MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

Centrum – Agregator de emailuri

Absolvent: **Bogdan SOCACIU-CUMPĂNAȘU**

Coordonator ştiinţific: **As. Ing. Cosmina-Daniela IVAN**

2018

MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

DECAN,
Prof. dr. ing. Liviu MICLEA

DIRECTOR DEPARTAMENT,
Prof. dr. ing. Rodica POTOLEA

Absolvent: **Bogdan SOCACIU-CUMPĂNAȘU**

Centrum – Agregator de emailuri

1. **Enunţul temei:** *Obiectivul acestui proiect este construirea unui sistem software responsabil cu agregarea emailurilor din diferite surse și afișarea acestora prin intermediul unei interfețe web. Punctul forte al sistemului va fi ușurința de instalare și intreținere a acestuia, ușurință atinsă prin intermediul utilizării unei arhitecturi bazate pe miscroservicii și a unor sisteme de management a infrastructurii precum Rancher.*

2. **Conţinutul lucrării:** *Introducere, Obiectivele Proiectului, Studiu Bibliografic, Analiză și Fundamentare Teoretică, Proiectare de Detaliu și Implementare, Testare și Validare, Manual de Instalare și Utilizare, Concluzii, Bibliografie*

3. **Locul documentării**: Universitatea Tehnică din Cluj-Napoca, Departamentul Calculatoare

4. **Consultanţi**: As. Ing. Cosmina-Daniela Ivan

5. **Data emiterii temei:** 1 februarie 2018

6. **Data predării:** 9 iulie 2018

Absolvent: _____

Coordonator ştiinţific: _____

MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

Declaraţie pe proprie răspundere privind autenticitatea lucrării de licenţă

Subsemnatul(a)_____
_____,
legitimat(ă) cu _____ seria _____ nr. _____
CNP _____, autorul lucrării
_____
_____
_____elaborată în vederea susţinerii examenului de finalizare a studiilor de licenţă la Facultatea de Automatică şi Calculatoare, Specializarea _____ din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea _____ a anului universitar _____, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării, şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

In cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

Data

_____

Nume, Prenume

_____

Semnătura

Cuprins

Capitolul 1. Introducere

Scopul acestui proiect este proiectarea și implementarea unui sistem software responsabil cu agregarea emailurilor. Odată cu adoptarea tot mai largă a sistemului de poștă electronică, au fost deschise miliarde de conturi de email, existând un număr foarte mare de utilizatori având multiple conturi de email personale. Sistemul propus va oferi utilizatorilor posibilitatea centralizării emailurilor din diferite surse, asigurând confidențialitatea și siguranța datelor.

1.1. Contextul proiectului

Una dintre principalele inconveniențe a sistemelor aflate momentan pe piață, precum Microsoft Exchange Server™, este nivelul de cunoștințe necesar pentru configurarea și instalarea sistemului respectiv. Această inconvenientă se reflectă în general prin configurări complexe, configurarea prin convenții sau nevoia instalării și configurării unei aplicații client care să permită interacțiunea utilizatorului cu server-ul.
De asemenea, poate apărea și problema scalabilității unei astfel de soluții. În general, soluțiile tradiționale de agregare sunt aplicații monolitice, instalate și rulate pe un server foarte puternic. Scalarea acestor tipuri de aplicații este un proces extrem de complex și ineficient, fiind posibile atingerea unor limitări hardware, precum ocuparea unui spațiu prea mare pe disc, în cazul aplicației client, sau atingerea unor limite de memorie, putere de procesare și lățime de bandă, în cazul aplicației server.

Totodată, întreținerea unui astfel de sistem poate să fie o sarcină foarte complexă. Dificultatea de actualizare și de descoperire a surselor unor eventuale probleme reprezintă un factor descurajator pentru unele departamente IT. În domeniul multor intreprinderi se dorește utilizarea unui sistem robust și flexibil în același timp, sistem care să facă față unei încărcări ridicate, dar să fie ușor de scalat, configurat și actualizat.

Un alt factor problematic este reprezentat de monitorizarea și întreținerea sistemului. Actualizările trebuie să fie efectuate într-un mod sincronizat, dând naștere unor perioade de timp în care sistemul devine inactiv, afectând utilizabilitatea acestuia. Totuși, există și soluții cloud precum Google Apps for Business, a căror natură oferă un nivel maxim de disponibilitate a sistemului, însă acestea nu oferă nivelul de confidențialitate a datelor necesar unor companii. Confidențialitatea datelor a devenit un subiect important de discuție în domeniul dezvoltării și utilizării aplicațiilor software, întrucât aceasta este amenințată de vastele breșe de securitate descoperite tot mai des la nivelul întreprinderilor mari precum Google, Yahoo și Microsoft. Astfel, tot mai multe companii aleg să își implementeze un canal de comunicare privat, majoritatea utilizând aceste canale private de comunicare complementar cu sistemele furnizate de companii precum cele amintite anterior.

De asemenea, unele servicii precum Microsoft Exchange Server sunt prea complexe pentru utilizatorul obișnuit, necesitând învățarea modului de utilizare a unei aplicații client nu tocmai simplă. Astfel, o soluție online poate oferi un nivel de utilizabilitate ridicat, asemănător altor sisteme administrate într-un mod centralizat, precum Google Apps for Business.

Totuși, există pe piață sisteme complexe precum Kerio Connect, sistem care poate să fie instalat local, asigurând confidențialitatea datelor, și este ușor de administrat. Însă,

acest sistem este un produs comercial cu costuri deloc neglijabile, costând 495$/an doar pentru 5 utilizatori, fiind necesară o plată suplimentară de 27$/an per utilizator suplimentar. În cazul multor întreprinderi mici și mijlocii, acesta este un cost ce poate fi greu de justificat de avantajele oferite de un sistem de acest tip.

Centrum dorește să ofere o alternativă capabilă să elimine aceste neajunsuri prin decizii arhitecturale precum:

- Instalarea aplicației într-un domeniu local, oferind un nivel mare de disponibilitate și asigurând confidențialitatea datelor.
- Utilizarea sistemului prin intermediul unui navigator web, eliminând necesitatea instalării unei aplicații client.
- Dezvoltarea aplicației utilizând o arhitectură bazată pe microservicii, garantând posibilitatea scalării ușoare a aplicației.
- Realizarea actualizărilor aplicației fără a afecta disponibilitatea acesteia, prin utilizarea unui proces bazat pe replicarea containerelor în care este instalat sistemul.

Totodată, Centrum este un sistem open-source, oferind caracteristicile evidențiate anterior fără costuri adiționale găzduirii sistemului. Astfel, sistemul propus are capacitatea de a deveni o unealtă utilizată de un segment de piață neacoperit de sistemele amintite anterior, fiind posibilă chiar concurența cu unele dintre aceste sisteme pe segmentele de piață acoperite de acestea. Acest ultim punct este posibil deoarece unele companii ar putea alege o versiune gratuită cu mai puține caracteristici, în cazul în care caracteristicile suplimentare oferite de soluțiile descrise mai sus nu sunt necesare pentru buna funcționare a comunicării necesare businessului.

1.2. Structura lucrării

În acest subcapitol se va prezenta structura lucrării, enumerând capitolele și descriind conținutul acestora.

- **Capitolul 1 – Introducere**: Acest capitol conține o prezentare scurtă temei proiectului, contextul problemei și o descriere sumară a soluției oferite de sistemul prezentat în această lucrare.
- **Capitolul 2 – Obiectivele Proiectului**: Capitolul acestă prezintă tema proiectului, evidențiind obiectivele propuse spre îndeplinire de către sistemul proiectat.
- **Capitolul 3 – Studiu Bibliografic**: În acest capitol este prezentat studiul necesar dezvoltării sistemului descris. Studiul reprezintă atât documentația tehnică necesară implementării și integrării aplicațiilor, cât și descrierea comparativă a unor soluții similare cu aplicația dezvoltată, subliniind diferențele dintre acestea. Astfel, este descrisă și poziționarea pe piață a sistemului.
- **Capitolul 4 – Analiză și Fundamentare Teoretică**: Conținutul acestui capitol reflectă, într-un mod structurat, tehnologiile utilizate pentru implementarea sistemului. Tot în acest articol sunt prezentate cerintele funcționale și non-funcționale, cazurile de utilizare și resursele necesare sistemului.
- **Capitolul 5 – Proiectare de Detaliu și Implementare**: Acest segment din lucrare este centrat în jurul arhitecturii și modelului de procesare al

aplicației. Vor fi descrise într-un mod detaliat componentele aplicației și impactul acestora în buna funcționare a sistemului propus.

- **Capitolul 6 – Testarea, Validarea și Evaluarea**: În această secțiune a lucrării se evidențiază metodele de testare ale sistemului, precum și rezultatul evaluării funcționale și non-funcționale a sistemului.
- **Capitolul 7 – Manual de Instalare și Utilizare**: Continutul acestui capitol înglobează manualul de instalare a aplicațiilor sistemului, dar și manualele de utilizare și administrare a aplicațiilor menționate.
- **Capitolul 8 – Concluzii**: În acest capitol sunt prezentate obiectivele realizate de sistemul descris, precum și eventuale dezvoltări ulterioare ale acestuia.

Capitolul 2. Obiectivele Proiectului

În acest capitol vor fi prezentate obiectivele proiectului, acoperind topicuri precum: specificarea problemei, poziționarea produsului și descrierea utilizatorilor și a părților interesate. Sistemul are scopul principal al agregării adreselor de corespondență electronica, scopurile secundare fiind reprezentate de diverse criterii nonfuncționale, precum simplitatea instalării, ușurința în utilizare și confidențialitatea datelor utilizatorilor.

2.1. Obiective principale

Scopul principal al lucrării este implementarea unui sistem complex cu ajutorul tehnicilor moderne din domeniul DevOps, precum containerizarea aplicațiilor și automatizarea instalării și a integrării acesteia. Totodată, ca urmare al acestui obiectiv, se dorește dezvoltarea unei soluții competente de agregare a conturilor de poștă electronică, o soluție care are potențialul de a ocupa un segment din piața actuală a aplicațiilor de administrare a poștei electronice.

De altfel, următoarele tabele descriu succint acest obiectiv, evidențiind problema rezolvată, poziționarea produsului și responsabilitățile părților interesate.

Problema	emailurilor decentralizate
care afectează	utilizatori de multiple conturi de email, atât locale, cât și online
impactul cărora este	dificultatea managementului emailurilor
poate fi soluționată prin	o platformă online flexibilă, eficientă din punct de vedere al costurilor și ușor de implementat și menținut, având rolul de centralizare a emailurilor din diferite surse.

Tabel 2.1 - Specificația problemei

Pentru	utilizatorii de multiple conturi de email
Care	simt nevoia de a își accesa emailurile din orice locație, într-un mod centralizat
Centrum	este un produs software
Care	oferă posibilitatea centralizării emailurilor din diferite surse
Spre deosebire de	servicii precum Microsoft Exchange Server și Google Apps for Business
Acest produs	este gratuit, sigur, ușor de instalat și de întreținut

Tabel 2.2 - Poziționarea produsului

Nume	Descriere	Responsabilități	Parte interesată
Șeful operativ	Managerul operativ al companiei	Cere instalarea unei instanțe Centrum în domeniul local al unei companii cu scopul de a crește productivitatea angajaților și siguranța datelor	Propriu
Departamentul IT	Departamentul responsabil cu managamentul sistemelor de comunicare din interiorul unei companii.	Instalează Centrum	Șeful operativ
Angajatul companiei	Utilizator primar al sistemului	Utilizează Centrum cu scopul de a își centraliza adresele de corespondență digitală, atât cele proprii, cât și adresele companiei	Șeful operativ

Tabel 2.3 - Descrierea utilizatorilor și a părților interesate

2.2. Obiective secundare

Primul obiectiv secundar pe care sistemul propus dorește să îl atingă este reprezentat de utilizabilitatea acestuia. Acest obiectiv este măsurat prin gradul de ușurință prin care un utilizator interacționează cu sistemul și este reflectat prin dificultatea procedurii de instalare și configurare a unui sistem de către un utilizator, cât și prin gradul de expresivitate și consistență a interfeței utilizator.

