a fost creat Typescript, acela de a dezvolta aplicații web mari.

TypeScript conține toate caracteristicile JavaScript, plus alte caracteristici proprii. La compilare, acesta limbajul TypeScript este mai întâi rescris în JavaScript și abia apoi este interpretat de browser. Astfel, este eliminată orice problemă de compatibilitate cu diferitele browser-e existente.

Datorită faptului că înglobează toate caracteristicile JavaScript, orice fișier cu extensia „.js” poate fi convertit într-un fișier TypeScript, cu extensia „.ts”. În plus, TypeScript poate utiliza orice librărie JavaScript. Principala utilizare a TypeScript este în framework-ul Angular.

2.4.9. Angular

Angular este un framework open-source de dezvoltarea a aplicațiilor web. Există două variante ale framework-ului:

- Angular JS: numit și Angular 1, este prima variantă a framework-ului, dezvoltată pentru a folosi JavaScript;
- Angular: care include toate variantele Angular lansate o dată cu varianta a doua a framework-ului. Față de prima versiune, cea de a doua a fost rescrisă complet pentru a folosi TypeScript.

Alături de React, Angular se numără printre cele mai utilizate framework-uri de front-end bazate pe JavaScript (TypeScript este bazat pe JavaScript și rescris în acesta la compilare). Față de React, care este o librărie, Angular este un framework de sine stătător.

Angular este gândit pentru a dezvolta aplicații web de tip SPA (Single-Page Application). Implementarea unei aplicații SPA presupune ca framework-ul să încarce un singur document web, iar apoi să modifice conținutul acestuia prin intermediul unui limbaj de programare. Astfel, la fiecare interacțiune a utilizatorilor sunt încărcate doar componentele noi, cele asupra cărora nu se acționează rămânând neschimbate. Această dinamică a componentelor

elimină necesitate reîncărcării unei paginii web în totalitate, iar timpul de așteptare este redus considerabil.

O aplicație Angular se bazează pe artefacte pentru a putea funcționa. Acestea reprezintă clase TypeScript pe care framework-ul le interpretează drept artefacte. Din aceste punct de vedere, artefactele Angular pot fi asemăunate cu bean-urile Spring, deoarece reprezintă unitatea de bază a framework-ului, pe care acesta le controlează, fiind implementări ale conceptului injectare a dependențelor. [9]

3. Implementare

3.1. Structura aplicației

Pentru implementarea aplicației, am ales să utilizez o arhitectură client-server, pe microservicii. Componenta de back-end constă în trei astfel de servicii:

- Serviciul de autentificare: Acesta gestionează logica de creare și gestionare a utilizatorilor, cât și logica de autentificare în aplicație;
- Serviciul de programare a ședințelor: Acesta are rolul de a reține toate ședințele programate și de a notifica utilizatorii aferenții. Tot odată, acesta conține toate datele necesare de validare a ședințelor, pentru a evita orice eventuale suprapuneri cu alte activități;
- Serviciul de ticketing: Acesta este responsabil de gestionarea ticket-elor deschise de către clienți. Fiecare ticket este asignat unui utilizator.

Componenta de front-end a fost dezvoltată la fel, asemenea unui serviciu web, care se are posibilitatea de a apela toate cele trei servicii back-end, în funcție de unde se află endpoint-ul dorit.

3.1. Baza de date

Serviciul de autentificare este conectat la o bază de date relațională MySQL, în timp ce restul serviciilor vor folosi câte o bază MongoDB. Astfel, pentru a respecta pe deplin principiul de separare a microserviciilor, fiecare utilizează propria bază de date.

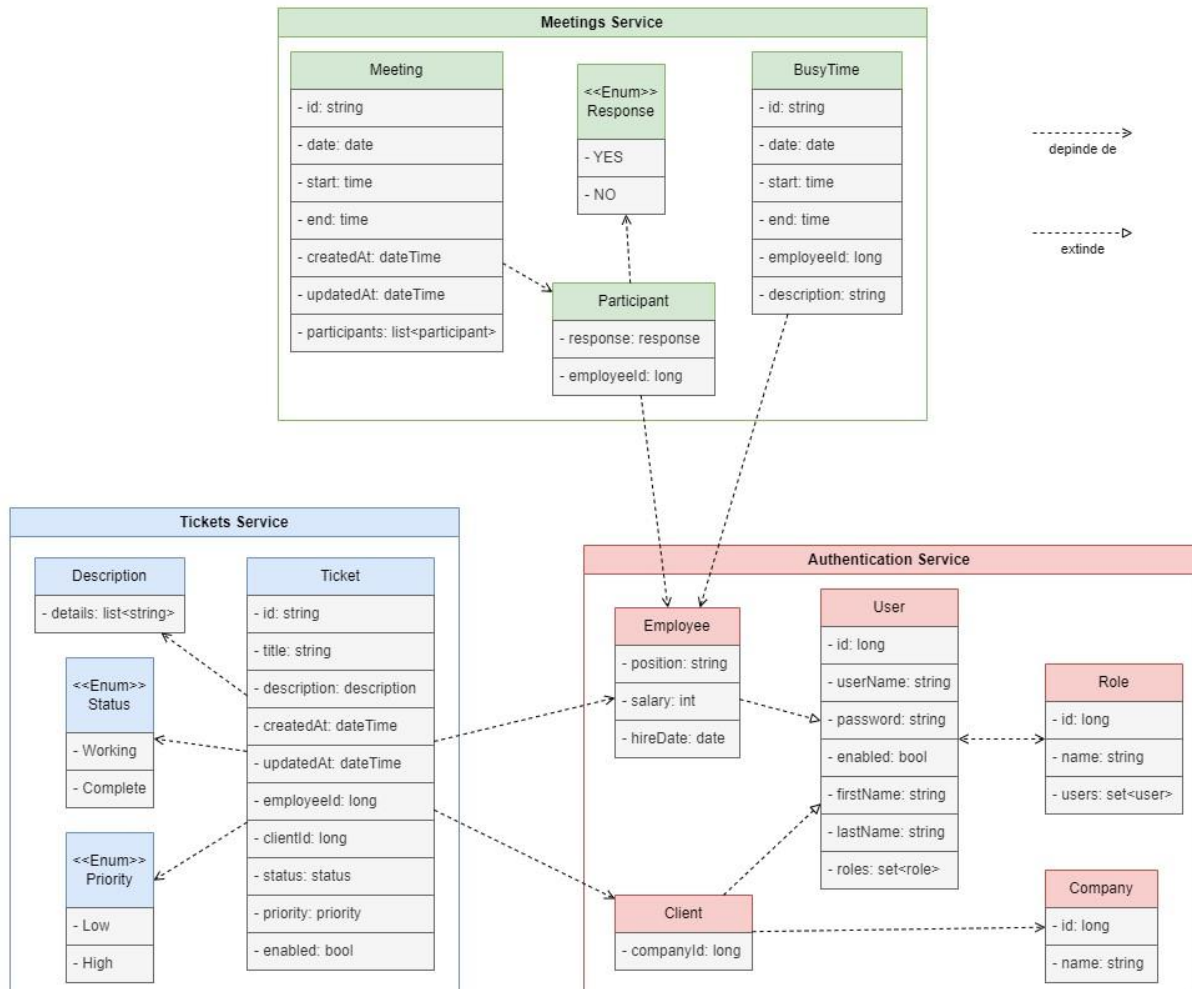


Fig. 3: Structura aplicației

În exemplul de mai sus, sunt toate entitățile aplicației și sunt grupate fiecare în funcție de serviciul din care fac parte. Entitățile din serviciul de Autentificare vor fi salvate într-o bază de date relațională MySQL, în timp ce entitățile serviciilor de Ședințe și Tickete vor fi salvate în baze de date MongoDB.

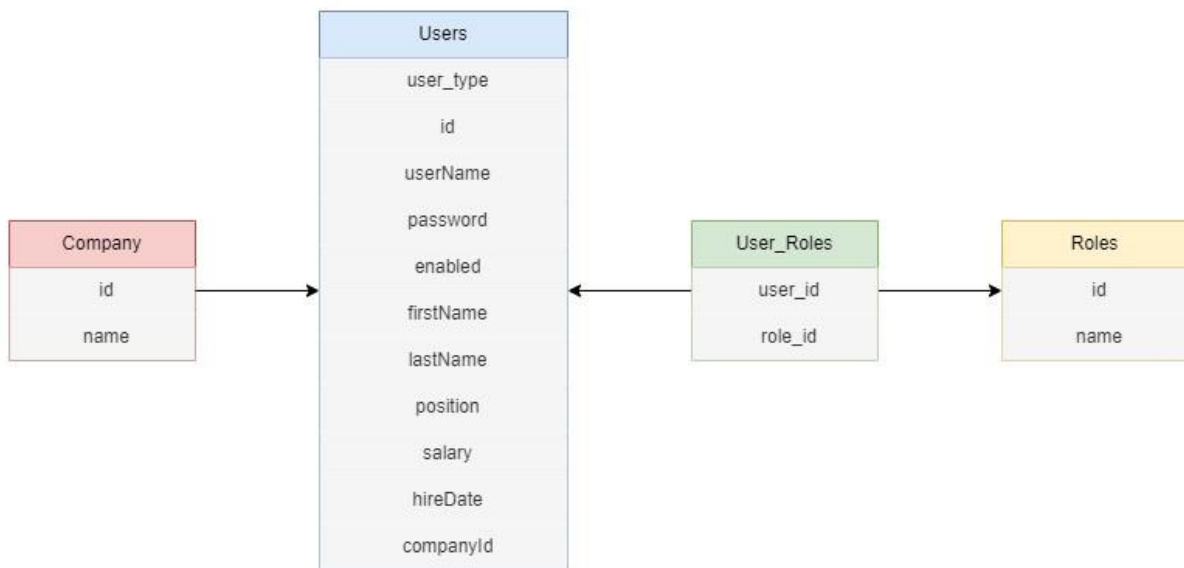
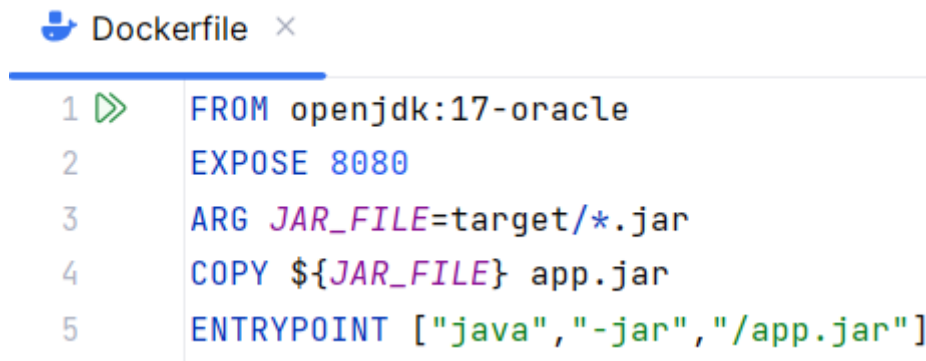


Fig. 4: Schema conceptuală a bazei de date relațională

În figura de mai sus este reprezentată schema conceptuală a bazei de date relațională MySQL, singura de altfel prezentă în aplicație. Pentru gestionarea relației de Many-to-Many, dintre tabelele „Users” și „Roles”, aceasta a fost împărțită în două relații de tip One-to-Many și a fost creată o tabelă auxiliară cu chei externe către cele două, numită „User_Roles”.

3.2. Containere Docker

Containerele Docker se bazează pe imagini. O imagine Docker reprezintă un pachet executabil ce conține fișierele de care aplicația are nevoie pentru a rula. Pentru a crea o astfel de imagine, Docker are nevoie de un fișier numit Dockerfile, în care sunt definite toate comenzile necesare pentru asamblarea imaginii. [3]



```
Dockerfile x
1 FROM openjdk:17-oracle
2 EXPOSE 8080
3 ARG JAR_FILE=target/*.jar
4 COPY ${JAR_FILE} app.jar
5 ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Fig. 5: Dockerfile

În imaginea de mai sus este ilustrat Dockerfile-ul serviciului de autentificare. Pe prima linie este definită imaginea de bază folosită pentru Java. Imaginile Docker au proprietatea de a moșteni imagini deja existente. Apoi este precizat portul prin care se realizează comunicare cu containerul. Mai departe v-a fi folosit un JAR, care are rolul de a partaja aplicația sub formă de pachet executabil, iar acest pachet este copiat în imagine. În final este precizată comanda care pornește containerul.

3.3. Aplicația Server

Aplicația server a fost creată folosind framework-ul Spring Boot, descris într-unul din capitolele anterioare. În acest capitol sunt prezentate concepte ale framework-ului și cum au fost acestea implementate în baza de date.

3.3.1. Maven

Maven reprezintă o tehnologie automată de construire a proiectelor Java. Rolul Maven este de a oferi o perspectivă asupra ciclului de viață al aplicației. Pe scurt, Maven gestionează cum este construit un produs software, precum și toate dependențele și plug-in-urile de care acesta are nevoie pentru a funcționa.

Principala caracteristică a proiectelor Maven cu care interacționează dezvoltatorii software este POM (Project Object Model). Fișierul „pom.xml” este întotdeauna localizat în directorul de bază al proiectului. Acesta conține toate informațiile cu privire la proiect, precum versiunea framework-ului în care este construit, precum și toate dependențele pe care acesta le folosește.

```
m pom.xml (Authentication) ×
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4      <modelVersion>4.0.0</modelVersion>
5      <parent>
6          <groupId>org.springframework.boot</groupId>
7          <artifactId>spring-boot-starter-parent</artifactId>
8          <!-- <version>2.7.5</version> -->
9          <version>2.4.5</version>
10         <relativePath/> <!-- lookup parent from repository -->
11     </parent>
12     <groupId>com.dissertation</groupId>
13     <artifactId>Authentication</artifactId>
14     <version>0.0.1-SNAPSHOT</version>
15     <name>Authentication</name>
16     <description>Authentication</description>
17     <properties>
18         <java.version>17</java.version>
19     <!-- added here -->
20     <spring-cloud.version>2020.0.5</spring-cloud.version>
21 </properties>
22 <dependencies>
```

Fig. 6: Fragment fișier pom.xml

3.3.2. IoC și DI

IoC (Inversion of Control) este un principiu software prin care control asupra flow-ului unor obiecte revine unui framework generic sau unui container. Este un principiu foarte des întâlnit în programarea orientată pe obiecte. [4]

Acest principiu este o componentă cheie în diferența dintre un framework și o librărie. O librărie constă într-o suită de funcții pe care dezvoltatorul are posibilitatea să le acceseze din cod. Fiecare apel către o librărie se realizează în ordinea scrierii și este întors imediat un răspuns, după care se continuă execuția. Ce face în plus un framework față de o librărie, prin intermediul IoC, este să controleze ordinea de execuție a codului aplicației, făcând apeluri către obiecte în funcție de context. Pentru a reuși asta, framework-ul introduce un nivel suplimentar de abstractizare.

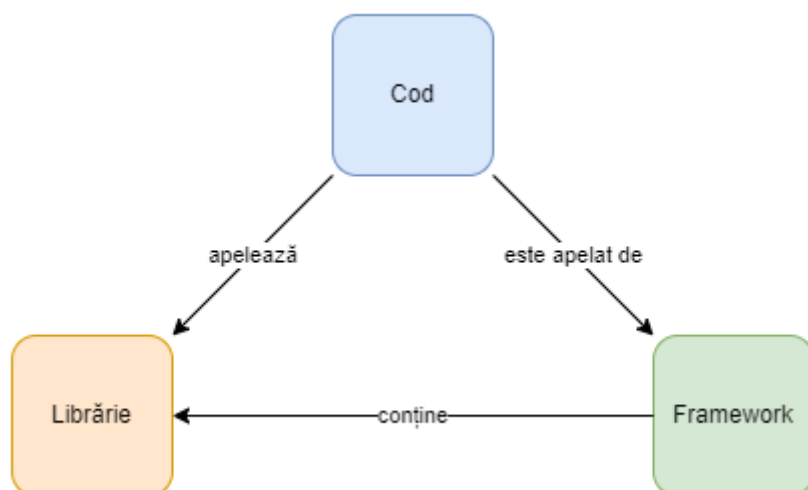


Fig. 7: Framework vs librărie

Așadar, conceptul de framework poate fi văzut ca o extindere al conceptului de librărie. Principalul avantaj al utilizării unui framework este acela al decuplării ordinii de execuție a codului de implementarea acestuia de către dezvoltatorul software. Astfel, crește modularitatea programului iar scalabilitatea este mai ușor de gestionat.

DI (Dependency Injection) reprezintă o tehnică prin care este implementat conceptul de IoC în Spring. Practic, prin injectarea unor obiecte cu alte obiecte, se înțelege setarea valorii unui parametru specific. [4]

În programarea tradițională, dacă am vrea să injectăm un obiect în altul, ar trebui să apelăm metoda de implementare a obiectului injectat din obiectul în care se dorește injectarea.

În framework-ul Spring, injectarea dependențelor se poate face mai simplu, în trei moduri: prin constructor, prin setter sau prin parametru. În toate cazuri, este folosită adnotarea „@Autowired”.

```
@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Autowired
    private EmployeeService (EmployeeRepository employeeRepository) {

        this.employeeRepository = employeeRepository;
    }

    @Autowired
    private void setEmployeeRepository (EmployeeRepository employeeRepository) {

        this.employeeRepository = employeeRepository;
    }
}
```

Fig. 8: Injectarea dependențelor

În exemplul de mai sus, este injectat obiectul de tip repository în cel de tip service. În prima situație este prezentată injectarea dependențelor prin parametru. Aceasta este cea mai simplă de implementat abordare și prin care este definit obiectul ca atribut al clasei, precedat de adnotarea „@Autowired”.

În cazul al doilea este prezentată injectarea dependențelor prin constructor. Această abordare presupune ca containerul Spring să apeleze un constructor în care argumentele sunt reprezentate de dependențele care se doresc a fi injectate.

În cel de-al treilea caz, este ilustrată injectarea dependențelor prin intermediul unei metode de tip setter. Asemănător injectării prin constructor, este creată o metodă ce primește ca parametru dependențele. Container-ul va apela mai întâi un constructor fără argumente pentru a inițializa bean-ul, apoi va putea fi apelată metoda setter.

Este posibil să fie combinate injectarea dependențelor prin constructor cu cea a injectării prin setter-i. O practică bună este aceea a utilizării constructorilor pentru dependențele

obligatorii, iar setter-i să fie utilizați pentru dependențele opționale. Echipa de dezvoltare a Spring recomandă folosirea constructorilor, deoarece astfel pot fi implementate componente ale aplicației ca și componente imutabile și se asigură faptul că acele componente nu sunt nule. Totodată, componentele sunt mereu întoarse inițializate. Deși este cea mai ușoară de implementat, injectarea dependențelor direct prin parametru ar trebui evitată, deoarece este mai costisitoare decât implementarea prin metode. Aceasta este des întâlnită în cazul claselor cu multe dependențe, dar aceste cazuri ar trebui evitate, clasa având prea multe responsabilități de gestionat.

Framework-ul permite ca injectarea dependențelor să se realizeze automat, prin declararea acestora într-un fișier de configurare, ceea ce permite container-ului spring să lege singur relațiile dintre bean-uri. Pentru a realiza asta, Spring folosește adnotarea „*@Configuration*”. În schimb, Spring Boot introduce adnotarea „*@SpringBootApplication*”, pentru a activa injectarea automată a dependențelor. Aceasta este echivalentă cu folosirea adnotării precedente, plus a celor de „*@EnableAutoConfiguration*” și „*@ComponentScan*”. Atunci când întâlnește această adnotare, framework-ul scanează automat pentru a detecta existența tuturor bean-urilor.

```
@SpringBootApplication
public class AuthenticationApplication {

    public static void main(String[] args) {
        SpringApplication.run(AuthenticationApplication.class, args);
    }