De asemenea, aplicația trebuie să ofere utilizatorilor posibilitatea de a își crea un cont și de a se autentifica folosind contul creat. În urma creării contului, acesta va trebui activat. Activarea contului este efectuată prin intermediul accesării unei adrese generată unic pentru fiecare utilizator. Această adresă este trimisă prin email, iar utilizatorul are 24 de ore pentru a folosi adresa generată.

Un alt obiectiv este reprezentat de posibilitatea dezvoltării continue a aplicației. Întrucât domeniul IT se dezvoltă într-un ritm rapid, aplicațiile dezvoltate trebuie să suporte acest ritm de dezvoltare, fiind necesar un grad mare de adaptabilitate a acestora. Acest ritm rapid de dezvoltare a industriei este complementat și de posibilitatea creșterii foarte rapide a numărului de utilizatori. Astfel, soluția dezvoltată trebuie să fie ușor de extins și de scalat.

Capitolul 3. Studiu Bibliografic

3.1. Microservicii

3.1.1. Introducere

Modelul arhitectural monolitic este utilizat într-un mod uzual în dezvoltarea aplicațiilor, acesta fiind eficient în cazul aplicațiilor mai puțin complexe, unde testarea, dezvoltarea si implementarea sunt sarcini relativ ușoare. Insă, în cazul aplicațiilor complexe, arhitectura monolitică devine un obstacol în calea dezvoltării și integrării. Livrarea continuă a aplicației devine dificilă, găsindu-ne deseori blocați în alegerile tehnologice initiale. Astfel, în cazul unor aplicații complexe, devine convenabilă folosirea arhitecturii bazate pe microservicii. [1]

Microserviciile au rolul de a împărți aplicația în seturi distincte de componente interconectate, fiecare componentă implementând un set propriu de caracteristici si funcționalități. Comunicarea dintre entități (module) este realizată prin intermediul unor mecanisme simple de comunicare, precum HTTP REST, și este construită în jurul unor domenii logice bine definite.

În zilele noastre, dezvoltatorii de aplicații sunt captivați de microservicii, însă mulți sunt, în continuare, sceptici cu privire la originalitatea acestui model arhitectural. Totuși, dezvoltatori precum Netflix, Amazon și eBay, pot confirma faptul că o implementare corectă a acestui model arhitectural aduce valoarea promisa, în cazul unei aplicații sau al unui sistem de mare anvergură.

3.1.2. Avantaje și dezavantaje

Arhitectura bazată pe microservicii prezintă următoarele avantaje principale:

- **Individualitatea serviciilor**. Permite o întelegere mai ușoară a aplicației, cât și dezvoltarea independentă a serviciilor, realizând o decuplare a acestora.
- **Ușurința introducerii unor noi limbaje si librării**, fiind posibilă testarea și integrarea acestora la nivelul unui singur serviciu.
- **Transferarea responsabilităților la nivel de serviciu**, precum polițe de securitate, scalare, configurarea hardware, etc.
- **Redundanța sistemului**, căderea unui serviciu necauzând căderea întregului sistem.

Totodată, microserviciile prezintă și un set de dezavantaje, utilizarea acestora fiind dăunătoare în unele situații. Principalul dezavantaj îl reprezintă creșterea complexității aplicațiilor prin creșterea dinamismului din sistemului. Astfel, apare nevoia unei automatizări la nivel înalt pentru a eficientiza dezvoltarea și integrarea sistemului. De asemenea, managementul comunicării între microservicii devine o sarcină complexă, fiind posibilă apariția unor probleme în comunicarea distribuită dintre acestea.

În ciuda dezavantajelor de mai sus, o arhitectura bazată pe microservicii poate aduce avantaje tangibile în cazul aplicațiilor complexe și cu un ritm rapid de dezvoltare, în special în cazul aplicațiilor de tip SaaS (Software-as-a-Service), precum Centrum.

3.1.3. Scalabilitate

Înainte de a intra în detalii în ceea ce priveste provocările aduse de microservicii, trebuie amintit nivelul superior de scalibilitate adus de o arhitectură bazată pe microservii, în comparație cu o arhitectură monolitică. Scalabilitatea poate fi reprezentată prin intermediul „cubului scalării" descris în „The Art of Scalability", observabil în Figura 3.1 [2].

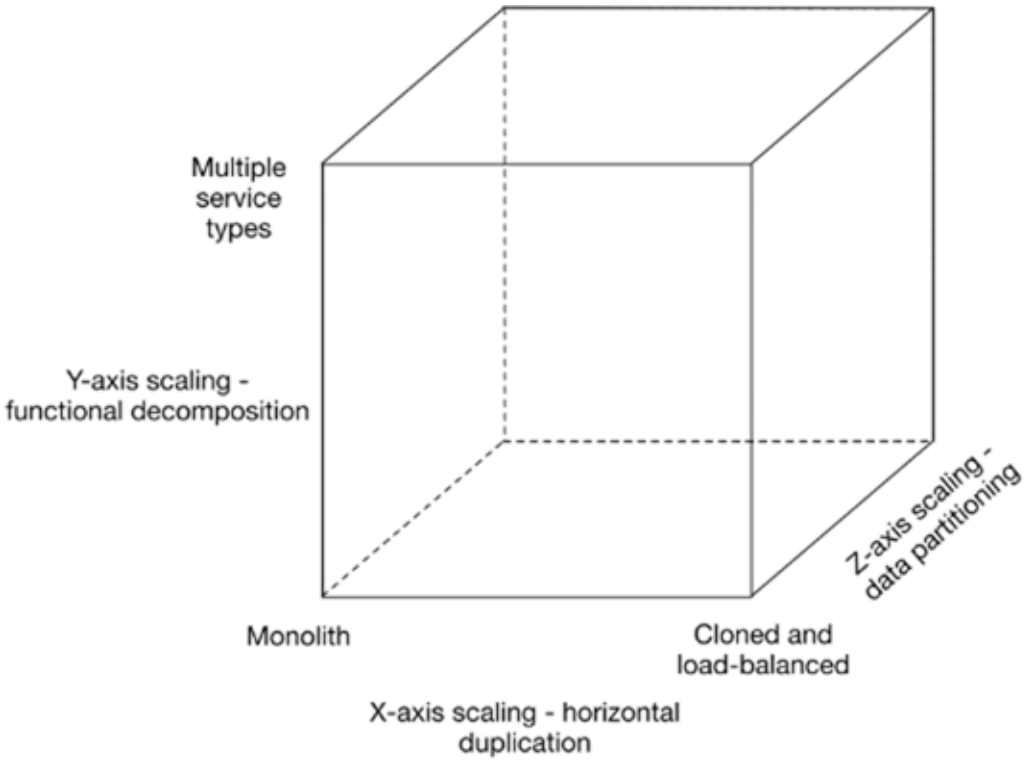

Figura 3.1 - "Cubul scalării"

După cum se poate observa, aplicațiile monolitice se pot scala prin intermediul duplicării instantelor aplicației sau prin partiționarea datelor (astfel încât diferite instanțe ale aplicație să interactioneze cu diferite subseturi de date). Prin împărțirea aplicației în servicii independente din punct de vedere funcțional, se adaugă o dimensiune extra în ceea ce privește nivelul de scalabilitate. Astfel, prin combinarea celor trei dimensiuni, se realizează o scalare independentă a fiecărui serviciu în funcție de nivelul de încărcare al acestuia, creând un avantaj considerabil în cazul aplicațiilor în care încărcarea nu este distribuită în mod egal la nivelul serviciilor.

3.1.4. Provocări și soluții

Dezvoltarea aplicațiilor folosind o abordare bazată pe microservicii cere un mod diferit de gândire a operațiilor de build, lansare și management. Nu se pot construi aplicații folosind premisa că știm dinainte totul legat de robustețea sistemului. In sisteme complexe, precum cele care folosesc o arhitectură bazată pe microservicii, astfel trebuie dezvoltată o abilitate de dezvoltare într-un mediu imprevizibil. În continuare, vom detalia câteva din

provocările implementării unui sistem bazat pe microservicii, oferind totodată soluții pentru a depăși aceste provocări.

3.1.4.1. Proiectarea robustă

În cadrul sistemelor complexe, defecțiunea unor componente poate apărea. Dispozitivele de stocare se pot defecta, cablurile de rețea pot fi deconectate sau deteriorate, se pot face actualizări și operații de întreținere a bazelor de date care pot avea un efect negativ asupra acestora, iar mașinile virtuale pot se pot corupe sau pot dispărea în totalitate. Un singur defect poate fi propagat în întregul sistem, ducând la căderea acestuia.

Tradițional, se încearcă prezicerea componentelor din aplicație ce s-ar putea defecta și construirea unor nivele suplimentare de protecție în jurul acestora. Acest mod de gândire devine problematic în cazul unor sisteme de dimensiuni mari, întrucât nu se poate prezice întotdeauna stabilitatea acestora. Defecte pot apărea mereu, astfel fiind necesară dezvoltarea unei aplicații rezistente, care să fie capabilă să suporte eventualele defecte, nu doar să le prevină.

Construirea sistemelor distribuite este diferită de construirea aplicațiilor monolitice prin introducerea noțiunii de comunicare prin intermediul unei rețele de calculatoare, comparativ cu apelurile locale ale aplicațiilor tradiționale ce folosesc un spațiu de memorie partajat. Rețelele sunt, la baza acestora, nefiabile. Comunicarea prin rețele poate eșua din diverse motive precum: puterea slabă a semnalului, conectori defectuoși sau de proastă calitate, firewall-uri. Aceste posibile probleme pot duce la o creștere în timpii de răspuns ai aplicației, însă pot provoca și o cădere completă a unor sisteme dependente de timpii de răspunși alterați.

3.1.4.2. Proiectarea conștientă de dependințe

Pentru a putea evolua rapid și pentru a fi agili din punct de vedere organizațional folosind sisteme distribuite, trebuie să proiectăm sisteme fiind conștienți de dependințele acestora. Astfel, vom slăbi cuplajul din interiorul echipelor noastre, al tehnologiilor folosite și al administrării. Unul dintre obiectivele finale ale utilizării microserviciilor este profitul din autonomia echipelor și dezvoltarea serviciilor autonome. Prin implementarea acestora, se construiește un mediu capabil de a face față schimbărilor rapide fără a avea un impact asupra serviciilor adiacente sau asupra întregului sistem. Acest fapt se transpune prin dependința de servicii, însă tratarea corectă a situațiilor în care acestea sunt indisponibile sau deteriorate.