}
```

Fig. 9: Inițializarea DI

3.3.3. Container IoC

Containerele IoC reprezintă caracteristica de bază a framework-urilor care implementează conceptul de inversiune a conrolului. Containerul are rolul de a crea obiectele, de a le lega împreună, de a le configura și de a gestiona durata lor de viață. În cazul framework-ului Spring, containerul Ioc Spring este instanțierea, configurarea și asamblarea bean-urilor.

Toate informațiile de configurare, sub formă de metadata, pe care containerul le primește despre obiectele pe care le gestionează sunt transmise prin intermediul adnotărilor. [4]

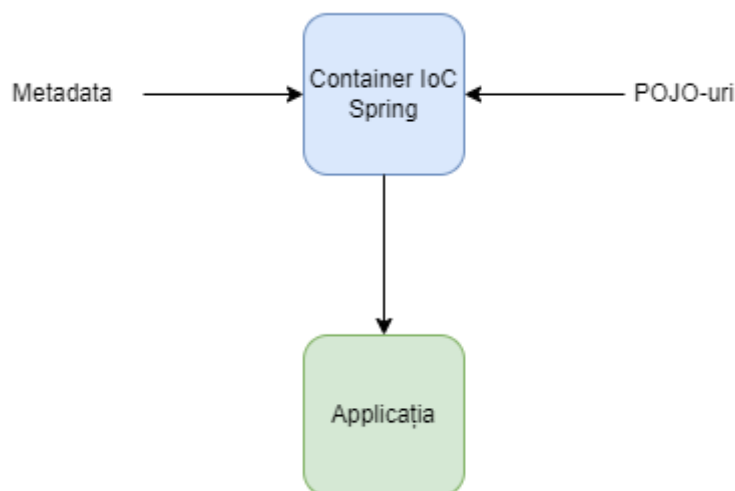


Fig. 10: Container Ioc

De multe ori, când se vorbește despre containere Ioc Spring, acestea mai sunt numite și context Spring. În descriere pe scurt, Spring Context reprezintă o zonă de memorie a aplicației în care sunt adăugate instanțe ale tuturor obiectelor pe care framework-ul le gestionează. Instanțele obiectelor adăugate în context sunt numite bean-uri. Nu toate obiectele din cadrul aplicației sunt gestionate de framework.

Containerul IoC Spring este reprezentat de interfețele BeanFactory și ApplicationContext. Prima este interfața de bază pentru accesarea containerului Spring și include metode pentru gestionarea bean-urilor. Cea de a doua este o sub-interfață a acesteia, ce conține metode în plus. Este recomandată folosirea în general a interfeței ApplicationContext.

3.3.4. Bean

În cadrul framework-ului Spring, bean-urile sunt unitatea structurală de bază. Bean-urile reprezintă obiectele cu care lucrează containerul Spring. Întreg ciclul de viață al acestor obiecte este controlat de către container. [4]

Metoda de bază pentru a utiliza un bean presupune mai întâi crearea unei clase adnotată cu „@Component”, care va sta la bază bean-ului. Apoi, este necesară adnotarea unei clase de configurare cu „@Configuration”. În interiorul acestei clase este definită o metodă adnotată cu „@Bean”, care va întoarce o instanță a unui obiect care va fi transformat într-un bean. În plus,

clasa de configurare este adnotată și cu „*@ComponentScan*”, pentru a indica scanarea aplicației de bean-uri.

Totuși, de cele mai multe ori în practică, sunt folosite adnotări stereotipice. Acestea reprezintă adnotări folosite de framework pentru a crea în mod automat bean-uri cu scopuri specifice în context. „*@Component*” reprezintă principala adnotarea stereotipică din framework-ul Spring. Din aceasta sunt derivate majoritatea celorlalte adnotări, precum:

- *@Service*: clasele adnotate cu „*@Service*” sunt cele care conține logica de business a aplicației. În general, serviciile sunt apelate de către controllere, iar la rândul lor apelează mai departe repository-urile. Serviciile nu trebuie să poată fi apelate din afara aplicației și nici nu trebuie să aibă acces la baza de date. Rolul lor este stric de a prelucra datele primite, acționând ca un layer intermediar între controllere și repository-uri.
- *@Repository*: în general, fișierele cu această adnotare sunt interfețe și sunt cele care fac legătura cu baza de date. Acestea conțin logica de scriere și citire a datelor. Asemenea serviciilor, repository-urile, nu trebuie să fie direct accesate din exterior și nici nu trebuie să prelucreze date. Scopul lor unic este să fie apelate de servicii și să efectueze operația cerută asupra bazei de date.
- *@Controller*: clasele cu adnotarea „*@Controller*” sunt clasele care realizează legătura dintre aplicație și exterior. În interiorul acestor clase sunt definite endpoint-urile, care sunt apelate de către alte aplicații. Controlerele nu manipulează date și nu au acces la baza de date, scopul lor unic fiind acela de a primi cereri de tip HTTP și de a returna răspunsurile necesare.

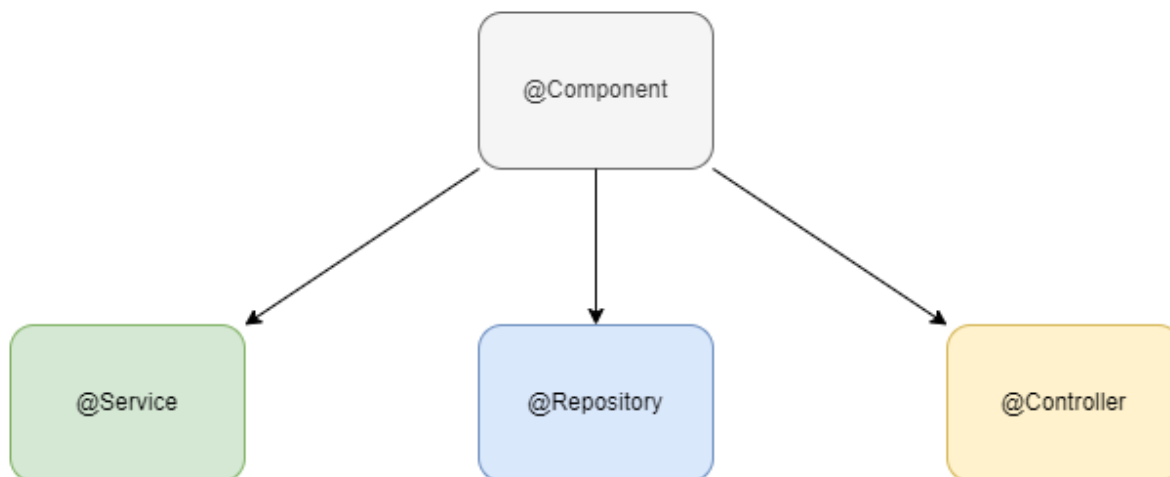


Fig. 11: Adnotări stereotipice

3.3.5. Scopuri Bean

Atunci când este definit un bean, este definit un mod de a crea instanțe ale claselor definite de bean-ul respectiv. Asta înseamnă că pot fi create mai multe instanțe ale aceluiași bean, folosind aceeași definiție. Această caracteristică a bean-urilor oferă o largă flexibilitatea framework-ului, deoarece definește mai multe posibilități în care poate fi folosită o clasă. În implementarea tradițională a codului, fără bean-uri, ar fi fost necesară redefinirea clasei pentru fiecare scenariu de utilizare.

Bean-urile pot fi definite utilizând unul din scopurile posibile. Framework-ul Spring suportă cinci astfel de scopuri: [4]

- singleton;
- prototype;
- request;
- session;
- application.

Singleton este scopul de bază utilizat de framework-ul Spring. Acesta presupune că este creată o singură instanță a bean-ului este creată și este partajată în întreaga aplicație. Orice cerere care se potrivește descrierii bean-ului respectiv, primește înapoi ca răspuns aceeași instanță a bean-ului respectiv. Atunci când un bean este definit ca fiind de tip singleton, containerul Spring

crează o singură instanță a acestuia, definită de către bean. Instanța este stocată în memoria cache a aplicației, iar toate cerințele referitoare la bean lucrează cu aceeași instanță.

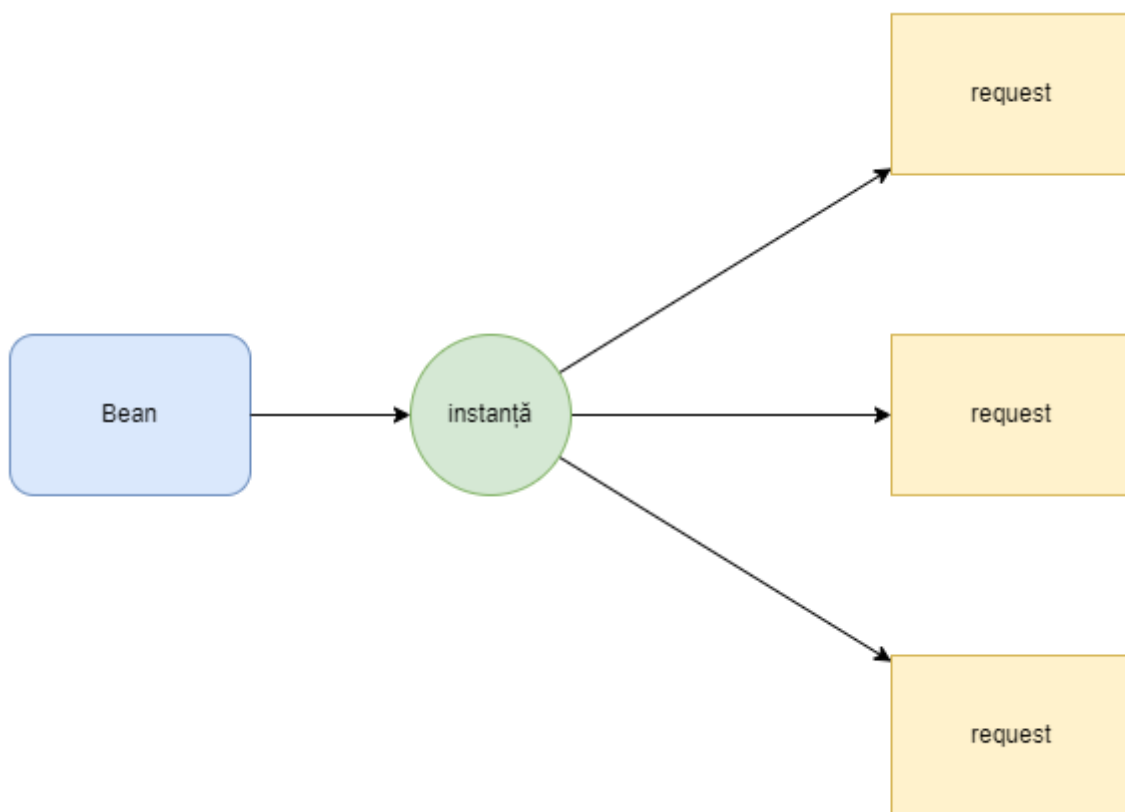


Fig 12: Bean Singleton

Prototype este scopul opus Singleton. Acesta presupune ca la fiecare cerere făcută către un bean, să fie creată o instanță nouă a clasei respective. De regulă, scopul prototype este folosit pentru bean-urile de tip stateful, iar singleton este folosit pentru bean-urile stateless. În cazul bean-urilor prototype, containerul Spring creează o instanță a acestora și o trimite mai departe clientului care a făcut cererea, fără a păstra vreo informație despre starea acestuia. Astfel, ciclul de viață al acestor bean-uri nu mai este gestionat de către aplicația server, distrugerea lor fiind responsabilitatea clientului. Pentru majoritatea claselor, se recomandă utilizarea bean-urilor singleton, al căror ciclu de viață este determinat de durata sesiunii. Pe de altă parte, bean-urile prototype ajută la scalabilitatea aplicației de-a lungul tuturor request-urilor.

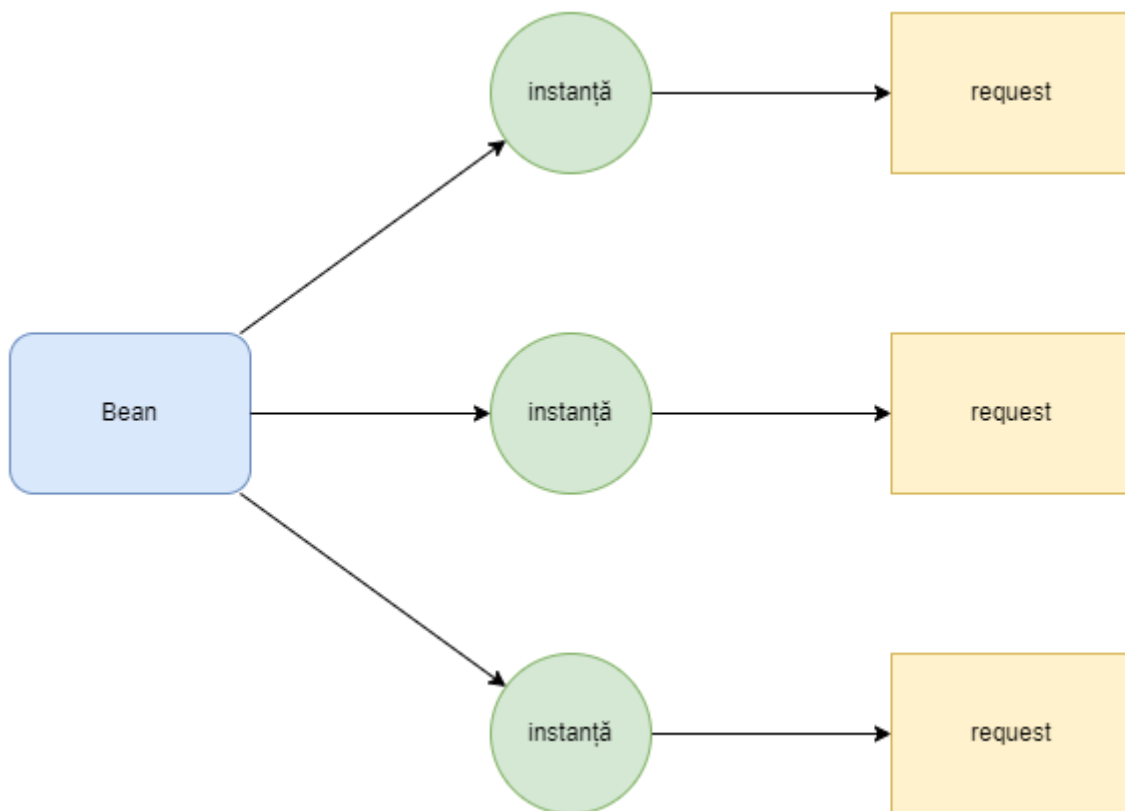


Fig. 13: Bean Prototype

Scopurile Request, Session și Application pot fi utilizate doar atunci când este implementată aplicația `ApplicationContext`. Aceasta este web-aware, adică aplicația conține endpoint-uri care pot fi apelate de către aplicații client, cum este în cazul aplicațiilor REST. În cazul Spring Web MVC, orice cerere primită de aplicație este gestionată de `DispatcherServlet`.

Acesta are rolul de a acționa ca un layer înaintea controller-ului. Orice request primit de aplicație ajunge mai întâi la `Dispatcher Servlet`, care înaintează apoi cererea către controller-ul în care se găsește implementat endpoint-ul dorit. Acesta întoarce apoi modelul dorit și specifică ce view trebuie să afișeze acel model. Modelul este afișat utilizatorului, iar o dată cu acesta este trimis răspunsul înapoi către browser de către `Dispatcher Servlet`.

`Servlet`-urile sunt o componentă esențială în dezvoltarea aplicațiilor web bazate pe limbajul de programare Java. Rolul acestora este de a gestiona partea de comunicare a aplicațiilor web server.

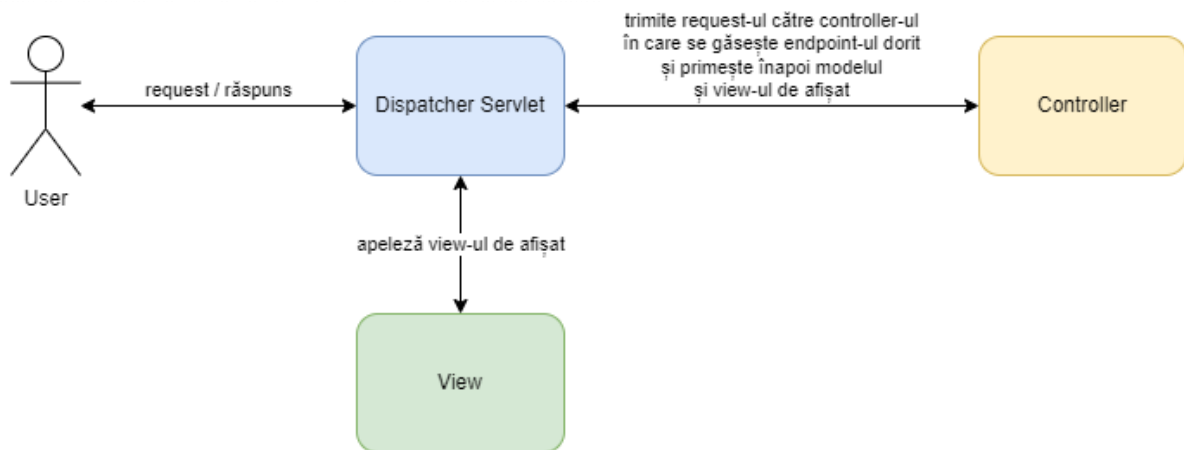


Fig. 14: Dispatcher Servlet

Request reprezintă bean-uri al căror existență este legată de cererile primite de aplicație. Aceste bean-uri sunt unice pentru fiecare cerere în parte, iar după executarea acestora, ele sunt distruse automat. Un exemplu de utilizare al acestor tip de bean-uri ar fi în cazul unei instanțe de autentificare în aplicație. Singurul lor scop este acela de a realiza autentificarea. Astfel, aceste bean-uri sunt, așa cum spune numele, doar la nivel de request.

Scope este un alt tip de bean-uri al căror existență este în strânsă legătură cu protocolul HTTP. Acestea sunt create atunci când o sesiune HTTP este creată între aplicație și client, iar durata lor de viață este dată de durata sesiunii. Asemenea bean-urilor de tip Request, bean-urile de tip Scope au avantajul de a rămâne la fel, indiferent de modificarea instanței bean-urilor, rolul lor unic fiind ce a gestiona sesiunile. Acest tip de bean-uri pot fi catalogate ca fiind la nivel de sesiune a utilizatorilor, deci cu o durată de viață mai lungă decât precedenta categorie.

Application este cel de-al treilea scop de bean-uri din această categorie și are perioada de viață cea mai lungă dintre acestea. Un astfel de bean este definit la nivelul întregii aplicații, mai exact la nivel de servlet. Poate fi asemănat din acest punct de vedere cu bean-urile de tip Singleton, diferența fiind că o aplicație poate avea mai multe servlet-uri, astfel Singleton având o arie de acoperire mai largă. Față de Request sau Session, Application are durata de viață cea mai lungă, bean-urile fiind distruse numai când execuția aplicației este terminată.

Spring oferă și posibilitatea definirii unor scopuri proprii pentru bean-uri sau chiar redefinirea unor scopuri deja existente. Totuși, acest lucru nu este recomandat, deoarece poate afecta restul bean-urilor din aplicație.

3.3.6. Aspecte

Pe lângă injectarea dependențelor, al doilea concept de bază al framework-ului Spring este reprezentat de Programarea Orientată pe Aspecte (AOP). Aceasta reprezintă o extensie a paradigmei Programării Orientate pe Obiecte (OOP). Dacă în OOP, accentul era pus pe clase ca stând la baza aplicațiilor, AOP pune accentul pe aspecte. Ca și DI, unde scopul era de a decupla obiectele unele de altele, pentru a fi gestionate de către framework, scopul AOP este acela de a decupla concern-urile cross-cutting față de obiectele asupra cărora acționează. Din acest punct de vedere, AOP poate fi asemănat cu trigger-ii din PL/SQL. În general, aspectele Spring se referă la tranzacții, securitate sau caching. [4]

Conceptele cheie ale AOP sunt: [4]

- Obiectele țintă: reprezintă obiecte ce conțin metode ale căror funcționalități sunt modificate de către aspecte. Aceste obiecte mai sunt numite și „sfătuite”;
- Aspectele: reprezintă comportamentul framework-ului Spring atunci când obiectele țintă sunt apelate. Aspectele acționează asupra mai multor clase, unele din ele adnotate cu „*@Aspect*”;
- Advice-urile: sunt cele care descriu modul și momentul în care sunt executate aspectele de către Spring. În timpul execuției unui program, sunt definite mai multe momente, precum execuția unor metode, tratarea unei erori etc. Advice-urile indică în care din aceste momente este declanșat un aspect. Advice-urile pot fi de mai multe tipuri, adică în timpul execuției, înainte sau după execuția metodei respective. Advice-urile pot fi văzute ca un interceptor.
- Pointcut-urile: sunt exact metodele pe care framework-ul Spring trebuie să le intercepteze. Pointcut-urile sunt metodele care conține momentele în care sunt declanșate advice-urile.
- Weaving: reprezintă legăturile create de Spring între aspecte și obiectele asupra cărora acționează. Astfel, se ajunge ca rezultat la „obiecte sfătuite”, exact scopul paradigmei de AOP.

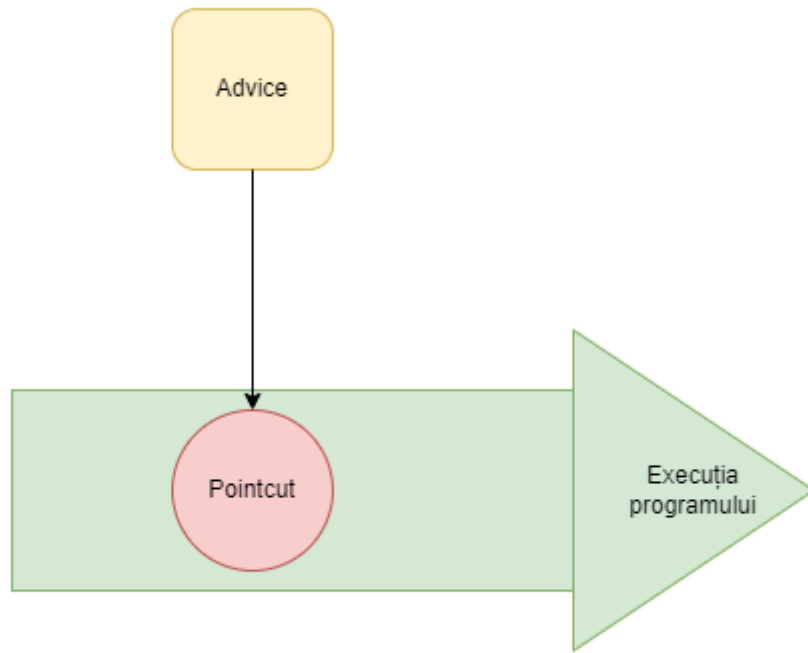


Fig. 15: Aspect

Tipurile de advice-uri AOP incluse în framework-ul Spring sunt:

- **@Before:** advice-urile sunt executate înainte de a fi executată metoda interceptată, dar nu previne și execuția acesteia, mai puțin în cazul tratării unei erori;
- **@After:** advice-urile sunt executate după ce metoda este executată, chiar și dacă are loc tratarea unei erori în cadrul metodei;
- **@AfterThrowing:** advice-urile sunt executate doar dacă metoda aruncă o eroare;
- **@AfterReturning:** advice-urile sunt executate doar dacă metoda a fost executată cu succes;
- **@Around:** advice-urile sunt executate fie la înainte, fie în timpul execuției metodei, sau metoda nu mai este executată de loc și este executat alt cod, de ex o tratarea unei erori.

3.3.7. Persistența datelor

3.3.7.1. Data Access Object

DAO (Data Access Object) reprezintă un pattern care presupune separarea logicii de business de persistența datelor. Acesta presupune utilizarea unui API care să preia toate cererile către baza de date. [7]

Repository este un alt pattern, cu rol asemănător cu cel al DAO. La fel ca acesta, are rolul de gestiona operațiile CRUD care sunt executate asupra bazei de date. Totuși, Repository este considerat ca fiind la un nivel mai sus decât DAO în ierarhia aplicației. Pentru a gestiona colecțiile de obiecte, un repository poate folosi mai multe clase DAO.

DAO poate fi definit ca un layer de abstractizare al persistenței datelor, în timp ce Repository poate fi definit ca un layer de abstractizare al colecțiilor de obiecte. Spring este dezvoltat astfel încât cele două pattern-uri să poată fi folosite în mod simplu cu tehnologie de accesare a datelor, cum sunt JDBC, Hibernate sau JPA.

3.3.7.2. Java Database Connectivity

Principala tehnologie de conectare cu o bază de date SQL în Java este reprezentată de către Spring Data JDBC (Java Database Connectivity). Acesta este un API care se bazează pe interogări SQL pentru a comunica cu baza de date. Astfel, JDBC reprezintă cel mai apropiat nivel al aplicației de baza de date și permite cel mai mult control în interacțiunea cu aceasta.

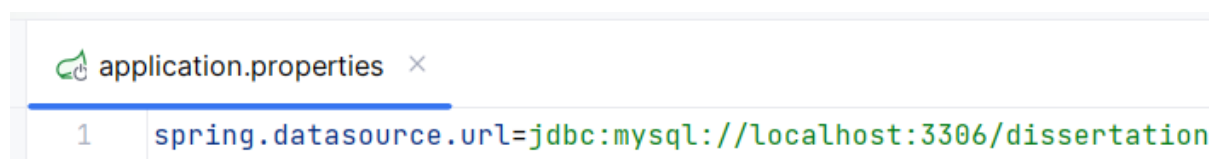


Fig. 16: Conectarea la baza de date

Pentru a se conecta la baza de date, JDBC folosește un URL. Acesta conține informațiile necesare despre bază și cum poate fi aceasta accesată. Mai întâi este definit protocolul prin care se realizează conexiunea. Apoi host-ul bazei de date, în cazul de mai sus fiind vorba despre o bază MySQL. Urmează adresa URL către care se pot face request-urile către bază, iar la final numele acesteia, în cazul MySQL, numele schemei.

Dezavantajul principal al JDBC îl reprezintă numărul mare de linii de cod necesare pe care trebuie să le scrie dezvoltatorii pentru a putea realiza operațiunile asupra bazei. În plus, JDBC necesită să fie tratate toate excepțiile care pot apărea în timpul executării cererii. Prima dată trebuie verificat dacă există conexiunea către baza de date. Apoi trebuie creată un statement care v-a reține detaliile query-ului SQL și care la final este executat și tratate erorile care pot apărea. La final, trebuie închise atât statement-ul cât și conexiunea, pentru a nu încărca inutil aplicația.

3.3.7.3. Object Realtional Mapping

ORM (Object Realtional Mapping) este o tehnică de programare al cărei scop este să facă legătura dintre programarea orientată pe obiecte și bazele de date. În Spring, acesta acționează ca un layer suplimentar între JDBC și restul aplicației. ORM-urile scurtează semnificativ durata de dezvoltarea codului, eliminând o bună parte din logica de gestionare a conexiunii la baza de date pentru fiecare interogare în parte. Printre cele mai populare astfel de tehnologii, în Spring se regăsesc Hibernate și JPA. ORM-urile oferă o mai bună implementare a conceptelor de IoC și AOP, care stau la baza Spring.

3.3.7.4. Hibernate

Hibernate este un framework de persistență a datelor open-source. Acesta oferă toate caracteristicile de bază ale unui ORM. Hibernate permite maparea obiectelor java pentru a putea fi apoi înțelese de către baza de date. Astfel este redus mult din codul care trebuie scris de dezvoltatori.

Printre caracteristicile principale aduse de Hibernate se numără: [7]

- Lazy loading: permite ca atunci când este citit un obiect din baza de date, să nu fie aduse toate detaliile despre acesta. Este o caracteristică utilă în special în cazul obiectelor care conțin colecții de alte obiecte;
- Eager fetching: este opusul metodei de mai sus. Acestea permite ca toate atributele unui obiect să fie citite din baza de date. Astfel, nu mai trebuie interogări suplimentare și pentru restul obiectelor din colecții;

- Cascading: permite ca atunci când se execută modificări asupra unui obiect care are în atributele sale și alte obiecte, modificările să se facă simultan asupra tuturor tabelelor aferente din baza de date.

În majoritatea aplicațiilor moderne Spring Boot, Hibernate reprezintă principala formă de interpretarea JPA.

3.3.7.5. Java Persistence API

JPA (Java Persistence API) este o librărie ce conține metode pentru a defini modul în care datele sunt salvate în baza de date. Față de Hibernate, care este un framework în sine, JPA este doar un layer suplimentar între un framework ORM și restul aplicației. Hibernate este cea mai populară implementare a JPA, cele două fiind folosite împreună ca o singură tehnologie în majoritatea aplicațiilor Spring Boot. Practic, implementarea JPA folosind Hibernate creează un ORM în sine. [7]

Pentru a funcționa, JPA se folosește de entități. Acestea nu reprezintă decât niște clase POJO (Plain Old Java Object), adnotate cu „@Entity”. Astfel, JPA va crea în baza de date o tabelă corespunzătoare pentru fiecare entitate în parte.