3.2. Stocarea distribuită a datelor

Bazele de date tradiționale oferă mecanisme avansate de stocare și interogare a datelor prin intermediul consistenței puternice și a tranzacțiilor. Aceste medii de stocare au atins un nivel nemărginit de fiabilitate, stabilitate și suport, însă, în ultimul deceniu, cantitatea de date stocate și analizate al unor aplicații a ajuns la dimensiunea și nivelul de complexitate marginal soluțiilor tradiționale. Date precum emailurile unui număr signifiant de utilizatori sau metricile generate de o rețea de senzori sunt exemple a unui fenomen apărut recent, numit **Big Data**. Clasele de sisteme de stocare capabile de a suporta integrarea cu un număr mare de date sunt descrise colocvial sub termenul de **baze de date NoSQL**, majoritatea oferind scalabilitate orizontală și o disponibilitate mai ridicată decât cea a soluțiilor relaționale, sacrificând însă unele mecanisme de interogare și garanția

consistenței. Aceste compromisuri sunt cruciale pentru programarea orientată pe servicii și pentru modelele de aplicații „as-a-service”, deoarece un serviciu de sine stătător nu poate fi mai scalabil și mai tolerant la erori decât sistemul de stocare de care depinde acesta. Aceste idei sunt definite în lucrarea [3]

3.2.1. Modele de date

Cea mai uzuală diferență dintre diferitele soluții NoSQL este modul de stocare si acces a datelor, fiecare sistem analizat în continuare fiind catalogat în funcție de acest criteriu.

3.2.1.1. Stocare de tip cheie-valoare

O stocare de tip cheie-valoare consta in identificarea datelor stocate prin intermediul unei chei unice. Astfel, din cauza acestui mod simplu de stocare, sunt suportate doar operațiile CRUD (Create, Read, Update, Delete). Bazele de date ce folosesc acest mod de stocare sunt catalogate ca fiind **schemaless**, orice metadate legate de structura datelor fiind codificate în logica aplicației (schema-on-read) fără a dispune de o definire prin intermediul unui limbaj de definire a datelor (schema-on-write).
Avantajele acestui model de stocare constă în simplicitatea acestuia. Aceste abstractizări simple ușurează procesele de partiționare și interogare a datelor, baza de date fiind capabilă de a o latență scăzută și o capacitate de transfer ridicată.

3.2.1.2. Stocare de tip document

O stocare de tip document este o stocare de tip cheie-valoare care restricționează valorile la tipuri de formate semi-structurate, precum document JSON. Acest mod de stocare oferă o flexibilitate crescută în accesarea datelor, fiind posibilă atât citirea unui întreg document, cât și a unor parți individuale din acesta. Astfel, se pot realiza operații complexe pe datele stocate, operații precum agregarea și căutarea textuală în interiorul colecțiilor de documente.

3.2.1.3. Stocare pe coloane

Stocarea pe coloane își atribuie numele modului de a explica structura datelor din cadrul acestui tip de stocare, o bază de date relațională unde unitatea principală de reprezentare a unei intrări în baza de date este coloana. Acest mod de organizare a datelor este asemănător unor mapări sortate și distribuite pe nivele multiple: la nivelul întâi sunt stocate chei cu ajutorul cărora se identifică rânduri din tabele, rânduri ce conțin structuri de tip cheie-valoare. Cheile aparținând primului nivel se numesc chei de rând, iar cele din al doilea nivel se numesc chei de coloană. Astfel, acest tip de stocare face posibilă organizarea datelor în tabele cu un număr flexibil de coloane, deoarece nu va exista o cheie de coloană fără o valoare corespunzătoare, fiind posibilă stocarea valorilor nule fără ocuparea unui spațiu adițional.

3.2.2. Cerințe funcționale

În continuare, vor fi evidențiate o serie de cerințe funcționale uzuale, fiind comparat suportul unor sisteme de stocare NoSQL reprezentative tipurilor de stocare de mai sus cu cel oferit de un sistem relațional de management al datelor. [4]

	MongoDB	Redis	Hbase	Riak	Cassandra	MySQL
Scanarea Interogărilor	✓	✓	✓	✗	✓	✓
Tranzacții ACID	✗	✓	✗	✗	✗	✓
Scrieri Condiționale	✓	✓	✓	✗	✓	✓
Join-uri	✗	✗	✗	✗	✗	✓
Sortări	✓	✗	✓	✗	✓	✓
Filtrarea Interogărilor	✓	✗	✗	✗	✗	✓
Căutare Textuală	✓	✗	✗	✓	✗	✗
Analiză	✓	✗	✓	✓	✓	✓

Tabel 3.1 – Compararea sistemelor de stocare pentru o serie de cerințe funcționale uzuale

După cum se poate observa, utilizarea unei soluții precum MongoDB nu este posibilă fără anumite compromisuri, fiind nesuportate operațiile de Join specifice bazelor de date relaționale, precum și tranzacțiile ACID. Din fericire, aceste cerințe funcționale nu sunt importante pentru aplicația dezvoltată.

De asemenea, se poate face o comparație directă între MongoDB și Redis, cel din urmă oferind suport pentru tranzacții ACID însă având neajunsuri în cadrul unor cerințe esențiale pentru sistemul propus. Printre aceste neajunsuri se numără lipsa posibilității de sortare a datelor, absența suportului pentru căutare textuală, cât și imposibilitatea filtrării interogărilor.

Totuși, o bază de date relațională precum MySQL suportă majoritatea cerințelor din tabel, însă aceasta nu are gradul de flexibilitate oferit de o soluție NoSQL. În plus, MySQL nu oferă suport pentru căutare textuală, fiind necesară introducerea în sistem al unei utilitare precum Elasticsearch pentru atingerea acelui obiectiv.

Prin urmare, în ceea ce privește cerințele funcționale ale unui sistem de management al datelor, MongoDB este soluția potrivită aplicației descrise, acesta suportând operații esențiale bunei funcționări al aplicației, făcând compromisuri în zona criteriilor neimportante sistemului.

3.2.3. Cerințe non-funcționale

Totodată, vom compara suportul acelorași sisteme pentru o serie de cerințe non-funcționale uzuale.

	MongoDB	Redis	Hbase	Riak	Cassandra	MySQL
Scalabilitatea Datelor	✓	✗	✓	✓	✓	✗
Scalabilitatea Scrierilor	✓	✗	✓	✓	✓	✗
Scalabilitatea Citirilor	✓	✓	✓	✓	✓	✓
Elasticitate	✗	✗	✓	✓	✓	✗
Consistență	✓	✓	✓	✗	✗	✓
Disponibilitatea Citirii	✓	✓	✗	✓	✓	✗
Disponibilitatea Scrierii	✗	✗	✗	✓	✓	✗
Durabilitate	✓	✓	✓	✓	✓	✓

Tabel 3.2 – Compararea sistemelor de stocare pentru o serie de cerințe non-funcționale uzuale

Teoria CAP, creată de Eric Brewer în [5], stipulează faptul că ”din cele trei proprietăți ale sistemelor distribuite de stocare a datelor; consistența datelor, disponibilitatea sistemului și toleranța la partiționarea rețelei, pot fi atinse doar două în același timp”. Considerând faptul că ”în sistemele distribuite complexe, partiționarea rețelei se va întâmpla”, alegerea va fi între consistență și disponibilitate. [6]

În cazul nostru, se preferă consistența în detrimentul disponibilității, alegerea fiind un mod de stocare al datelor consistent și tolerant la partiționare. Totuși, deși MongoDB este un sistem de stocare CP (consistent și tolerant la partiționare), acesta oferă și un mecanism de creștere a disponibilității prin intermediul seturilor de replicare, însă alegerea acestei metode va prioritiza consistența în cazul în care va fi necesară o alegere dintre cele două.

Pentru a înțelege motivul din spatele acestei decizii, trebuie să fie înțeles modul de funcționare al sistemului MongoDB. În cazul unei căderi a conexiunii dintre membrii unui set de replicare, MongoDB va alege întotdeauna varianta care garantează consistența. Astfel, acesta nu va mai accepta cereri de scriere a datelor cât timp nu este sigur că scrierea acestora poate fi propagata la nivelul întregului set, asigurând consistența datelor prin sacrificarea disponibilității.

Proprietățile descrise în paragraful anterior pot fi observate și în tabelul de mai sus, evidențiindu-se lipsa disponibilității scrierii pentru MongoDB și prezența acestei proprietăți pentru sistemele AP (disponibile și tolerante la partiționare) precum Cassandra

și Riak. Scenariul descris anterior este unul extrem, însă teoria CAP tratează astfel de excepții și clasifică sistemele NoSQL de stocare a datelor în funcție de aceste cazuri.

3.2.4. Partiționarea datelor (Sharding)

Sharding-ul este o metodă de distribuire a datelor stocate într-o bază de date pe multiple noduri ale acelei baze de date. Această distribuire este posibilă prin împărțirea datelor dintr-o colecție în partiții și stocarea acestor partiții pe instanțe diferite ale sistemului.

Partiționarea datelor oferă numeroase avantaje, precum:

- Reducerea dimensiunii indecșilor, determinând o creștere a performanței.
- Posibilitatea scalării orizontale a stocării datelor din punct de vedere al performanței.
- Posibilitatea segmentării datelor după anumite criterii precum localizarea geografică a acestora.

Totodată, acest mod de gestionare a datelor creează și dezavantaje precum:

- Creșterea nivelului de interconectare dintre servere, rezultând în creșterea latenței interogărilor.

3.3. Soluții similare

În momentul de față, există un număr mare de sisteme similare cu Centrum, însă sistemul descris în această lucrare reușește să ocupe un loc liber între sistemele similare acestuia. În această secțiune, vom explora plasarea prototipului subiect al acestei lucrări prin listarea principalilor competitori și prin evidențierea avantajelor oferite de sistemul prezentat.

3.3.1. Microsoft Exchange Server

Comparația va începe cu Microsoft Exchange Server[1], un server de email dezvoltat de Microsoft. Acesta rulează exclusiv pe platforma Windows, rolurile de Mailbox și Edge Transport Server rulând pe Windows Server, iar utilitarele de management putând rula pe Windows Server sau Windows 8.1/10. Exchange Server a fost inițial dezvoltat pentru uz intern, fiind licențiat ulterior ca și un produs al companiei Microsoft ca software on-premises și ca un software-as-a-service (SaaS).

Această soluție este, momentan, cea mai completă de pe piață, având un ciclu foarte activ de dezvoltare. Totuși, Microsoft Exchange Server prezintă câteva limitări, printre care:

1. Poate rula doar pe sisteme de operare Windows. Astfel, Microsoft Exchange Server poate duce la incompatibilități cu alte sisteme sau produse, translatând, totodată, problema în costuri adiționale de operare, întrucât sistemele de operare Microsoft Windows 8.1/10 și Microsoft Windows Server nu sunt oferite gratuit.
2. Arhitectura este construită pe folosind un model monolitic. Acest neajuns se translatează într-o dificultate și o ineficiență în ceea ce privește scalarea serviciilor, limitând, de asemenea, și disponibilitatea serviciului în momentul instalării actualizărilor sau a reconfigurării produsului, crescând complexitatea

[1] https://docs.microsoft.com/en-us/exchange/

procesului de administrare. Un administrator va fi nevoit să treacă sistemul pe un server secundar în momentul actualizării sau reconfigurării acestuia, alternativa fiind o perioadă de inactivitate a sistemului. Problema persistă și în cazul prezentei unui server secundar, acesta fiind nevoit să dețină un număr de resurse adiționale specifice unei aplicații monolitice.

3. Este dificil de configurat, instalat și întreținut.

3.3.2. Google Apps for Business

A doua soluție pe care o vom explora este Google Apps for Business[2], o suită de servicii cloud oferite de Google, sucursală Alphabet Inc.. Această soluție nu necesită instalare și configurare, fiind ușor de setat și utilizat. Totodată, componenta de management al email-urilor se poate lega la diferiți clienți de email, precum Microsoft Outlook, Mozilla Thunderbird, Evolution, etc., oferind, simultan, și o soluție web completă.

Costând doar 5$ pe lună per utilizator și incluzând suport 24/7, aceasta este o soluție foarte bună pentru companii mici și mijlocii. Totuși, nicio soluție nu este perfectă, Google Apps for Business având propriile imperfecțiuni.