```

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="user_type", discriminatorType = DiscriminatorType.STRING)
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @NotNull(message = "Username cannot be null!")
    private String userName;
    private String password;
    private boolean enabled;
    private String firstName;
    private String lastName;

    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.PERSIST)
    @JoinTable(name = "user_roles",
        joinColumns = {
            @JoinColumn(name = "user_id")
        },
        inverseJoinColumns = {
            @JoinColumn(name = "role_id")
        }
    )
    private Set<Role> roles;
}

```

Fig. 17: Entitatea Utilizator

În figura de mai sus este ilustrată entitatea „User”, pentru care JPA va crea în baza de date tabela corespunzătoare „users”. Atributele clasei vor reprezenta coloane în baza de date. Orice entitate JPA are nevoie de o cheie primară pentru a fi identificată în mod unic, la fel ca entitățile stocate într-o bază de date relațională. Astfel, în clasa „User” este definit atributul „id”, adnotat cu „@Id”. Adnotarea „@GeneratedValue” permite ca cheia primară să fie generată automat de către JPA. Astfel, cheia v-a fi mereu unică, fără a fi necesare acțiuni suplimentare de implementat în cod mai târziu.

După cum spune și denumirea lor, bazele de date relaționale folosesc pe relațiile dintre tabele pentru a reprezenta legăturile dintre date. JPA are capacitatea să gestioneze și acesta aspect, relațiile dintre entități reprezentând și relațiile din bază. În figura de mai sus este definită o relație de tip many-to-many între entitățile „User” și „Roles”. Un utilizator poate avea mai

multe roluri, iar mai mulți utilizatori pot avea același rol. Adnotarea „@JoinTable” permite JPA să creeze o tabelă auxiliară în baza de date, pentru a reprezenta relația many-to-many sub forma a două relații de tip one-to-many. Pentru realizarea interogărilor către baza de date, au fost folosite și două funcționalități ale Hibernate. Prima este cea de Eager fetch, deoarece este nevoie și de rolul utilizatorului atunci când se face autentificarea. A doua este cascadare persistențelor, astfel încât să fie inserat automat și un rând în tabela auxiliară, atunci când este setat rolul unui utilizator. Alte tipuri de cascadări nu s-au folosit, deoarece ar fi dus la alterări nedorite ale datelor. De exemplu, în cazul ștergerii unui utilizator, ar fi putut fi șters din greșeală și rolul acestuia, chiar dacă alți utilizatori aveau același rol.

Entitatea „User” mai are o adnotare care specifică un discriminator în cazul moștenirii clasei. În mod automat, moștenirea în JPA este de tip Single Table. Astfel, pentru entitățile derivate din clasa de bază, nu se vor crea tabele suplimentare. Toate atributele lor vor fi afișate într-un singur super-tabel din baza de date. JPA v-a crea o coloană adițională în tabelă, în care va preciza, pentru fiecare rând inserat, cărei entitate îi corespunde. Aceasta valoarea este folosită pentru a identifica carei entităţi îi corespunde fiecare obiect.

3.3.7.6. Spring Data JPA

Spring Data JPA reprezintă încă un layer în plus în arhitectura aplicației, peste JPA. Acesta elimină de tot implementarea claselor DAO sau a repository-urilor, reducându-le pe cele din urmă la o simplă interfață. Spring Data constă într-o serie de interfețe de tip repository deja definite, pe care repository-urile aplicației le pot extinde. Astfel, este eliminată nevoia de a mai implementa metodele CRUD de bază, pe care se presupune că majoritatea aplicațiilor le folosesc. [7]


```

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    @Query("SELECT e FROM Employee e WHERE e.userName = :userName")
    Optional<Employee> findByUserName(String userName);

    1 usage
    List<Employee> findAllByEnabled(Boolean enabled);