O problemă majoră pentru unele companii este faptul că toate datele utilizatorilor, cât și email-urile manipulate de sistem sunt stocate în centrele de stocare a datelor din cadrul companiei Google. Această imperfecţiune se manifestă prin limitarea controlului asupra datelor personale, fiind, totodată, un risc în ceea ce privește securitatea datelor sensibile

3.3.3. Zimbra Open-source

Zimbra[3] este o platformă de administrare a conturilor de poștă electronică oferită de Synacor, disponibilă în două versiuni: o versiune plătită și o versiune open-source. Acest sistem este independent de platformă, atât din perspectiva utilizatorilor cât și din perspectiva administratorilor de sistem.

Totodată, Zimbra suportă instalarea locală, oferind un grad mare de confidențialitate a datelor, însă instalarea, configurarea, administrarea și scalarea acestuia este dificilă comparativ cu o soluție dezvoltată cu ajutorului tehnologiilor precum Docker, Grafana și Rancher.

3.3.4. Kerio Connect

Kerio Connect[4] este o soluție complexă de mesagerie și colaborare, având caracteristici necesare întreprinderilor precum: posibilitatea administrării sarcinilor, întâlnirilor și a contactelor. Acest serviciu este țintit către mediul business, având și un preț corespunzător mediului.

Acesta poate fi folosit local, datele fiind păstrate pe serverele companiei, sau în cloud, datele fiind stocate pe serverele Kerio. Atât utilizarea, cât și instalarea locala sunt independente de platformă, gradul ce confidențialitate fiind determinat de locația instalării.

De asemenea, serviciul este dificil de instalat și configurat în cazul unei instalări locale, comparativ cu Zimbra sau Centrum.

[2] https://gsuite.google.com/
[3] https://www.zimbra.com/documentation/
[4] http://cdn.kerio.com/dwn/connect/connect-8.5.3-5082/kerio-connect-userguide-en-8.5.3-5082.pdf

3.3.5. Comparație

Comparativ cu soluțiile descrise anterior, Centrum este un prototip pentru un sistem de management al email-urilor cu o disponibilitate ridicată, o instalare rapidă, open-source, independent de platformă și o confidențialitate a datelor ridicată. Această comparație este reflectată în tabelul de mai jos.

Caracteristică	Microsoft ES	Google Suite	Zimbra Open-Source	Kerio Connect Cloud	Kerio Connect Local	Centrum
Open-source	✗	✗	✓	✗	✗	✓
SO Suportate pentru instalare	Windows 8.1/10, Server	Nu se aplică	Independent de platformă	Nu se aplică	Independent de platformă	Independent de platformă
Integrarea cu Dispozitive Mobile	Independent de platformă	Independent de platformă	Independent de platformă	Independent de platformă	Independent de platformă	Independent de platformă
Confidențialitatea Datelor	✓	✗	✓	✗	✓	✓
Instalare și administrare ușoară	✗	✓	✗	✗	✓	✓

Tabel 3.3 - Tabel comparativ între sistemul realizat și sisteme similare

Capitolul 4. Analiză şi Fundamentare Teoretică

4.1. Cerințe funcționale

Cerințele funcționale ale unei aplicații definesc o serie de funcții ale unui sistem și a componentelor acestuia, o funcție fiind definită ca fiind un set de valori de intrare, comportament sistemului sau al unei componente, și un set de valori de ieșire. Pentru o mai bună expunere a cerințelor funcționale ale sistemului descris, acestea au fost organizate în următorul tabel.

Cod	Descriere
CF1	**Gestionarea utilizatorilor**
CF1.1	Înregistrarea utilizatorilor
CF1.2	Autentificarea utilizatorilor
CF1.3	Deconectarea utilizatorilor
CF1.4	Resetarea parolei utilizatorilor
CF2	**Gestionarea conturilor de email**
CF2.1	Adăugarea conturilor
CF2.2	Ștergerea conturilor
CF3	**Gestionarea emailurilor**
CF3.1	Reîncărcarea listei agregate de emailuri
CF3.2	Citirea emailurilor
CF3.3	Răspunderea la emailuri
CF3.4	Compunerea și trimiterea emailurilor
CF3.5	Căutarea în lista de emailuri

Tabel 4.1 – Cerințe funcționale

Organizarea cerințelor funcționale a fost făcută pe categorii, cerințele majore fiind împărțite în mai multe componente. Astfel, putem distinge 3 categorii principale de cerințe funcționale: gestionarea utilizatorilor, gestionarea conturilor de email și gestionarea emailurilor.

Gestionarea utilizatorilor este reprezentată de operțiuni precum înregistrarea, autentificarea și deconectarea unui utilizator. Înregistrarea este realizată de utilizator prin introducerea numelui de utilizator dorit, adresei de email asociată cu contul, parolei, numelui și a prenumelui. În urma înregistrării, utilizatorul trebuie să confirme înregistrarea prin accesarea unei adrese unic generate trimisă de către server pe adresa de email asociată contului. În urma confirmării înregistrării este posibilă autentificarea utilizatorului prin intermediul numelui de utilizator introdus anterior și a parolei. Utilizatorul are posibilitatea de a reseta parola și de a se autentifica automat în cazul utilizărilor ulterioare a aplicației.

Următoarea categorie este reprezentată de gestionarea conturilor de email, permițând utilizatorului adăugarea și ștergerea conturilor de email atașate la contul său Centrum.

În final, ultima categorie cuprind acțiuni referitoare la managementul emailurilor agregate. Acestea acțiuni sunt principalele cerințe funcționale ale sistemului propus, constituind funcționalitatea principală a acestuia.

4.2. Cerințe non-funcționale

În domeniul ingineriei software, o cerință non-funcțională este o cerință utilizată pentru a judeca modul de operare a unui sistem. Acestea sunt deseori numite ”atribute calitative” ale unui sistem, evidențiind scopul acestora, cerințele non-funcționale fiind elemente descriptive ale calității utilizării unui sistem software dintr-un punct de vedere non-comportamental.

4.2.1. Performanță

Un factor foarte important pentru un utilizator al sistemului este performanța acestuia, performanță care poate fi împărțită în următoarele:

- **Timpul de raspuns**: Maxim 3 secunde timp de răspuns pentru aplicația web.
- **Capacitatea de încărcare**: Sistemul trebuie să susțină cel putin 200 de utilizatori concurenți și trebuie să fie capabil de a procesa cel puțin 100 de cereri pe secundă.

4.2.2. Utilizabilitate

Utilizabilitatea reprezintă gradul de ușurința în utilizarea unei aplicații de către un client sau consumator, cu scopul atingerii unor anumite obiective cuantificabile de efectivitate, eficiență și satisfacție. Acest criteriu este influențat în principal de designul interfeței aplicației, aceasta fiind nevoită să urmeze anumite standarde pentru a crea o experiență plăcută.

Pentru buna transparență a acestui criteriu, acesta va fi împărțit în următoarele cerințe:

- Ușurința în înțelegere:
  - Elementele componente ale interfeței (ex: meniuri, formulare) trebuie să fie ușor de înțeles.
  - Rolul sistemului trebuie să fie ușor de înțeles.
- Ușurința în învățare:
  - Administratorul de sistem trebuie să aibă la dispoziție un manual de utilizare.
  - Manualul trebuie să fie potrivit contextului și capabil de a explica modul de executare a sarcinilor comune.
  - Sistemul trebuie să fie ușor de învățat.
- Ușurința în operabilitate:
  - Elementele și acțiunile disponibile în interfață trebuie să fie consistente.
  - Mesajele de eroare trebuie să fie ușor de înțeles de către utilizatorii aplicației.

4.2.3. Disponibilitate

Sistemele de monitorizare, management și agregare trebuie să fie active în orice moment, sistemul de centralizare al emailurilor urmând să persiste orice date care nu s-au trimis sau nu au fost consumate.

Pentru realizarea redundanței sistemelor s-a implementat o soluție bazată pe containere Docker având setate o poliță de repornire a acestora în cazult apariției unor erori. Totodată, este posibilă implementarea unui sistem de load balancing al containerelor, clonând containerele active și redirecționând traficul în cazul unei căderi a sistemului.

Baza de date constă într-un cluster de MongoDB, având un nivel de disponibilitate ridicat datorită utilizării seturilor de replicare. În cazul unei defecțiuni la nivelul unui nod din cadrul replica seturilor, va avea lor înlocuirea nodului respectiv, chiar și în cazul căderii nodului principal a clusterului.

De asemenea, sistemul poate fi găzduit pe mai multe mașini din locații diferite, oferind un nivel foarte ridicat de disponibilitate.

4.2.4. Securitate

Unul dintre cele mai importante aspecte non-funcționale ale unui sistem este securitatea acestuia. În cazul aplicației descrise în această lucrare, nivelul de securitate este ridicat, acest aspect fiind implementat cu ajutorul următoarelor cerințe:

- Utilizarea protocolului de comunicare HTTPS pentru asigurarea confidențialității datelor aflate în tranzit.
- Stocarea datelor sensibile în formă criptată.
- Restricționarea bazată pe roluri a operațiilor posibile unui utilizator.
- Necesitatea autentificării utilizatorilor pentru accesarea sistemului.

4.2.5. Portabilitate

Gradul de portabilitate al sistemului este dat de ușurința de instalare și utilizare al acestuia, în contextul diferitelor platforme și sistemelor de operare existente. Având acest criteriu în atenție, Centrum reușește să atingă un nivel ridicat de portabilitate.

Din punctul de vedere al unui administrator de sistem, Centrum este usor de instalat și este independent de platformă, acesta funcționând pe orice sistem de operare suportat de Docker.

Din punctul de vedere al unui utilizator, este posibila accesarea aplicației folosind orice navigator web și orice sistem de operare pe care este posibila instalarea unui navigator.

4.3. Cazuri de utilizare

Secțiunea aceasta prezintă principalele cazuri de utilizare a sistemului cu ajutorul diagramelor expuse mai jos. Fiecare diagramă va fi însoțită de o descriere a acțiunii, actorului, precondițiilor și rezultatului.

Pentru ca un utilizator să poată interacționa cu sistemul propus, acesta trebuie să se autentifice. Prin urmare, primul caz de utilizare întâlnit de acesta este reprezentat de autentificarea sa în sistem. În urma realizării acestui pas, utilizatorul va fi redirecționat către pagina principală a aplicației. Următoarea diagramă descrie principalul caz de utilizare al sistemului.

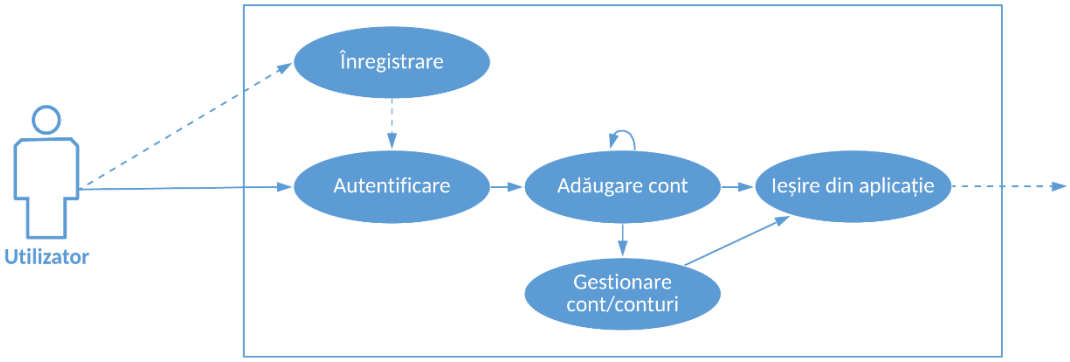

Figura 4.1 – Diagrama cazului principal de utilizare al sistemului

Caz de utilizare: Înregistrare
Actor: Utilizator neînregistrat
Descriere:

1. Utilizatorul accesează pagina de înregistrare, introduce datele necesare (nume de utilizator, email, parolă, nume, prenume) și confirmă introducerea datelor prin apăsarea butonului ”Register”.
2. După procesarea și validarea datelor introduse, acesta primește un email pe adresa asociată anterior contului. Utilizatorul deschide într-un navigator web adresa primită, confirmând înregistrarea și fiind redirecționat către pagina principală a aplicației.