}

```

Fig. 18: Repository Angajat

Interfețele definite de dezvoltatori sunt adnotate cu „@Repository”. În exemplul de mai sus, interfața „EmployeeRepository” extinde interfața „JpaRepository”, care primește ca parametri entitatea asupra căreia acționează și identificatorul acesteia. În interiorul interfeței au fost definite doar metode suplimentare de executat asupra bazei de date. Se observă că nici query-urile nu mai trebuie scrise, acestea fiind generate automat pe baza numelui metodelor. Totuși se pot defini în continuare query-uri, atunci când se dorește mai mult control asupra interogărilor.

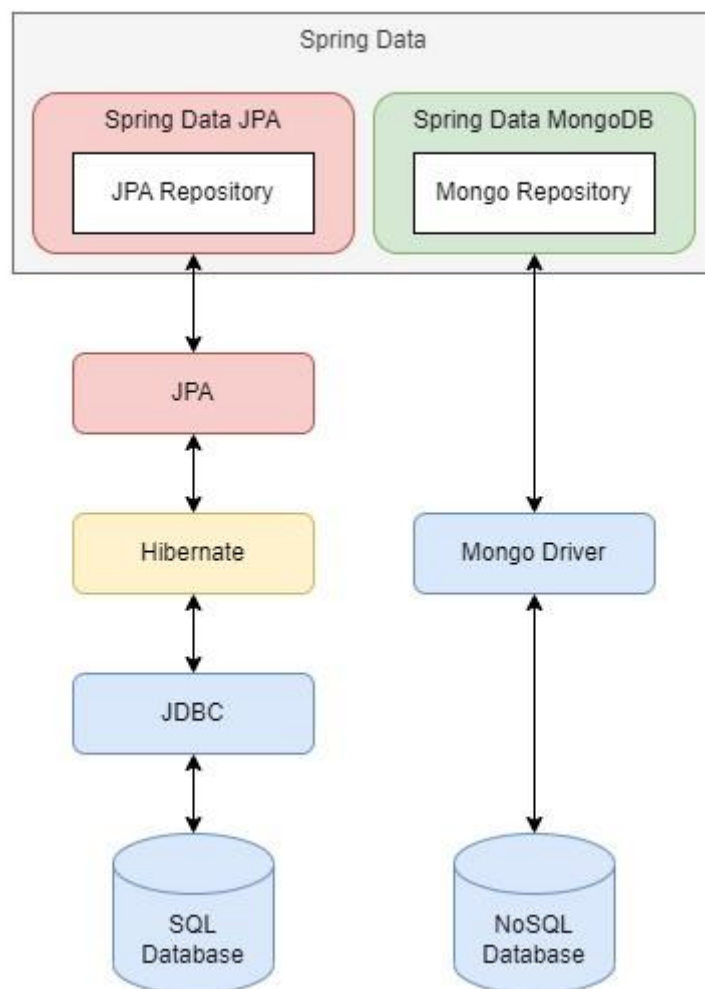


Fig. 19: Arhitectura Spring Data

3.3.8. Spring MVC

Principalele aplicații web dezvoltate la început în Spring se bazează pe arhitectura MVC. Model-View-Controller (MVC) poate fi definit drept un design pattern ce se bazează pe separarea unei aplicații în trei componente logice: Model, Controller și View. Principiul pe care se bazează această arhitectură este numit „Separation of concerns”. Astfel, fiecare componentă deservește roluri specifice și diferite în dezvoltarea aplicației:

Controller-ul: este componenta principală a aplicației și are rolul de a conecta toate celelalte componente. Controller-ul este responsabil pentru logica de business a aplicației. Acesta primește request-urile din partea view-urilor, manipulează datele folosind modelele, și alege care view va afișat în continuare utilizatorului.

Modelul: are rolul de a defini structura informației cu care lucrează aplicația. Informația este astfel structurată pentru a fi transferată între view-uri și controller-e. Modelele sunt clasele de bază ale aplicației și conțin doar metode de scriere și citire a datelor. Acestea sunt folosite pentru a comunica cu baza de date.

View-ul: gestionează interacțiunea utilizatorului cu aplicația. Acesta gestionează logica de front-end și are rolul de a afișa elementele în pagină, generând interfața utilizatorului (UI). Informațiile conținute de modele sunt afișate în view-uri prin intermediul controller-elor.

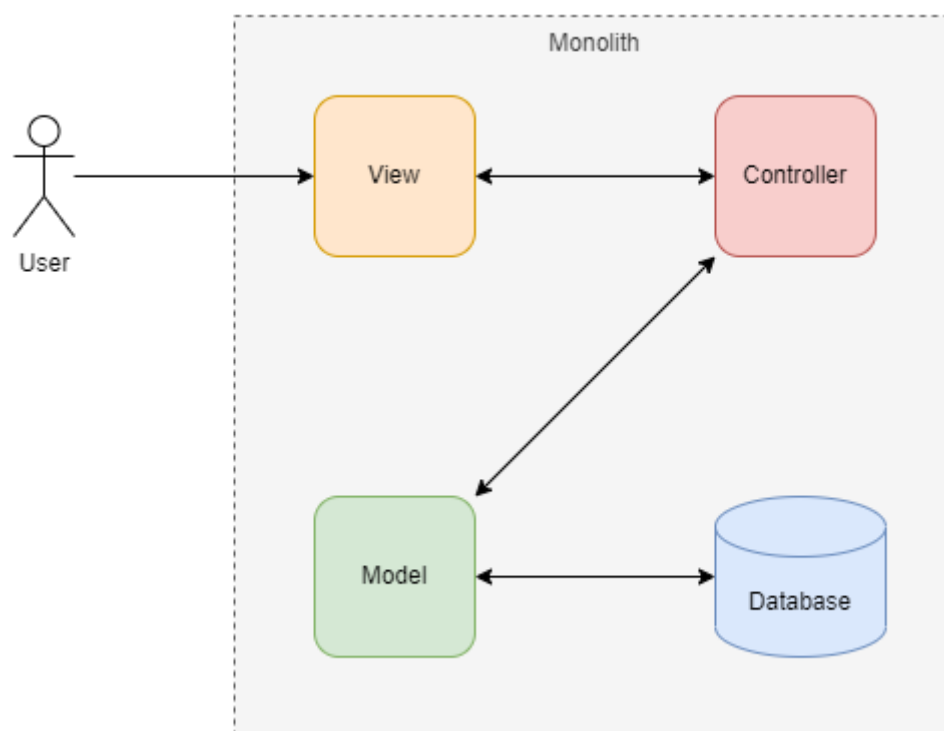


Fig. 20: Arhitectura generală MVC

3.3.9. Servicii web

Serviciile web reprezintă aplicații web care funcționează pe baza unui set de protocoale ce permit schimbul de date cu alte aplicații. Principalul avantaj al serviciilor web îl reprezintă faptul că, o dată ce au fost create, acestea pot fi refolosite de mai multe ori, de diferite aplicații, fără a mai fi necesar efortul de a repeta logica din spatele acestora.

3.3.10. REST API

API (Application Programming Interface) permite ca două sau mai multe aplicații să comunice între ele prin intermediul unor protocoale standard. Cele mai populare cazuri de utilizare al API-urilor este reprezentat de aplicațiile bazate pe arhitectura client-server. Există mai multe tipuri de API-uri, cel mai des utilizate fiind cele bazate pe protocolul REST (REpresentational State Transfer).

REST este un standard web care se bazează pe protocolul HTTP (HyperText Transfer Protocol). Popularitatea REST este datorată simplității sale și a conceptului de resource-based, interacțiunile REST fiind bazate pe protocoale deja existente în domeniul web. Statusul acestor interacțiuni este comunicat prin intermediul unor coduri numerice, folosite pentru a detecta erori:

- 100-199: răspuns informativ. De obicei când cererea a ajuns, dar nu conține și un corp, indicând că acesta urmează să vină;
- 200-299: succes. Indică că cererea a fost primită și procesată. Cel mai des utilizat este statusul 200 ok;
- 300-399: redirecționare. Este un status intermediar, care indică necesitatea unor alte acțiuni din partea clientului;
- 400-499: eroare client. Indică faptul că a fost produsă o eroare în cerere de către client. De exemplu, în cazul codului 400, datele au fost introduse incorect. În cazul 404, resursa căutată de către client nu a fost găsită.
- 500-599: eroare server. Este cea mai gravă dintre erori, atunci când aplicația server nu este capabilă să întoarcă un răspuns aplicației client. Aceste erori trebuie evitate întotdeauna și trebuie tratate cazurile de excepție.

Protocolul HTTP utilizează o colecție de metode de request pentru a determina ce tip de interacțiune are loc între aplicația client și cea server. Cele mai des folosite astfel de metode sunt cel de GET, POST, PUT și DELETE. [12]

- GET: această metodă este folosită pentru a obține date de la aplicația server, folosind un URI (Uniform Resource Identifier). Metoda GET nu trimite date către server;
- POST: este metoda prin care sunt trimise date către server, în general atunci când este creată o resursă nouă. Datele sunt de obicei structurate în corpul cererii, de obicei în format JSON (JavaScript Object Notation);

- PUT: această metodă este folosită pentru a actualiza resurse deja existente. Poate fi văzută ca o combinație între metodele GET și POST, deoarece atât primește, cât și trimite informații serverului. Asemănător metodei precedente, metoda PUT poate conține datele într-un body sau poate folosi parametri;
- DELETE: metoda are rolul de a șterge resurse. Asemănător metodei GET, aceasta nu trimite date noi către server.

3.3.11. Controller REST

Pentru a gestiona toate request-urile venite de la client, framework-ul Spring folosește conceptul de Front Controller, implementat sub forma Dispatcher Servlet. Acesta a fost prezentat într-un capitol anterior, când am vorbit despre ciclul de viață al bean-urilor. Dispatcher Servlet este folosit atât în cazul controller-elor MVC, cât și în cazul celor REST. Față de controller-ele MVC, cele REST nu mai întorc și un view, ci întorc doar datele sub formă de JSON.

```
@RestController
@RequestMapping("/employees")
@CrossOrigin
@Validated
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/{id}")
    @PreAuthorize("hasRole('admin') or hasRole('employee')")
    public ResponseEntity<EmployeeDto> getById(@PathVariable Long id) {

        return ResponseEntity.ok().body(employeeService.getById(id));
    }
}
```

Fig. 21: Controller Angajat

Controller-ele sunt adnotate cu „@RestController” și se folosesc de servicii ca dependențe pentru a trimite datele primite mai departe spre prelucrare. În figura de mai sus, este prezentat un endpoint prin care aplicația client dorește să primească înapoi angajatul cu un anumit id, transmis ca parametru în url-ul cererii și folosind „@PathVariable” pentru extragerea acestuia.

3.3.12. Tratarea excepțiilor

Nu întotdeauna request-urile primite de la aplicația client pot fi executate cu succes de aplicația server. Așa cum am prezentat mai sus, serviciile Web REST folosesc protocolul HTTP pentru a comunica cu alte servicii. Am precizat că acesta se bazează coduri numerice pentru a indica dacă interacțiunea dintre servicii s-a finalizat cu succes sau care este statusul acesteia.

Pentru a întoarce răspunsul HTTP corect, framework-ul Spring oferă suport pentru gestionarea acestor răspunsuri și pentru interceptarea erorilor apărute pe parcursul executării task-urilor. În figura în care este prezentat Controller-ul Angajat, se poate observa cum metodele din acesta întorc un obiect de tip `ResponseEntity`. Această clasă reprezintă un răspuns HTTP în întregime, obiectele sale conținând atât status-ul request-ului cât și datele întoarse în caz de succes.

```

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler({
        ResourceNotFoundException.class,
        BadRequestException.class,
        DisabledException.class,
        BadCredentialsException.class,
        UsernameNotFoundException.class,
        RuntimeException.class,
        Exception.class,
        ExpiredJwtException.class,
        AuthenticationException.class,
        IOException.class,
        ServletException.class,
        Exception.class
    })
    public ResponseEntity handle(Exception e) {

        return ResponseEntity.badRequest().body(e.getMessage());
    }
}

```

Fig. 22: Tratarea excepțiilor

Pentru a intercepta erorile apărute în timpul execuției codului, Spring oferă mai multe metode, cea mai populară la ora actuală fiind utilizarea „@ControllerAdvice”. Acesta reprezintă un mod unitar de a centraliza toate excepțiile pe care aplicația le poate arunca. Adnotarea permite modelarea răspunsului dorit în caz de eroare și oferă posibilitatea ca mai multe excepții să ofere același tip de răspuns.

Tratarea excepțiilor este posibilă datorită conceptului de AOP, concept ce stă la baza Spring și a fost prezentat într-un din capitolele anterioare.

3.3.13. WebClient

Pentru comunicarea între microservicii a fost folosită interfața WebClient. Aceasta reprezintă principalul mod de a accesa endpoint-urile altor servicii. Interfața permite declararea url-ului aferent endpoint-ului, a headere-lor necesare (cum ar fi cel de autentificare în imaginea de mai jos), cât și a json-urilor sau parametrilor necesari. Practic, un obiect de tip WebClient conține toate datele necesare unui request. Pentru a extrage datele din răspuns primit de la serviciul accesat, Spring oferă două clase: [6]

- Mono: folosit pentru a extrage o singură valoare din stream-ul de date primit. După extragerea unei valori, marchează parcurgerea ca fiind completă;
- Flux: folosit pentru a extrage întreaga secvență de date transmise prin intermediul stream-ului de date.

```
13 usages
private Employee getEmployeeById(Long employeeId, String header) {

    Employee employee = new Employee();
    try {
        WebClient webClient = WebClient.builder()
            .baseUrl("localhost:8080/employees/" + employeeId)
            .defaultHeader(HttpHeaders.AUTHORIZATION, header)
            .build();
        Mono<Employee> employeeMono = webClient.get().retrieve().bodyToMono(Employee.class);
        employee = employeeMono.block();
    } catch (Exception e) {
        throw new ResourceNotFoundException("Employee with id " + employeeId + " not found!");
    }

    return employee;
}
```

Fig. 23: WebClient

Interfața WebClient a fost introdusă o dată cu framework-ul Spring Web Reactive și aparține librăriei Spring Webflux. Programarea reactivă presupune crearea de aplicații de tip responsive, ușor scalabile, care să gestioneze mai eficient task-uri asincrone și concurente. Dacă aplicațiile server sunt cel mai des tip REST, aplicațiile client sunt reactive. Angular de exemplu se bazează pe observabile pentru a primi stream-uri de date de la server.

3.3.14. Spring Security

Spring Security este un framework din familie de proiecte Spring care gestionează logica de autentificare și autorizare a aplicației. Acesta are rolul de a proteja resursele serviciului web și de a permite accesul numai în cazul utilizatorilor autorizați. Există mai multe metode în care poate fi implementată securitatea aplicației, cea mai modernă abordare în cazul serviciilor REST fiind utilizarea token-urilor.

Numită și „autentificare la purtător” (Bearer Authentication), aceasta reprezintă o modalitate de autentificare a request-urilor HTTP prin utilizarea unor token-uri. Fiecare aplicație client primește după autentificare un token unic, pe care îl va folosi pentru a autoriza fiecare request pe care-l trimite către aplicația server. Token-urile reprezintă un șir de caractere criptat cu o cheie deținută de server. Acestea conțin informații despre utilizator precum username-ul său. Token-urile au o durată de viață limitată, așa că acestea trebuie reînnoite periodic prin realizarea din nou a autentificării. Fiind criptate, token-urile pot fi transmise public între serviciile web, deoarece în cazul unei interceptări, informațiile rămân sigure, iar până la o eventuală spargere a token-ului, acesta ar fi deja expirat și inutil.

Spring Boot permite în mod automat configurarea de bază a Spring Security, care introduce un bean cu rol de filtrare numit „SpringSecurityFilterChain”. Acesta are rolul de a lega toate filtrele de securitate folosite de aplicație și de a trece request-ul prin fiecare. Framework-ul implementează interfața „UserService”, ce conține metoda „loadByUsername”, ce are rolul de a culege datele necesare despre utilizator, precum username, parolă sau rolurile acestuia.

```
try {
    authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(userName, password));
} catch (DisabledException e) {
    throw new DisabledException("User is disabled!");
} catch (BadCredentialsException e) {
    throw new BadCredentialsException("Bad credentials from user!");
}
```

Fig. 24: AuthenticationManager

Autentificarea efectivă a unui utilizator este realizată prin intermediul interfeței „AuthenticationManager”. Aceasta conține metoda „authenticate” care primește ca parametrii datele de autentificare ale utilizatorului. Metoda compară datele primite cu cele existente în baza de date și confirmă dacă acestea se potrivesc. Clasa „User” definită în aplicație conține mai multe detalii despre acesta, cum ar fi dacă contul său încă mai este activ. Parolele stocate

în baza de date sunt criptate folosind interfața „PasswordEncoder”. În cazul în care metoda este executată cu succes și nu este aruncată nicio excepție, procesul de autentificare poate continua, prin crearea unei instanțe a „UserDetails”, ce va conține username-ul, parola și rolurile. Datele sunt apoi folosite pentru a genera un token nou, care este transmis înapoi aplicației client ca răspuns.

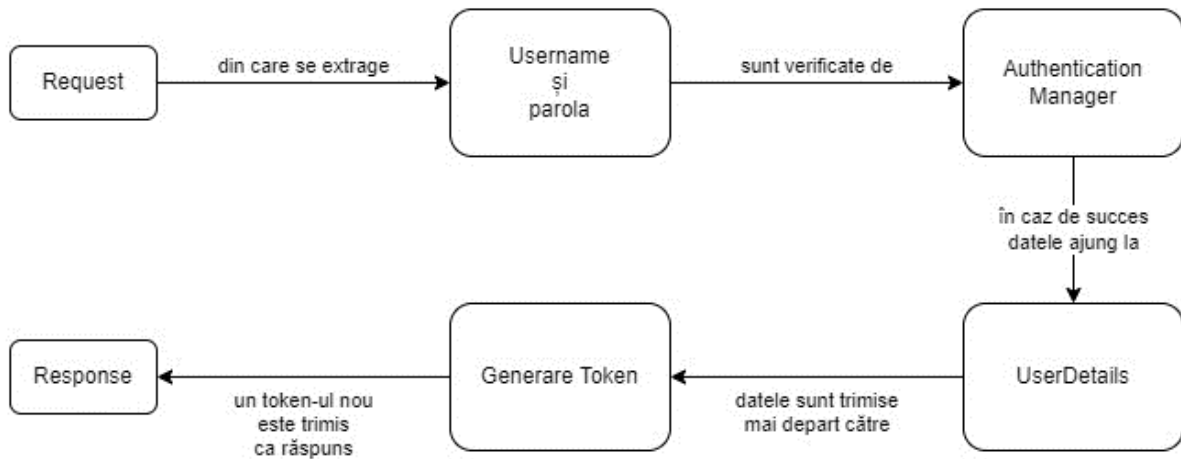


Fig. 25: Proces Autentificare

```

if(jwtUtil.validateToken(jwtToken, userDetails)) {

    UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken =
        new UsernamePasswordAuthenticationToken(userDetails, credentials: null, userDetails.getAuthorities());
    usernamePasswordAuthenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
    SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);