Precondiții:

- Utilizatorul trebuie să acceseze pagina de înregistrare.
- Utilizatorul trebuie să introducă o adresă de email validă.
- Utilizatorul trebuie să introducă o parolă suficient de complexă.

Rezultat:

- În cazul înregistrării cu succes, utilizatorul poate începe să utilizeze sistemul
- În cazul unei erori, utilizatorul este nevoit să încerce din nou procesul de înregistrare.

Caz de utilizare: Autentificare
Actor: Utilizator înregistrat
Descriere:

1. Utilizatorul accesează pagina de autentificare, introduce datele necesare (nume de utilizator și parolă) și confirmă autentificarea prin apăsarea butonului ”Login”.

Precondiții:

- Utilizatorul trebuie să acceseze pagina de autentificare.
- Utilizatorul trebuie să introducă o combinație validă de nume de utilizator și parolă.

Rezultat:

- În cazul autentificării cu succes, utilizatorul poate să utilizeze sistemul

- În cazul unei erori, utilizatorul este nevoit să încerce din nou procesul de autentificare.

În urma autentificării utilizatorului în sistem, acesta va putea utiliza întregul spectru de acțiuni oferite de aplicația dezvoltată. În continuare sunt evidențiate cazurile de utilizare posibile unui utilizator autentificat.

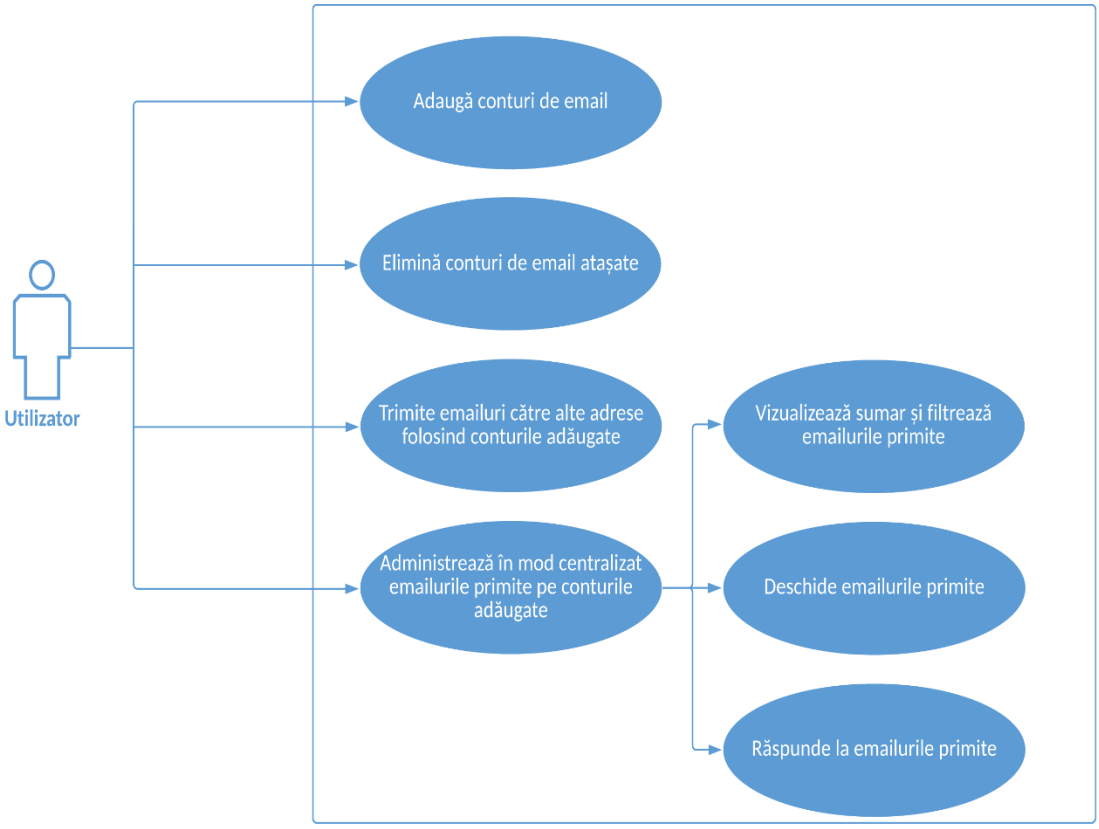

Figura 4.2 – Diagrama cazurilor de utilizare posibile unui utilizator autentificat

Caz de utilizare: Adăugarea conturilor de email
Actor: Utilizator autentificat
Descriere:

1. Utilizatorul deja autentificat accesează panoul de control aferent contului prin selectarea opțiunii ”Dashboard” a meniului oferit în colțul dreapta-sus al paginii, fiind etichetat cu numele de utilator al persoanei autentificate în sistem.
2. Odată ajuns în panoul de control, utilizatorul selectează din meniul din partea stângă a paginii opțiunea ”Accounts”, urmată de sub-opțiunea ”Add Account”.
3. Utilizatorul introduce un nume pentru noul cont, adresa de email ce dorește să fie atașată, împreună cu parola contului de email.
4. În urma introducerii datelor necesare adăugării contului, utilizatorul trebuie să apese butonul ”Save and Continue”, urmând ca sistemul să proceseze cererea acestuia și să îl notifice în scurt timp de rezultatul acțiunii sale.

Precondiții:

- Utilizatorul să fie autentificat.

Rezultat:

- În cazul adăugării cu succes al contului, utilizatorul va fi notificat de succesul operațiunii și va putea începe administrarea emailurilor din contul respectiv.
- În cazul unei erori, utilizatorul este nevoit să încerce din nou procesul de adăugare al contului.

Caz de utilizare: Eliminarea conturilor de email atașate
Actor: Utilizator autentificat
Descriere:

1 Utilizatorul deja autentificat accesează panoul de control aferent contului prin selectarea opțiunii ”Dashboard” a meniului oferit în colțul dreapta-sus al paginii, fiind etichetat cu numele de utilator al persoanei autentificate în sistem.
2 Odată ajuns în panoul de control, utilizatorul selectează din meniul din partea stângă a paginii opțiunea ”Accounts”, urmată de sub-opțiunea ”List Account”.
3 Utilizatorul alege dintr-un tabel contul de email dorit și îl elimină din sistem prin apăsarea butonului de ștergere aflat în ultima coloană al tabelului. (*Opțional*) *Utilizatorul poate filtra prin lista de emailuri atașate prin apăsarea butonului ”Filter” aflat în antetul tabelului.*

Precondiții:

- Utilizatorul să fie autentificat.
- Utilizatorul trebuie să fi atașat sistemului cel puțin un cont de email.

Rezultat:

- În cazul eliminării cu succes al contului, utilizatorul va fi notificat de succesul operațiunii.
- În cazul unei erori, utilizatorul este nevoit să încerce din nou procesul de eliminare al contului.

Caz de utilizare: Trimiterea emailurilor folosind conturile adăugate
Actor: Utilizator autentificat
Descriere:

1. Utilizatorul autentificat accesează formularul de trimitere al emailurilor prin selectarea opțiunii ”Compose” al meniului din antetul paginii principale al aplicației.
2. Utilizatorul completează formularul, selectând adresa de pe care acesta dorește să trimită emailul, completând câmpurile ”To”, ”Cc” și ”Subject” cu valori corespunzătoare și redactând conținutul corespondenței utilizând editorul de text disponibil în pagină.
3. Utilizatorul confirmă trimiterea emailului prin apăsarea butonul ”Send”.

Precondiții:

- Utilizatorul să fie autentificat.
- Utilizatorul trebuie să fi atașat sistemului cel puțin un cont de email.

Rezultat:

- Emailul redactat va fi trimis adreselor specificate în câmpurile ”To” și ”Cc”.

Caz de utilizare: Administrarea centralizată a emailurilor
Actor: Utilizator autentificat
Descriere:

1. Utilizatorul autentificat vizualizează, în pagina principală a aplicației, emailurile agregate din conturile de email atașate de acesta.
2. *(Opțional) Utilizatorul poate filtra emailurile folosind căsuța de căutare disponibilă în pagină.*
3. *(Opțional) Utilizatorul poate deschide emailurile pentru a vedea întregul conținut al acestora prin simpla selectare a emailului dorit.*
4. *(Opțional) Utilizatorul poate compune emailuri direct din vederea descrisă la punctul 3 prin accesarea butoanelor ”Reply”, ”Reply All” sau ”Forward”.*

Precondiții:

- Utilizatorul să fie autentificat.
- Utilizatorul trebuie să fi atașat sistemului cel puțin un cont de email cu cel puțin un email stocat.

Rezultat:

- În cazul redactării unui email, acesta va fi trimis adreselor specificate în câmpurile ”To” și ”Cc”.

După cum se poate observa, sistemul propus oferă utilizatorului opțiunile necesare administrării emailurilor. Însă, la nivelul unei organizații, sistemul va fi instalat și administrat de către unul sau mai mulți administratori. Aceștia vor avea control deplin asupra sistemului și vor fi asistați în cadrul acțiunii de administrare de către utilitarele de monitorizare disponibile în cadrul sistemului.

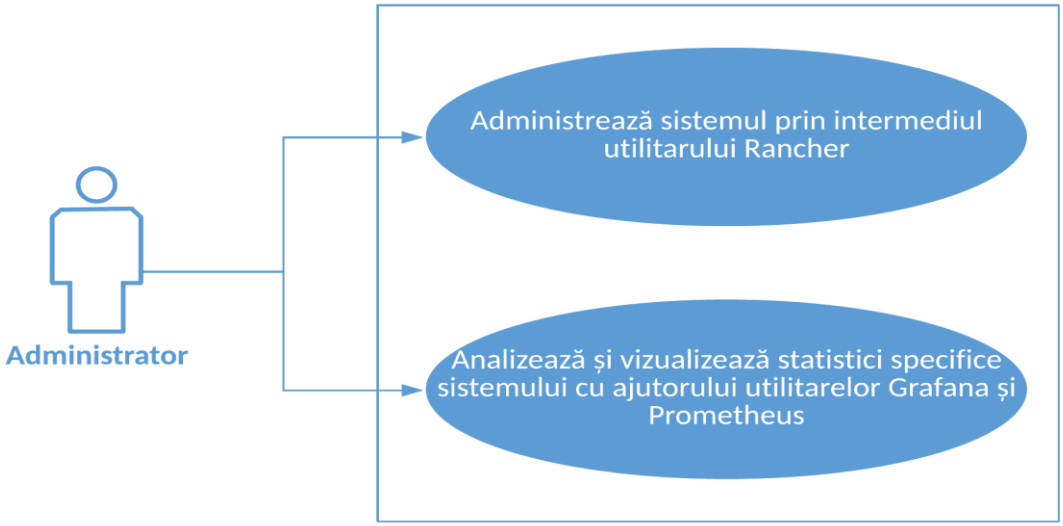

Figura 4.3 – Diagrama cazurilor de utilizare posibile unui administrator

Caz de utilizare: Administrarea sistemului prin intermediul utilitarului Rancher
Actor: Administrator
Descriere:

1. Administratorul se autentifică în Rancher.
2. În urma autentificării, acesta administrează sistemul folosind acțiunile disponibile, precum: adăugarea de noi hosturi, scalarea instanțelor aplicației web sau a aplicației server, adăugarea de load balancere sau proxyuri, etc.

Precondiții:

- Administratorul trebuie să fie înregistrat în sistem

Rezultat:

- În cazul autentificării cu succes, administratorul poate utiliza utilitarul Rancher pentru a administra sistemul.
- În cazul unei erori, acesta este nevoit să încerce din nou procesul de autentificare.

Caz de utilizare: Monitorizarea sistemului folosind Grafana și Prometheus
Actor: Administrator
Descriere:

1. Administratorul se autentifică în Grafana.
2. În urma autentificării, acesta monitorizează sistemul folosind panourile disponibile în aplicație. *(Opțional) Administratorul are posibilitatea de a seta alerte pentru a fi notificat în cazul apariției unui comportament defectuos al sistemului.*

Precondiții:

- Administratorul trebuie să fie înregistrat în sistem

Rezultat:

- În cazul autentificării cu succes, administratorul poate utiliza utilitarul Grafana pentru a monitoriza sistemul.
- În cazul unei erori, acesta este nevoit să încerce din nou procesul de autentificare.

4.4. Perspectiva tehnologică

4.4.1. Oracle Java 8

Java[5] este un limbaj de programare puternic tipizat și orientat-obiect conceput de către James Gosling la compania Sun Microsystemes (achiziționată de Oracle în ianuarie 2010). Limbajul java împrumută o mare parte din sintaxă de la limbaje tradiționale precum C și C++, însă dispune de un model mai simplu al obiectelor și prezintă mai puține facilități de nivel jos.

Acest limbaj a fost construit cu scopul de a oferi programatorilor un limbaj în care aceștia să pot dezvolta aplicații capabile să ruleze pe orice sistem. Astfel, codul Java compilat poate rula pe orice platformă fără a necesita o recompilare, caracteristică datorată mașinii virtual pe care codul Java este executat. Pentru a atinge acest nivel de portabilitate,

[5] https://docs.oracle.com/javase/8/docs/

codul Java este compilat într-o reprezentare intermediară, numită Java bytecode, cod care se execută pe mașina virtuală Java (JVM). Instrucțiunile din reprezentarea intermediară a codului au o reprezentare analoagă în cod mașină. Prin urmare, mașina virtuală Java este asemenea unui translator între hardware și codul Java, traducând instrucțiunile scrise în codul mașină specific arhitecturii pe care este rulată mașina virtuală. [4]

Totodată, Java asigură și managementul memoriei unei aplicații prin intermediul garbage collector-ului (GC). Acesta permite programatorilor să creeze obiecte noi, fără grija alocării și dealocării memoriei, întrucât colectorul are rolul de a revendica automat memoria pentru a fi refolosită. Astfel, dezvolarea aplicațiilor în limbajul Java este rapidă și conține puțin cod de administrare a resurselor, eliminând, simulan, scăpările de memorie și alte probleme de alocare și dealocare.

Având în vedere numele sistemului de management al memoriei, este posibilă interpretarea greșită a modului de acțiune a acestuia. Se poate crede că acest sistem colectează și înlătură obiectele ”moarte”. În realitate, colectorul face operația opusă celei descrise anterior, obiectele ”vii” fiind urmărite, iar restul fiind marcate pentru ștergere.

Pentru determinarea obiectelor inactive, JVM-ul rulează intermitent un algoritm numit ”marchează și șterge”. După cum se poate concluziona din numele acestuia, algoritmul este un proces simplu cumpus din doi pași:

1. Algoritmul traversează toate obiectele referențiate, începând cu rădăcinile codului colector și marchează orice obiect găsit ca fiind ”viu”.
2. Întregul spațiu de memorie neocupat de obiectele marcate la pasul anterior este revendicat.

În continuare vom detalia câteva concepte fundamentale ale limbajului și platformei Java, precizate în [5].

4.4.1.1. Practicalitate

James Gosling, creatorul limbajului Java, a descris acest limbaj de programare ca fiind un muncitor, fiind proiectat cu scopul de a da dezvoltatorilor posibilitatea de a implementa sisteme complexe cu un minim de efort, oferind totodată dezvoltatorilor capabilitatea de a întelege și întreține codul în viitor. Bineînteles, este posibilă scrierea unui cod ininteligibil în Java, așa cum este posibil în orice limbaj de programare, însă prin folosirea unor convenții de scriere a unui cod curat, acesta devine mai lizibil comparativ cu majoritatea limbajelor de programare.

4.4.1.2. Compatibilitate

Companiile Sun și ulterior Oracle au efectuat eforturi majore cu scopul de a asigura funcționarea codului scris pentru o versiune de Java pe orice versiune mai nouă a limbajului de programare. Deși au existat excepții de la această regulă, precum assert-urile din Java SE 1.4 și enumerațiile din Java SE 5, această caracteristică a limbajului Java este foarte convingătoare pentru dezvoltatori în alegerea unui mediu de programare a unui viitor proiect. Nevoia de a modifica codul scris pentru a obține funcționarea aplicației cu o nouă versiune a limbajului de programare folosit aduce o datorie tehnică considerabilă.

4.4.1.3. Scalabilitate, performanță și fiabilitate

Având peste douăzeci de ani de dezvoltare și mii de ani de muncă cumulată, Java este o platformă solidă și fiabilă. Aceasta funcționează la un nivel apropiat, chiar superior

în unele situații, cu un cod nativ, fapt datorat optimizărilor dinamice ale JVM-ului. Optimizarea codului la nivelul mașinii virtuale Java are loc atât static, cât și dinamic, prin intermediul analizei statistice. Un exemplu este reprezentat de ”statificarea” anumitor variabile în timpul rulării programului în cazul în care utilizarea acestora devine suficient de frecventă, fiind posibilă antrenarea mașinii virtuale cu date simulate înainte de deschiderea accesului către aplicație.

De asemenea, Java oferă un nivel de scalare extrem de ridicat. Exemple clare se pot observa în cazul companiilor mari precum:

- **Twitter**, care a recurs la mutarea aplicațiilor Ruby-on-Rails în JVM din cauza dificultății de scalare a acestora.
- **Spotify**
- **Facebook**
- **Salesforce**
- **eBay**
- **Oracle**

Totodată, **Hadoop**, **Cassandra** și **Spark**, baze pentru majoritatea proiectelor big data, sunt scrise in Java sau Scala și rulează în interiorul mașinii virtuale Java, evidențiind nivelul de performanță și scalare a aplicațiilor Java.

4.4.1.4. Dezvoltare proactivă și continuă

Acesta este unul dintre cele mai puternice puncte forte ale limbajul Java. Conform TIOBE, a avut loc o creștere semnificativă în popularitatea acestui limbaj, creștere care a început din Octombrie 2014, la scurt timp după lansarea versiunii 8 a limbajului. Oracle Java 8 a fost o schimbare majoră pentru dezvoltatori care folosesc Java datorită introducerii expresiilor Lambda și a API-ului de precesare a fluxului de date (Java Stream API).

Din momentul introducerii acestor noi caracteristici, dezvoltatorii Java au putut începe a programa într-un mod mai apropiat de paradigma de programare funcțională fără a fi nevoiți să învețe un limbaj nou, precum Scala. Totodată, aceste caracteristici simplifică programarea într-un mediu concurent, eliminând necesitatea de a scrie mult cod complex și predispus la erori de sincronizare.

Cu ajutorul proiectului de modularizare Jigsaw implementat în Java 9, aplicațiile complexe au devenit mult mai ușor de dezvoltat, instalat și întreținut, prin intermediul modularității proiectului.

Totodată, Java urmează să fie actualizat mai des în viitor pentru a ține pasul cu noile cerințe software. Oracle a anunțat o nouă foaie de parcurs a proiectului având un nou ciclu de lansare. Astfel, o versiune nouă a kit-ului de dezvoltare va fi lansată din șase în șase luni, urmând ca o versiune cu suport prelungit să fie lansată odată la trei ani. Prin urmare, noi caracteristici vor fi disponibile dezvoltatorilor mai rapid, păstrând, simultan, stabilitatea proiectului.

De asemenea, odată cu Java 9 a fost inaugarat un nou mecanism de dezvoltare a platformei, prin intermediul propunerii de îmbunătățirii a kit-ului de dezvoltare JEP11.

4.4.2. Spring Framework

Spring Framework[6] este un set de librării Java construit cu scopul de a ușura dezvoltarea aplicațiilor enterprise. Spring este un framework modular, oferind dezvoltatorilor de aplicații posibilitatea alegerii componentelor necesare pentru aplicația dezvoltată și eliminarea componentelor neutilizat pentru un management mai bun al resurselor.

Totodată, setul de librării Spring suporta managementul declarativ al tranzacțiilor, accesul la logica aplicației prin RMI sau cu ajutorul serviciilor web, și o variație de moduri de persistare a datelor.

Spring este proiectat cu scopul de a fii non-intruziv, eliminând dependințele între logica componentei de domeniu a aplicației și framework-ul propriu-zis. Totuși, în componentele de integrare a aplicației, precum componenta de acces a datelor, câteva dependințe vor apărea, însă acestea vor fi ușor de izolat în raport cu restul codului din aplicație.

De asemenea, Spring oferă un model cuprinzător de programare și configurare a aplicațiilor Java, oferind suport pentru injectarea dependințelor, procesarea evenimentelor, administrarea resurselor, internaționalizarea aplicației, validarea datelor, programarea orientată pe aspect, testarea prin intermediul mock-urilor, accesarea datelor din diverse medii de stocare și multe altele.

În continuare vor fi enumerate și descrise sumar principalele componente ale setului de librării Spring care sunt utilizate în dezvoltarea aplicației descrise în această lucrare:

- **Spring Beans:** Obiecte fundamentale are unei aplicații Spring, administrate de containerul de inversiune a controlului și create pe baza unor configurări transmise containerului.
- **Spring Context:** Reprezintă containerele de inversiune a controlului responsabile cu instanțierea, configurarea și asamblarea bean-urilor prin agregarea configurațiilor specifice acestora din fișiere XML, anotări Java și/sau cod Java.
- **Spring Core:** Este containerul principal al colecției de librării Spring, realizând operațiuni importante dezvoltării unei aplicații, precum: injectarea dependințelor, prelucrarea evenimentelor și a resurselor, internaționalizarea, validarea și legarea datelor, conversia tipurilor și programarea orientată pe aspect.
- **Spring Data:** Este un sistem de mapare a obiectelor într-un mediu de stocare, având scopul de a furniza un model de acces al datelor familiar și consistent.
- **Spring Security:** Librărie a cărei scop principal este asigurarea serviciilor de autentificare și autorizare.
- **Spring MVC:** Un framework web bazat pe Java Servlet API, având rolul de a simplifica dezvoltarea aplicațiilor web sau a aplicațiilor ce deservesc aplicații web

[6] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/

4.4.3. REST

REpresentational **S**tate **T**ransfer sau **REST** este un mod de interoperabilitate între dispozitivele cu acces la internet. REST este deseori comparat cu SOAP, însă această comparație nu este în totalitate validă, întrucât SOAP este un protocol de comunicare, REST fiind un stil arhitectural.