}
  
```

Fig. 26: UsernamePasswordAuthenticationToken și SecurityContextHolder

Cea de-al doilea rol de bază al Spring Security este cel de autorizare al request-urilor. Fiecare endpoint poate fi accesat numai de utilizatorii cu rolurile potrivite. Astfel, fiecare request este interceptat și filtrat. Din acesta este extrat header-ul care conține token-ul de autorizare. Token-ul este verificat și din el se obține username-ul. Acesta este verificat în baza de date, iar apoi se face validarea token-ului. Dacă datele din token corespund cu cele citite din baza de date, atunci datele sunt trimise spre autentificare folosind clasa „UsernamePasswordAuthenticationToken”. Mai departe, datele o dată mapate datele utilizatorului autentificat, acestea sunt stocate în SecurityContextHolder.

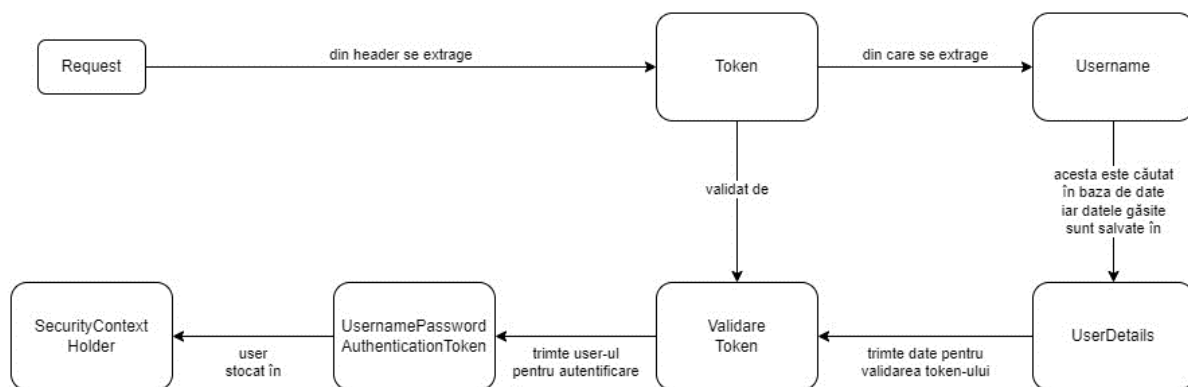


Fig. 27: Proces Autorizare

```

@Override
protected void configure(HttpSecurity httpSecurity) throws Exception {
    httpSecurity.cors();
    httpSecurity.csrf().disable() HttpSecurity
        .authorizeRequests().antMatchers("/login").permitAll() ExpressionInterceptUrlRegistry
        .antMatchers(HttpHeaders.ALLOW).permitAll()
        .anyRequest().authenticated()
        .and() HttpSecurity
        .exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint) ExceptionHandlingConfigurer<HttpSecurity>
        .and() HttpSecurity
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

    httpSecurity.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);
}

```

Fig. 28: WebSecurityConfigurer

Pentru a activa implementarea securității aplicației, Spring Security se folosește de interfața „WebSecurityConfigurerAdapter”. Aceasta reprezintă o clasă abstractă, ce conține deja metodele de configurare de bază și care poate fi apoi extinsă de clasa de configurare definită de dezvoltator, adnotată cu „@EnableWebSecurity”. Metoda implementată și rescrisă de clasă specifică, printre altele, ce request-uri au nevoie de autentificare și care nu, identifică o clasă care să gestioneze excepțiile aruncate la autentificări nereușite și cine se ocupă de filtrarea request-urilor. Totodată, permite utilizarea de request-uri CORS (Cross-Origin Resource Sharing), pentru servicii integrate, în cazul clienților web cu domenii diferite.

3.3.15. Arhitectura pe Microservicii

Microserviciile reprezintă o arhitectură de dezvoltare software ce presupune ca întregul proiect să fie împărțit în părți independente una de alta. Fiecare proiect de acest fel este gestionat separat de către echipa sa.

În aplicațiile clasice, de tip monolit, totalitatea proceselor executate de aplicație sunt strâns legate între ele. În cazul în care unul din procese este mai des utilizat decât celelalte, atunci trebuie alocate mai multe resurse pentru întreaga aplicație, inclusiv pentru procesele neutilizate atât de des.

De asemenea, în cazul în care anumite caracteristici ale aplicației nu mai corespund cerințelor de business ale proiectului și se dorește o reconfigurare a acesteia, întreaga arhitectură a aplicației va trebui scalată la noile cerințe. Adăugarea de noi funcționalități sau actualizarea celor deja existente poate fi o adevărată provocare în aplicațiile monolit.

Totodată, în cazul unor defecțiuni apărute la una din caracteristicile aplicației, deși poate un aspect minor al acesteia, întregul produs software se poate găsi în situația în care nu mai poate funcționa. Acest lucru duce și la un proces de testare al aplicației mult mai îndelungat și mai complex pentru fiecare modificare a codului.

În cazul arhitecturilor software bazate pe microservicii, aplicațiile sunt dezvoltate prin prisma unor componente independente care rulează fiecare în paralel anumite caracteristici ale aplicației. Aceste servicii comunică între ele prin intermediul unor interfețe API. Regula de bază este ca fiecare serviciu al aplicației să gestioneze o singură funcție de business a acesteia. Deoarece serviciile funcționează independent unul de altul, modificările asupra codului lor sursă sunt mai ușor de implementat, iar eventualele dificultăți întâmpinate în dezvoltarea aplicației sunt mai simplu de gestionat.

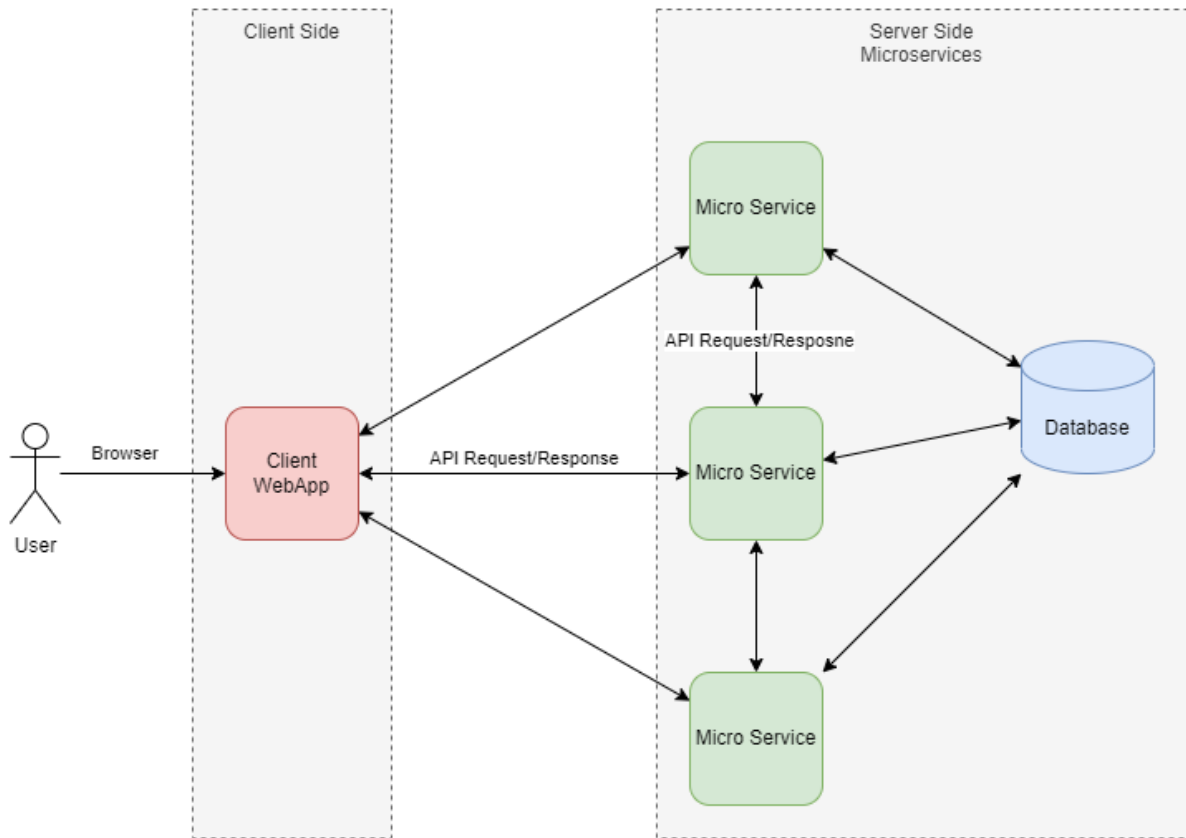


Fig. 29: Arhitectura pe Microservicii

3.4. Aplicația Client

Aplicația client a fost dezvoltată folosind Angular. Framework-ul fost descris într-unul din capitolele anterior. Înainte de a prezenta ce tehnologii au fost implementate în aplicație, voi face o scurtă prezentare a conceptului de DOM, pentru a înțelege mai bine cum funcționează Angular.

DOM (Document Object Model) este o interfață cu rolul de a reprezenta componentele paginii web sub formă de date. Acestea sunt reprezentate sub formă de obiecte sau noduri. Astfel, este creată posibilitatea ca limbajele de programare să interacționeze cu componentele web. Practic, DOM poate fi descris drept puntea de legătură dintre JavaScript și HTML. [8]

DOM nu este parte a unui limbaj de programare. Acesta este un API Web, folosit de către limbajele de programare pentru a putea interacționa cu limbajele markup ca HTML sau CSS. Astfel, DOM este independent de limbajul de programare și poate fi folosit împreună cu oricare dintre acestea.

Pentru accesarea DOM-ului nu este nevoie de nicio configurare din partea dezvoltatorului de aplicații web. Acesta este pur și simplu accesat prin intermediul limbajului de programare, în general JavaScript.

3.4.1. Componente

Componentele reprezintă principalul artefact al Angular. Acestea gestionează funcționalitatea de afișare a aplicației, și layer-ul cu care utilizatorul interacționează în mod direct. Componentele sunt reprezentate sub formă de folder-e, ce conține cel puțin trei fișiere: [9]

- HTML: gestionează elementele care sunt afișate în pagină de către componentă. Poate fi asemănat cu layer-ul de View din MVC;
- CSS: stilizează elementele din pagină. Totuși în cadrul sintaxei html se pot utiliza framework-uri precum Bootstrap care să preia această sarcină. Bootstrap folosește nume predefinite de clase pentru a stiliza componentele HTML;
- TypeScript: este fișierul de bază al componentei. Acesta definește structura acesteia, gestionează funcționalitățile ei, modul în care sunt afișate elementele, preia și afișează date utilizatorului și comunică cu alte artefacte.

```
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {
```

Fig. 30: Componenta Acasă

În imaginea de mai sus este prezentată declararea componentei „home”. Pentru a identifica artefactele, Angular folosește decoratori, asemănători adnotărilor din Spring. Aceștia conțin metadata despre componentă, fără de care aceasta ar fi o simplă clasă TypeScript. Decoratorul are rolul de a defini:

- Selectorul: este folosit de Angular pentru a încărca elementele componentei în DOM, atunci când tag-ul HTML indicat este găsit într-o componentă deja încărcată;
- Template-ul: indică fișierul HTML din care Angular preia elementele componentei și le încarcă în DOM;
- Stilul: indică fișierul CSS aferent componentei care stilizează elementele HTML ale componentei.

Ciclul de viață al unei componente începe atunci când aceasta este apelată prima dată de către Angular și unul dintre elementele ei este încărcat în DOM. Pe parcurs, componenta poate rămâne activă și prin componentele copil. Ciclul ia sfârșit atunci când ultimul ei element sau al unei componente copil este șters din DOM.

Înțelegerea acestui ciclu de viață este esențială pentru utilizarea interfețelor de tip hook. Acestea depind de momentele în care componenta este creată și ștearsă. În exemplul de mai sus, clasa „HomeComponent” implementează interfața „OnInit”, ce permite implementarea metodei „ngOnInit()”. Aceasta permite ca anumite metode ale clasei de exemplu să fie executate imediat ce componenta a fost creată, sau inițializarea anumitor variabile, cum ar fi cea de timp pentru a utiliza ora curentă.

3.4.1.1. Data biding

Principalul mod de comunicare între fișierul HTML al componentei și cel TypeScript îl reprezintă conceptul de Data Biding. Astfel, starea aplicației poate fi actualizată în timp real, în funcție de interacțiunea cu utilizatorul. [9]

Există trei tipuri de realizare a conceptului de Data Biding, în funcție de direcția datelor:

- Către template: atunci când datele ajung din fișierul TypeScript în cel HTML. Acesta este cel mai des întâlnit tip de biding și are cele mai multe scenarii de utilizare. De obicei, datele sunt trecute între paranteze pătrate, dar în cazul interpolării, datele sunt trecute între două rânduri de acolade;

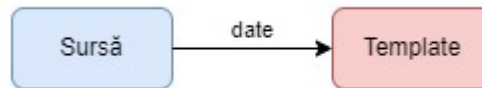


Fig. 31: Data binding către Template

- Către sursă: atunci când datele ajung din fișierul HTML în cel TypeScript. Cel mai des întâlnit în cazul Event Biding, de exemplu la apăsarea unui buton. Datele sunt trecute între paranteze rotunde;

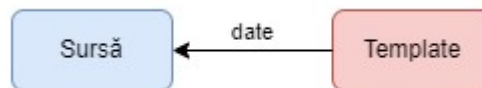


Fig. 32: Data binding către Sursă

- Two-way: atunci când datele pot fi modificate din ambele părți. Sunt folosite ambele tipuri de paranteze simultan pentru a marca datele.

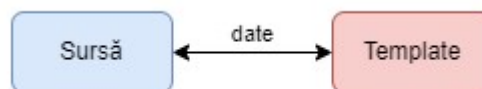


Fig 33: Data binding în ambele direcții

3.4.2. Servicii

Serviciile reprezintă un artefact al aplicațiilor client care are rolul de a gestiona request-urile către aplicația server, cât și alte funcționalități, pe care mai multe componente le împart. În general, serviciile gestionează metodele CRUD care pot fi executate asupra unui model. În Angular, interfețele reprezintă o modalitate de a structura datele prin maparea lor sub formă de obiecte.

Conceptul de injectare a dependențelor este reprezentat în Angular de către servicii, acestea crescând modularitatea framework-ului. Astfel, componentele gestionează doar interacțiunea cu utilizatorul, iar cerințele suplimentare, precum conectarea la server, cad în sarcina controller-elor. [9]

```
@Injectable({  
  providedIn: 'root'  
})  
export class EmployeeService {  
  
  no usages  
  constructor(private http: HttpClient, private userService: UserService) { }
```

Fig. 34: Serviciul Angajat

În exemplul de mai sus este definit serviciul Angajat, care gestionează operațiile CRUD ce pot fi operate mai departe de către aplicația server, asupra entității „Employee”. Se observă în constructorul clasei faptul că un serviciu poate depinde de mai multe servicii sau de alte clase.

Principalul avantaj al serviciilor îl reprezintă capacitatea lor de a fi injectate, iar astfel funcționalitățile lor sunt refolosite cu ușurință. Pentru a defini un serviciu, este utilizat decoratorul „@Injectable”. Pentru a injecta dependențele, Angular creează un injector implicit, iar serviciile create sunt tot în mod implicit înregistrate în injectorul „root”. Astfel, serviciile pot fi accesate ca dependențe de orice componentă a aplicației sau de chiar alte servicii. Pentru a injecta o dependență, aceasta trebuie trecută ca parametru în constructorul componentei, la fel ca în figura de mai sus.

3.4.3. Observable

RxJS (Reactive Extensions Library for JavaScript) este o librărie utilizată de Angular pentru a implementa observabilele și tot ce presupune comunicarea cu aplicația client. Acestea reprezintă conceptul de programare reactivă, prezentat în capitolul de backend, care stă la baza funcționării framework-urilor de front-end. Programarea reactivă are drept scop procesarea evenimentelor asincrone. Astfel, pot fi procesate mai multe evenimente intercalat.

Observabilele au rolul de a facilita comunicare dintre componente diferite ale aplicației, cum ar fi comunicarea cu alte servicii web. Acestea sunt un tip de date „lazy”: sunt declarate, dar nu și executate pe loc. Observabilele au nevoie de observatori, care primesc date de la primele prin metoda de subscripție. [10]

```
getById(id: number) : Observable<Employee> {
  let token : string | null = localStorage.getItem( key: 'token');
  let headers : HttpHeaders = new HttpHeaders( headers: {
    'Authorization': 'Bearer ' + token
  });
  let options : {headers: HttpHeaders} = {headers: headers};
  let url : string = this.baseUrl + '/' + id;
  return this.http.get<Employee>(url, options)
}

getById(id: number) : void {
  this.employeeService.getById(id).subscribe( observer: {
    next: (response: Employee) : void => {
      this.employee = response;
    },
    error: error => {
      this.toastr.error(error.error);
    }
  })
}
```

Fig. 35: Utilizarea Observabilelor

În exemplul de mai sus este prezentată citirea unui angajat cu un id anume din aplicația server. Prima metodă este din serviciul Angajat și are rolul de a crea request-ul de trimis către back-end, cu url-ul, header-ele și parametrii necesari. La răspunsul primit de la server, metoda întoarce înapoi o observabilă.

Cea de-a doua metodă face parte dintr-o componentă și are rolul a obține datele de la observabilă. Pentru aceasta, este creată o subscripție către observabilă. Valorile primite de la observabilă, sunt accesate prin intermediul unui obiect de tip observator. Acesta din urmă, reprezintă o serie de apeluri către observabilă și are trei metode:

- Next: în cazul în care răspunsul primit este cel dorit, iar request-ul a fost executat cu succes;
- Error: în cazul în care execuția request-ului a eșuat și a fost primit un mesaj de eroare;
- Complete: în cazul în care request-ul a fost executat cu succes, această metodă este apelată după execuția metodei next.

3.4.4. Module

Angular este un framework care se bazează pe modularitate și injectarea dependențelor, iar pentru implementarea acestor concepte, se bazează pe module, numite. Rolul modulelor este acela de a centraliza componentele, serviciile și alte artefacte cu funcționalități asemănătoare, într-un singur loc. La rândul lor, modulele pot face parte din alte module. Angular definește un astfel de modul în mod implicit la nivelul aplicației, numit „App Module”. Modulele reprezintă o clasă TypeScript, adnotată cu decoratorul „@NgModule”. [9]

```

@NgModule({
  declarations: [
    AppComponent,
    NavBarComponent,
    HomeComponent,
    LoginComponent,
    UserComponent,
    imports: [
      BrowserModule,
      AppRoutingModule,
      HttpClientModule,
      BrowserAnimationsModule,
      FormsModule,
      bootstrap: [AppComponent]

```

Fig. 36: App Module

În exemplul de mai sus sunt prezentate câteva fragmente din modulul rădăcină „App Module”. Majoritatea librăriilor importate de Angular reprezintă tot module Ng, cum ar fi „HttpClientModule”. Din acest punct de vedere, App Module poate fi comparat cu maven, care gestionează dependențele Spring.

Printre proprietățile prezentate în figura de mai sus, se numără:

- Declarații: unde sunt trecute serviciile și componentele membre;
- Importurile: reprezintă alte librării sub formă de module folosite de clase ale aplicației;
- Bootstrap: indică componenta de bază care este încărcată în DOM la pornirea aplicației.

3.4.5. Routare

Un tip special de module în Angular îl reprezintă cel de routare a aplicației, care are rolul de a descrie cum poate fi accesată fiecare componentă și de către cine. Framework-ul definește acest serviciu de routare în mod implicit, iar acesta permite navigarea între view-uri, în funcție de interacțiunea cu utilizatorul.

```

const routes: Routes = [
  {path: '', component: HomeComponent},
  {
    path: '',
    runGuardsAndResolvers: 'always',
    canActivate: [AdminAuthGuard],
    children: [
      // {path: 'admin-detail', component: AdminDetailComponent},
      {path: 'employees-list', component: EmployeesListComponent},
      {path: 'employee-create', component: EmployeeCreateComponent},
      {path: 'employee-details', children: [
        {path: ':id', component: EmployeeDetailsComponent},
      ]}
    ]
  }
]

```

Fig. 37: Modulul de routare

În exemplul de mai sus, sunt prezentate routele pentru fiecare componentă în parte. Se poate observa cum componenta Home este accesată în mod implicit și poate fi accesată de oricine. O adresă poate avea mai multe componente copil, în cazul în care aceste componente împart aceeași adresă la început. În schimb, următoarele componente din listă pot fi accesate numai de către administrator. Pentru aceasta, Angular necesită implementarea unor fișiere de tip Guard, care acționează ca niște filtre pentru fiecare componentă pentru care acesta este marcat.