Principalele caracteristici ale stilului arhitectural și de comunicare REST sunt următoarele:

- REST este un protocol independent, decuplat de HTTP (HyperText Transfer Protocol). Astfel, o aplicație REST poate utiliza orice protocol reprezentat printr-o schemă URI standardizată, navigarea printr-un API REST fiind posibilă asemănător cu urmărirea link-urilor URL de pe paginile web.
- Posibilitatea scalării la un număr mare de componente și interacțiuni între acestea.
- REST oferă simplicitatea unei interfețe uniforme, standardizate.
- Capabilitatea de a modifica componentele aplicației pentru a suporta constrângeri noi, modificarea putând avea loc în timpul rulării aplicației.
- Portabilitatea componentelor prin mutarea codului aplicației împreună cu datele utilizate de aceasta.
- Asigurarea unui mediu de comunicare transparent.

4.4.4. Angular 6

Angular[7] este un framework de JavaScript, open-source și orientat pe partea de front-end. Scopul acestuia este augmentarea aplicațiilor web prin implementarea capabilității de suportare a modelului arhitectural MVC (Model-View-Controller) și prin reducerea liniilor de cod JavaScript necesare pentru a crea respectiva aplicație. Tipurile de aplicații dezvoltate cu ajutorul framework-ului Angular 6 se mai numesc colocvial ”Aplicații Uni-Pagină” (Single-Page Applications).

Acest framework are la îndemână un set impresionant de utilități specifice limbajului Javascript, este ușor de învățat și perfomant. Totodată, Angular suportă Typescript, un superset al limbajului Javascript construit pentru dezvoltarea orientată pe obiect, rapidă și modulară a aplicațiilor web. Prin utilitățile disponibile dezvoltatorilor de aplicații Angular se numără și **RxJS**, o librărie Javascript care permite comunicarea asincronă între servicii prin intermediului unui model ”push”.

De asemenea, Angular este sprijinit de Google și de milioanele de dezvoltatori care îl folosesc, fiind ușoară găsirea de soluții pentru majoritatea problemelor întâmpinate în dezvoltarea aplicației, motiv pentru care s-a optat pentru acest framework.

4.4.5. MongoDB

După cum se poate observa în analiza comparativă dintre MongoDB[8] și alte sisteme similare acestuia, descrisă în capitolul 3 al acestei lucrări, MongoDB este soluția care se potrivește cel mai mult nevoilor sistemului.

[7] https://angular.io/docs
[8] https://docs.mongodb.com/

Avantajele acestui sistem de stocare, precum posibilitarea partiționării și segmentării datelor, suportul pentru căutare textuală, consistența datelor și organizarea acestora în structuri de tip document, sunt perfect complementare cu nevoile aplicației dezvoltate. Totodată, dezavantajele sistemului nu au impact major asupra aplicației. Lipsa tranzacțiilor ACID și a join-urilor, precum și stocarea datelor fără o schemă predefinită, nu sunt criterii necesare bunei funcționări a sistemului descris.

4.4.6. Docker

O problemă a arhitecturii bazate pe microservicii este prezența multor componente dinamice, acestea complicând procesul de dezvoltare a aplicației. Astfel, conceptul imutabilității livrării unui produs a ajuns un concept principal în lumea dezvoltării microserviciilor. Prin imutabilitatea livrării se încearcă reducerea numărului de componente dinamice cu ajutorul unor imagini construite anticipat. Cu ajutorul acestor imagini putem crea diverse medii de dezvoltare si livrare, medii deja configurate, eliminând mare parte din datoria tehnică necesară instalării.

Docker[9] este o soluție de management a containerelor, creată cu scopul de a elimina problema descrisă în paragraful anterior, eliminând totodată cazurile în care aplicația dezvoltată rulează doar pe anumite mașini și configurații.

Creat de Solomon Hykes, un inginer al dotCloud, o companiei de dezvoltare a soluțiilor PaaS, Docker oferă o modalitate de împachetare a aplicațiilor, alături de toate dependințele necesare rulării, în imagini de dimensiuni relativ mici.

Totodată, Docker poate crea instanțe ale acestor imagini, cu scopul de a rula aplicații într-un container Linux cu acces izolat la componentele hardware ale sistemului, precum: procesor, memorie, rețea și spațiu de stocare. Se poate spune că aceste containere sunt un mod de virtualizare la nivelul unei aplicații sau a unui proces, permițând unui proces să ruleze ca și cum ar avea acces total la partea hardware a sistemului pe care containerul este găzduit, în realitate fiind posibilă doar accesarea resurselor alocate containerului pe care procesul rulează.

De exemplu, se poate porni un container Docker care să folosească un anumit procent din procesor, un segment din memoria RAM și având o limită asupra lungimii de bandă pe care o poate ocupa în rețea. Din exteriorul containerului, pe mașina gazdă, aplicația arată ca un proces obișnuit, fără virtualizări ale driverelor, sistemului de operare, rețelelor conectate, și fără hipervizori speciali. Astfel, este posibilă rularea mai multor containere pe aceeași mașină, crescând eficiența sistemului fără a adăuga costuri suplimentare specifice unui sistem de operare adițional și a unei mașini virtuale clasice, scopuri ce ar fi fost necesare pentru a atinge același nivel de izolare.

Cu toate acestea, modul de funcționare nu este unul revoluționar. Elemente precum *namespaces*, *cgroups* și *chroot* au existat în nucleul sistemului de operare Linux de ani buni, acestea putând fi folosite pentru a crea iluzia virtualizării unei aplicații. Containerele de linux există de mai mult de 10 ani, procesul de virtualizare specificat anterior fiind prezent în sistemele de operare Solaris și FreeBSD înaintea acestora. Totuși, utilizarea acestor primitive Linux sau a abstractizărilor acestora, precum *lxc*, obișnuia să fie o sarcină foarte complexă. Apariția Docker-ului a constituit o simplificare a API-ului și a experienței de dezvoltare și de integrare a containerelor Linux. Docker a inovat prin introducerea unei

[9] https://docs.docker.com

interfețe unice care oferă opțiuni avansate de management al containerelor, containere bazate pe imaginii descrise într-un format bine specificat, revoluționând modul de împachetare și livrarea a aplicațiilor.

Odată ce avem o imagine, este posibilă crearea cu ușurință a unui număr mare de containere bazate pe imaginea respectivă. Imaginile pot depinde una de cealaltă și pot fi structurate în straturi construite pe baza diferențelor dintre o imagine de bază și fișierele specifice aplicației, eficientizând astfel distribuție imaginilor prin eliminarea nevoii de a distribui o imagine completă, diferențele fiind suficiente pentru crearea unei instanțe a imaginii dorite. Totodată, în cazul descoperirii unei vulnerabilități la nivelul unei imagini de bază, aceste imagini pot fi reconstruite fără a fi nevoie de refacerea fiecărei mașini virtuale.

De asemenea, un container este foarte ușor de rulat și de distribuit, fiind posibilă mutarea acestuia de pe mașina unui dezvoltator pe alte medii de dezvoltare și testare sau în producție, distribuția fiind portabilă și fără grija copierii dependințelor necesare pentru rularea aplicației.

În concluzie, Docker este platforma ideală pentru suportul arhitecturii sistemului propus.

4.4.7. Rancher

Rancher[10] este o platformă software open-source care permite organizațiilor să ruleze și să administreze Docker și Kubernetes în medii de producție. Cu ajutorul lui Rancher nu este nevoie construirea unui sistem de gestionare a containerelor, acesta fiind capabil de a orchestra o infrastructură bazată pe containere.

La baza platformei Rancher se află următoarele concepte majore:

- **Orchestrarea infrastructurii**. Rancher se folosește de resursele oricărei colecții de hosturi de Linux gestionată în cloud, precum Amazon Web Services, Microsoft Azure sau serverele proprii ale unor companii. Hosturile de Linux pot fi mașini virtuale sau mașini fizice, Rancher având nevoie doar de resursele hardware alocate acesteia. Din perspectiva platformei de orchestrare, diferența dintre o instanță a unei mașini virtuale de la un furnizor de servicii cloud și un server găzduit local este imperceptibilă.
- **Orchestrarea și programarea containerelor**. Mulți utilizatori ai containerelor folosesc un framework de orchestrare și programare al acestora. Rancher include în suita sa de aplicații cele mai populare platforme de orchestrare, incluzând Docker Swarm, Kubernetes și Mesos. Un utilizator poate crea multiple clustere de containere folosind utilitarele amintite anterior și poate folosi uneltele native specifice utilitarului folosit pentru a administra clusterul creat.
- **Catalogul de aplicații**. Utilizatorii platformei Rancher au posibilitatea lansării unor clustere cuprinse dintr-un număr mare de containere printr-un singur buton, folosind catalogul de aplicații. Inginerii de sistem pot administra aplicațiile lansate și pot realiza actualizări automate ale acelor aplicații în momentul în care o nouă versiune este disponibilă. Rancher este conectat implicit la două cataloage publice ce cuprind un număr mare de

[10] https://rancher.com/docs/rancher/v1.6

aplicații populare în lumea dezvoltării de aplicații software, însă utilizatorii platformei nu sunt limitați la aceste aplicații implicite, având posibilitatea creării propriilor cataloage private sau publice.

- **Controlarea accesului**. Pentru a asigura siguranța infrastructurii, Rancher suportă diverse moduri de autentificare a utilizatorilor, precum Active Directory, LDAP, autentificare prin GitHub și autentificare prin nume de utilizator și parola. Adițional, platforma oferă un control de acces bazat pe roluri, fiind posibilă administrarea granulară a permisiilor unui utilizator.

În concluzie, Rancher este un utilitar crucial în găzduirea aplicației și în administrarea eficientă a containerelor descrise la pasul anterior.

4.4.8. Nexus Repository Manager

Nexus[11] este un manager de repertorii, oferind utilizatorilor posibilitatea de a colecta și administra dependințele necesare dezvoltării aplicațiilor. Acesta se folosește în principal pentru stocarea librăriilor utilizate, precum dependințe Maven, însă se poate folosi și pentru stocarea privată a imaginilor Docker. Astfel, putem salva și versiona imaginile construite cu scopul de a le folosi în viitor fără a fi necesară reconstrucția acestora.

În cazul Centrum, un manager de repertorii precum Nexus este un element foarte important al componentei de administrare, întrucât Rancher nu este capabil de construirea imaginilor. Prin urmare, acestea trebuie construite local sau într-un sistem de integrare și dezvoltare continuă, precum Jenkins, și salvate într-un repertoriu accesibil Rancher, precum Nexus.

4.4.9. Gradle

În trecut, build-ul unei aplicații avea simpla responsabilitate de a compila și împacheta respectiva aplicație, însă acest pas a necesitat schimbări pentru a se acomoda la evoluția dezvoltării a aplicațiilor, care a introdus nevoia automatizării pașilor de build a aplicației. În prezent, proiectele din domeniul software implică lucruri cu stive tehnologice complexe, de mari dimensiuni, având incorporate mai multe limbaje de programare și aplicând un spectru larg de strategii de testare. Odată cu creșterea în utilizare a metodologiei Agile, build-ul aplicației trebuie să suporte atât integrări timpurii ale codului scris în diferite limbaje de programare, cât și o livrare rapidă și ușoară a aplicației în mediile de testare și producție.

Gradle[12] reprezintă un pas în evoluția sistemelor de automatizare a build-urilor bazate pe JVM. Acesta este construit pe baza conceptelor specifice altor sisteme, precum Apache Ant și Apache Maven, introducând un limbaj bazat pe Groovy cu scopul de a descriere configurația proiectelor. Întrucât Gradle este nativ JVM-ului, este posibilă scrierea unor logici complexe în limbaj Java sau Groovy.