```
canActivate(): Observable<boolean> {  
  return this.userService.currentUser$.pipe(  
    map( project: user : User | null => {  
      if (!user) return false;  
      if (user.role == 'admin') {  
        return true;  
      } else {  
        this.toastr.error( message: 'Access denied!');  
        return false;  
      }  
    })  
  )  
}
```

Fig. 38: Admin Guard

Guard-urile reprezintă de fapt niște simple clase TypeScript care pot fi injectate, asemenea serviciilor. În plus, acestea implementează „CanActivate”, cum metoda omonimă. Metoda folosește o variabilă de tip observabilă, în cazul de mai sus „currentUser\$”. Această variabilă reține ultimul utilizator autentificat în aplicație și este folosită pentru a determina dacă autentificarea a fost realizată sau nu.

4. Concluzii

Deși conceptele prezentate în aplicație nu sunt printre cele mai complexe întâlnite într-o aplicație web, acestea oferă suficientă robustețe proiectului pentru a rezolva problema căreia i se adresează. Respectarea conceptelor de bază ale framework-urilor și a bunelor practici de scriere a codului permit aplicației să poată fi cu ușurință dezvoltată în viitor.

Tehnologiile prezentate în lucrare se numără printre cele mai populare și folosite la ora actuală pe piață. Cu alte cuvinte, suportul pentru aplicației nu v-a fi oprit prea devreme. Astfel, consider că aplicația ar putea avea un ciclu de viață destul de lung. Printre principalele tehnologii care au fost folosite, se numără:

- MySQL;
- MongoDB;
- Docket;
- Spring Boot (și arhitectura pe microservicii);
- Angular.

Având în vedere cele prezentate în lucrare, consider că arhitectura bazată pe microservicii reprezintă viitorul aplicațiilor web. Avantajele oferite de acestea le fac net superioare aplicațiilor de tip monolit, în majoritatea scenariilor de utilizare.

Sper să pot continua dezvoltarea aplicației, iar o dată cu aceasta, să-mi îmbunătățesc cunoștințele de programare. Aș vedea în viitor, dezvoltarea unei aplicații mobile, complementare celei web.

Vă mulțumesc

Bibliografie

- [1] <https://www.mongodb.com/nosql-explained> (Ultima accesare 21.07.2023)
- [2] <https://www.redhat.com/en/topics/virtualization> (Ultima accesare 25.07.2023)
- [3] <https://docs.docker.com/reference/> (Ultima accesare 28.07.2023)
- [4] <https://docs.spring.io/spring-framework/docs/5.3.x/reference/html/index.html>
(Ultima accesare 15.08.2023)
- [5] <https://www.baeldung.com/spring-boot> (Ultima accesare 20.08.2023)
- [6] <https://www.baeldung.com/spring-5> (Ultima accesare 20.08.2023)
- [7] Craig Walls, *Spring in Action, Fourth Edition*
<https://www.manning.com/books/spring-in-action-fourth-edition> (Ultima accesare
20.08.2023)
- [8] <https://developer.mozilla.org/en-US/docs/Web> (Ultima accesare 22.08.2023)
- [9] <https://angular.io/docs> (Ultima accesare 24.08.2023)
- [10] <https://rxjs.dev/guide/overview> (Ultima accesare 24.08.2023)

Anexe

1. Utilizarea aplicației

În continuare vor fi prezentate câteva dintre scenariile de utilizare ale aplicației, pentru a putea vedea cum funcționează aplicația.

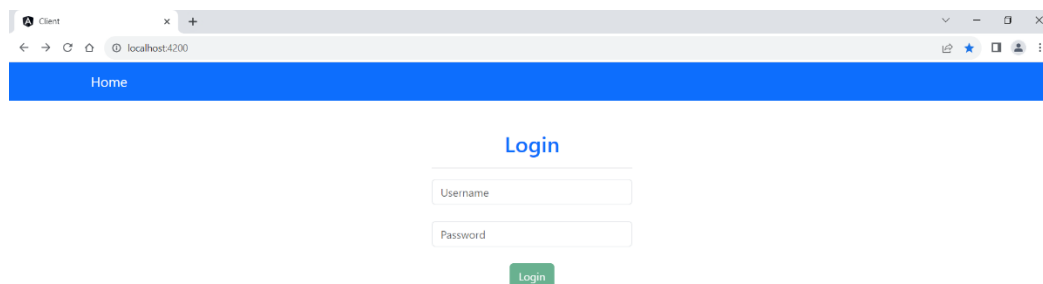


Fig. 1: Pagină autentificare

În imaginea de mai sus este afișată pagina de autentificare. Aceasta este disponibilă la momentul în care se accesează aplicația. Dacă nu este autentificat, utilizatorul nu poate folosi aplicația.

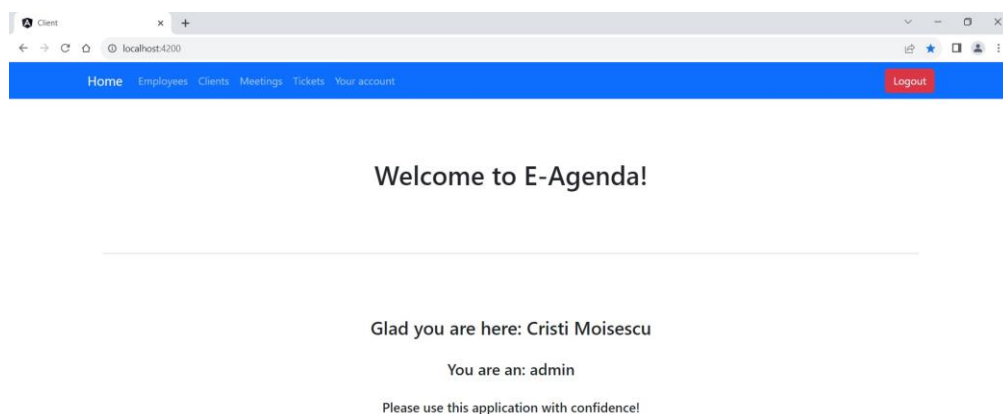


Fig. 2: Pagina Acasă pentru administrator

În figura de mai sus, este prezentată prima pagină a aplicației, atunci când este autentificat un administrator.

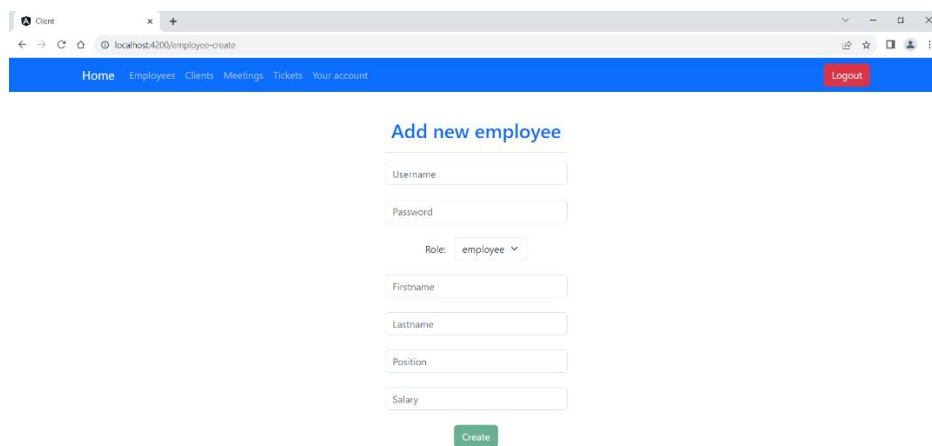
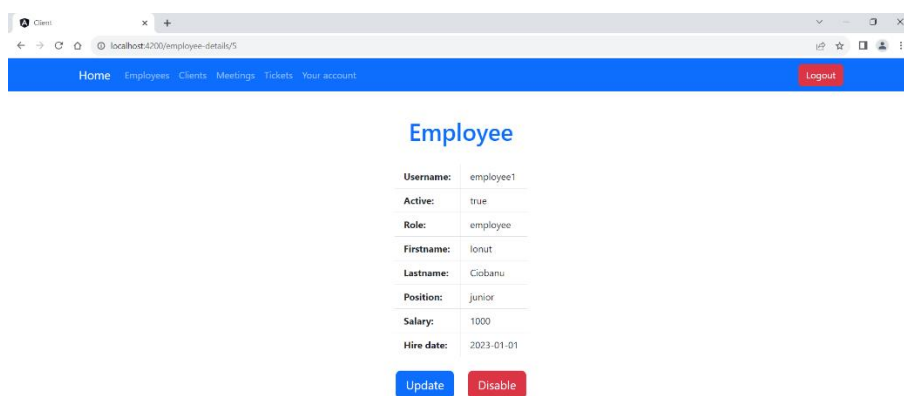
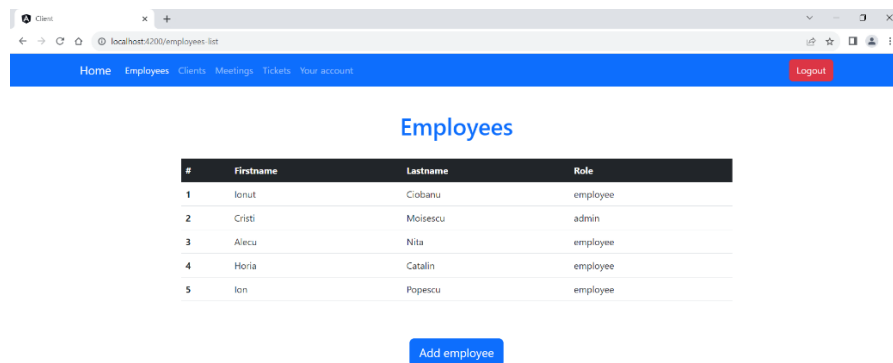
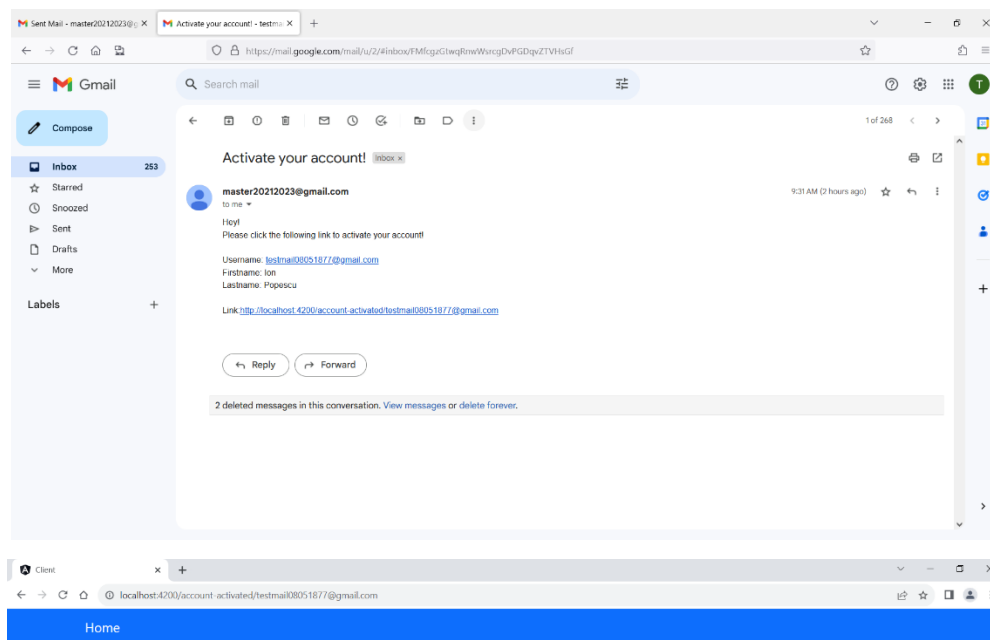


Fig. 3: Paginile lista angajaților, detalii despre angajați și formular adăugare angajați

Administratorul este cel mai important utilizator al aplicației și cel care gestionează întreaga activitate a acesteia. În primul rând, acesta este responsabil de adăugarea utilizatorilor în aplicație. În figura precedentă este prezentat formularul de adăugare a unui angajat nou. Pentru a își activa contul, noul utilizator trebuie mai întâi să confirme adresa de mail.



Account activated!

Fig. 4: Activarea contului

Utilizatorul își poate folosi apoi datele pentru a se autentifica în aplicație,

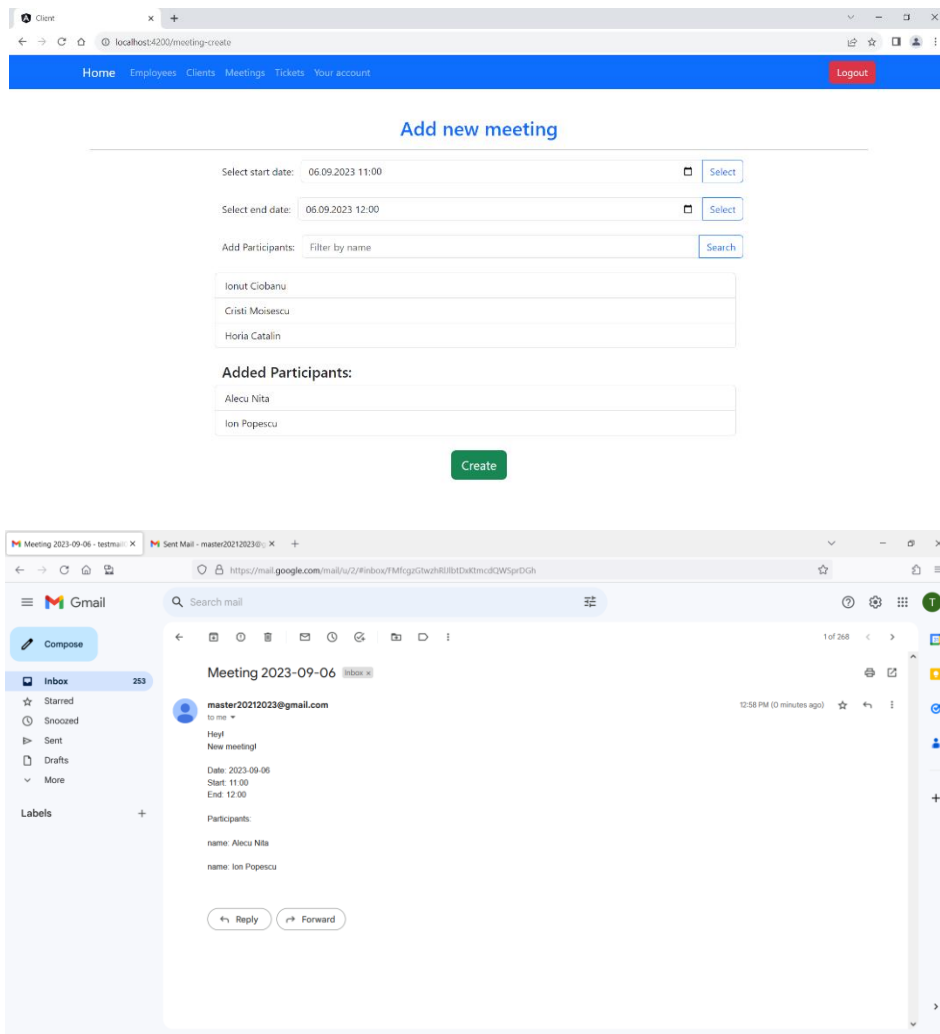


Fig. 5: Programare ședință

Totodată, administratorul este responsabil și de programarea ședințelor. La programarea unei ședințe, utilizatorul v-a primi pe adresa de mail o notificare privind ședința.

În cazul utilizatorilor cu rol de angajat, aceștia au posibilitatea de a vedea ședințele la care li ce cerea prezența și ce ticket-e sunt disponibile.

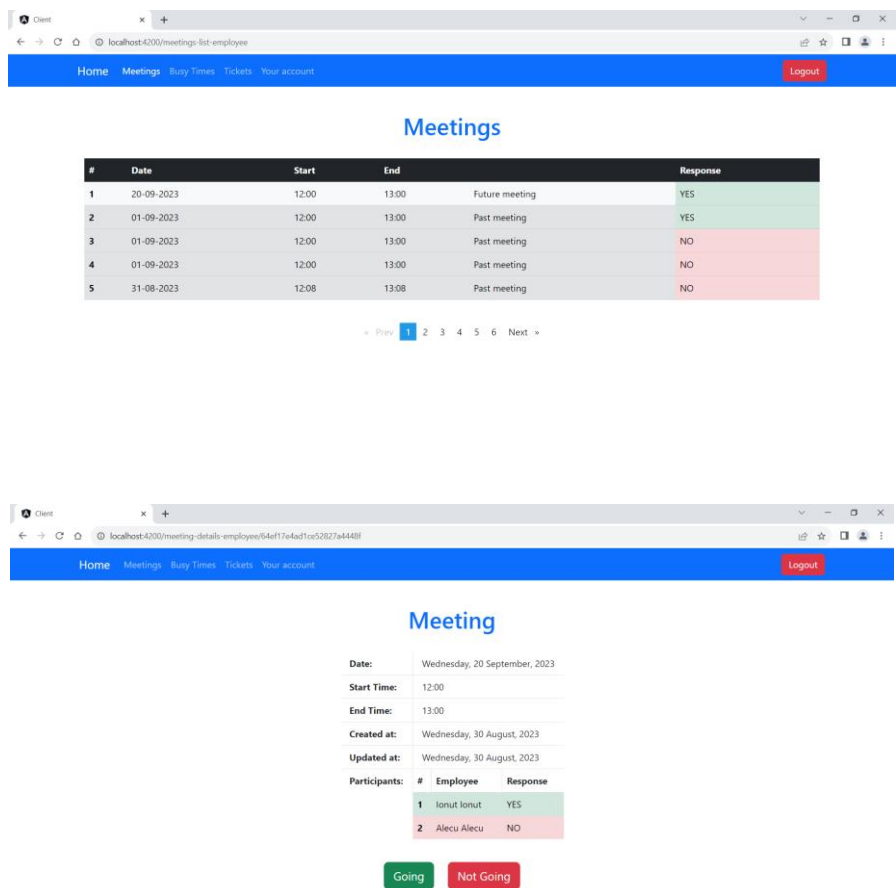
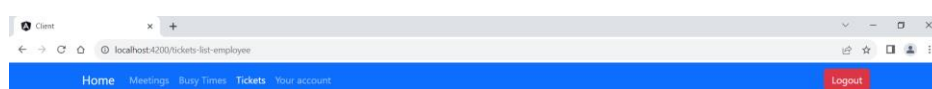


Fig. 6: Detalii ședință

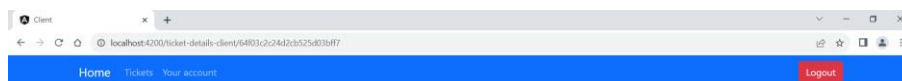
În exemplul de mai sus, utilizatorul trebuie să răspundă dacă participă sau nu la ședință.



#	Title	Created At	Company	Employee	Priority	Status
1	Calculator defect	27-07-2023 15:10	Hotel Parc	Ionut Ciobanu	Low	Working
2	Eroare aplicatie	27-07-2023 15:14	Hotel Parc	Ionut Ciobanu	Low	Complete



#	Title	Created At	Company	Employee	Priority	Status
1	Server defect	31-08-2023 10:07	Hotel Parc		High	Working
2	Router eroare	30-08-2023 18:09	Hotel Parc		Low	Working
3	Cablu rupt	31-08-2023 09:10	Hotel Parc		Low	Working
4	Router nefunctional	01-09-2023 15:20	Hotel Parc		Low	Disabled



Title:	Server defect
Priority:	High
Client:	Madalin Stanciu
Company:	Hotel Parc
Employee:	No user assigned!
Created At:	Thursday, 31 August, 2023
Updated At:	Friday, 01 September, 2023
Description:	Serverul nu mai porneste dupa o pana de curent. Toata activitatea este blocata
<div>Update</div> <div>Disable</div>	

Fig. 7: Gestionarea ticket-elor

Angajații își pot vedea ticket-ele pe care le au asignate, cât și pe cele încă ne atribuite nimănui. Aceștia pot accesa mai multe detalii despre ticket și decid dacă se ocupă de el sau nu.

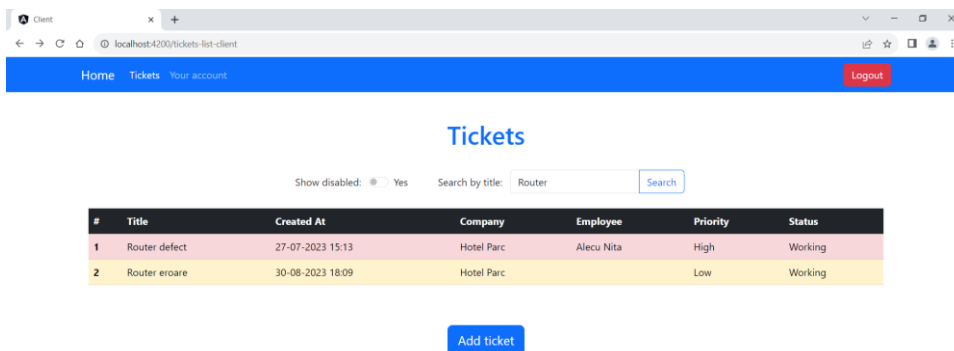
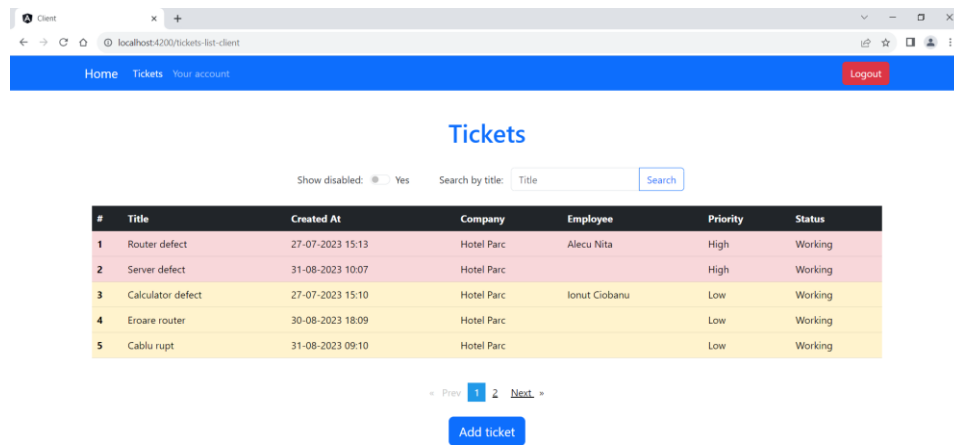


Fig. 8: Ticket-e

Clientul are posibilitatea de a vedea stadiul ticket-elor companiei pentru care lucrează.