Gradle este un sistem open-source de automatizare a buildului unei aplicații. Acesta este construit pe baza conceptelor specifice altor sisteme de automatizare precum Apache Ant și Apache Maven, introducând un limbaj de descriere a configurării proiectelor bazat pe Groovy. [6]

Principalele caracteristici ale acestui utilitar, utilizabile de sistemul propus, sunt:

[11] https://help.sonatype.com/repomanager3
[12] https://docs.gradle.org/current/userguide/userguide.html

- Versitalitatea reflectată prin posibilitatea gestionării build-ului pentru un număr mare de limbaje și platforme, împreună cu capacitatea de a gestiona mai multe reportorii de proiecte simultan.
- Ușurința de automatizare a proceselor. Gradle este livrat împreună cu un API bogat și cu un ecosistem de plugin-uri foarte matur, oferind posibilitatea modelării, integrării și sistematizării livrării unei aplicații.
- Rapiditatea construirii aplicațiilor, realizată prin intermediul mecanismelor de caching și evitare a compilării.

Prin urmare, Gradle este utilitarul potrivit organizării procesului de construire a proiectului propus.

4.4.10. Prometheus

Una dintre cele mai importante responsabilități a unui administrator de sistem este monitorizarea sistemului. Aceasta constă în abilitatea de a cunoaste starea sistemului la un moment dat de timp, starea putând reprezenta metrici precum: resursele utilizate, comenzile rulate, traficul de date în interiorul sistemului, etc.

Prometheus[13] este un sistem open-source de monitorizare și alertare adoptat de un număr mare de companii încă de la începuturile sistemului, în 2012.

Prometheus funcționează prin extracția unor seturi de date din diferite puncte ale infrastructurii. Există un număr ridicat de exportatori ale acestor date, fiind totodată posibilă crearea unor exportatori proprii sau exportarea datelor la nivelul unei aplicații prin intermediul librăriilor de Java, Go, Python și Ruby oferite de echipa de dezvoltare a sistemului Prometheus.

Principalele caracteristici ale lui Prometheus, aplicabile în sistemul propus, sunt următoarele:

- Model de date multidimensional bazat pe vizualizarea datelor în timp după numele metricii și perechi cheie/valoare.
- Un limbaj de interogare flexibil capabil de a gestiona multidimensionalitatea datelor stocate.
- Independența nodurilor și eliminarea necesității unui sistem de stocare distribuit.
- Colectarea datelor prin apeluri HTTP.
- Suportarea multiplelor modalități de afișare și interactionare cu datele stocate.
- Descoperirea și configurarea dinamică sau statică a exportatorilor folosiți în colectarea datelor.
- Posibilitatea pasării datelor către Prometheus prin intermediul unor porți intermediare la nivel de aplicație.

Instalarea sistemului este simplă, fiind documentată foarte bine pe pagina web a sistemului.

[13] https://prometheus.io/docs/

4.4.11. Grafana

Grafana[14] este o platformă open-source de analiză și vizualizare a datelor, concepută pentru a lucra împreună cu surse de date precum Prometheus, Elasticsearch, InfluxDB si Graphite. Modul de folosirea a acestei aplicații se plasează în jurul căutării, vizualizării și interacțiunii cu datele stocate în instanțele de Prometheus conectate la acesta, fiind posibilă afișarea acestora în diverse formate.

Astfel, Grafana ușurează analiza volumelor mari de date, oferind o interfață web cu ajutorul cărora se pot crea și partaja spații dinamice de afișare în timp real a interogărilor Prometheus.

[14] http://docs.grafana.org/

Capitolul 5. Proiectare de Detaliu si Implementare

În acest capitol se va descrie proiectarea sistemului, incluzând aplicația în sine, componenta de monitorizare, componenta de administrare și structura bazei de date. Vor fi prezentate diagrama arhitecturii alese, arhitectura, schema de instalare a aplicației și diagramele de clasă pentru componentele principale ale aplicației.

5.1. Arhitectura sistemului

5.1.1. Arhitectura generală

Arhitectura întregului sistem respecta modelul unei arhitecturi bazate pe microservicii, având un nivel mare de granularitate. Interacțiunea cu clientul este realizată prin aplicația web din cadrul segmentului front-end al sistemului, iar interacțiunea cu administratorul sistemului va fi preluată, în funcție de context, de componenta de monitorizare a aplicației (Grafana) sau de componenta de administrare a infrastructurii (Rancher).

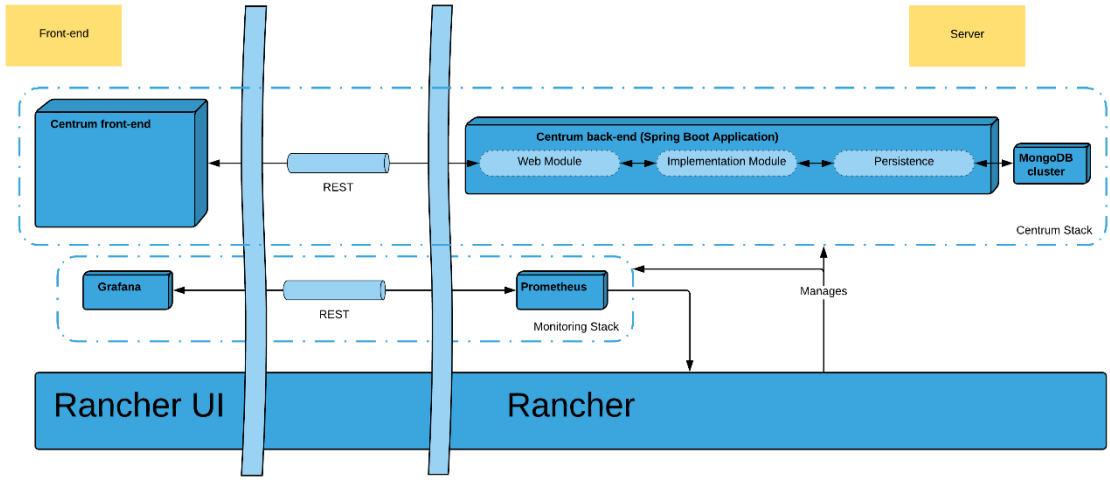

Figura 5.1 - Arhitectura generală a sistemului

5.1.2. Arhitectura aplicației web

Angular este o platformă care permite construirea aplicațiilor web în HTML și TypeScript[15] (un superset al limbajului Javascript). Acesta este, însuși, scris în Typescript și implementează un set de funcționalități importabile prin intermediul unor librării asemănătoare unor plugin-uri.

Principalele unități funcționale ale unei aplicații Angular sunt modulele acesteia, denumite NgModules, care pun la dispoziția aplicației un context pentru compilarea componentelor. Acestea colectează codul potrivit contextului declarat în seturi funcționale, cu scopul de a fi utilizat în interiorul aplicației, aplicația în sine fiind un set de module. O aplicație Angular trebuie să conțină cel puțin un modul rădăcină, cu scopul de a face bootstrap aplicației, alături de care pot fi declarate module care personalizează ,în mai mult detaliu, comportamentul aplicației.

[15] https://www.typescriptlang.org/docs/

Alături de module, o aplicație Angular conține și componente, fiind nevoie și declararea unei componente rădăcină, asemenea modulelor. Acestea au diverse roluri în aplicație, putând determina un proces mai granular de control al acesteia, putând defini șabloane pe care Angular să le afișeze utilizatorilor, modificând conținutul acestora în funcție de logica aplicației. Totodată, componentele pot utiliza servicii, acestea având rolul de a furniza anumite funcționalități injectabile în logica aplicației. Serviciile pot fi declarate la nivelul unei componente ca și dependințe, rezultând într-un cod modular, reutilizabil și eficient. Atât componentele, cât și serviciile, sunt simple clase, având proprii descriptori care descriu tipul clasei și informează platforma despre modul de utilizare a acestora.

Metadata generata de descriptorul unei componente asociază acea componentă cu șablonul declarat de aceasta. Astfel, se poate descrie atât partea vizuală a unei pagini, cât și comportamentul acesteia, la nivelul unei componente. Șablonul combină specificația HTML cu directive și legături specifice Angular-ului, care oferă posibilitatea modificării dinamice a codului HTML. Metadata unei clase de tip serviciu furnizează informația necesara platformei pentru a face disponibilă funcționalitatea sa la nivelul componentelor prin intermediul sistemului de injectare a dependințelor.

Componentele unei aplicații Angular pot defini vederi și le pot ierarhiza. În urma definirii și ierarhizării vederilor, se poate folosi serviciul de rutare al platformei cu scopul de a defini căi de navigare între vederi. La un nivel înalt, arhitectura unei aplicații Angular poate fi observată în diagrama următoare [7], toate entitățile prezentate în diagramă fiind utilizate intens în cadrul sistemului propus:

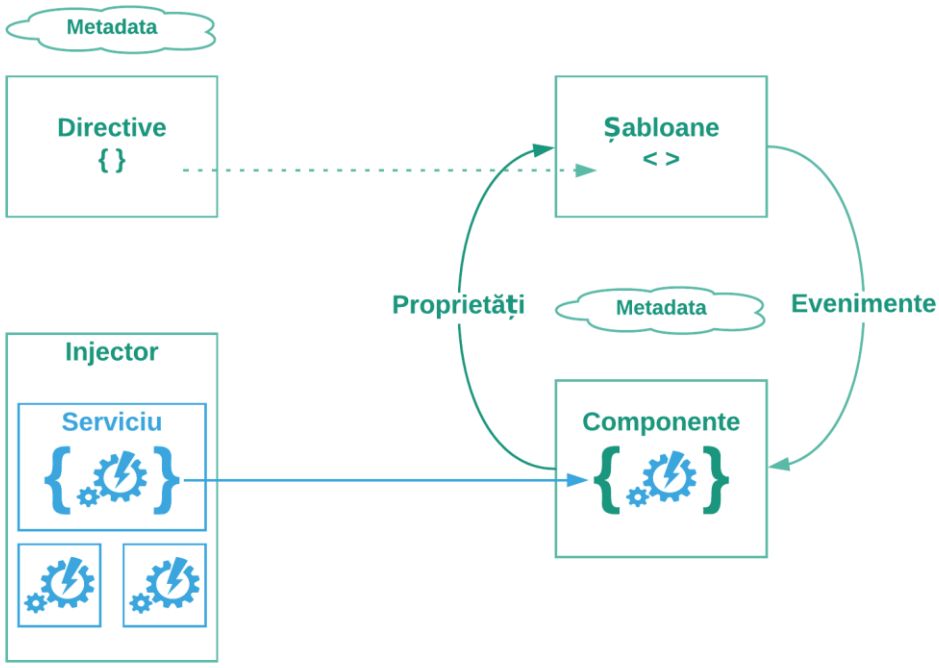

Figura 5.2 – Conceptualizarea arhitecturii Angular în Centrum

După cum se poate observa, o componentă și un șablon definesc o vedere. Decoratorul clasei care declară componenta se ocupă cu generarea metadatelor și a pointerului către șablonul asociat componentei. Directivele și legăturile bazate pe

proprietăți și evenimente modifică vederea în funcție de logica aplicației și de datele procesate de aceasta.

Totodată, injectorul de dependințe se ocupă cu servirea serviciilor către componente. Un exemplu de serviciu injectat este chiar serviciul care se ocupă de rutarea traficului în interiorul aplicației.

5.1.3. Arhitectura agregatorului

Arhitectura agregatorului respectă modelul arhitecturii pe trei nivele. Acest model arhitectural este specific aplicațiilor care servesc traficul în mod server și se caracterizează prin prezența următoarelor nivele:

- Nivelul de prezentare (Web module în diagrama sistemului)
- Nivelul Logic
- Nivelul de persistență a datelor

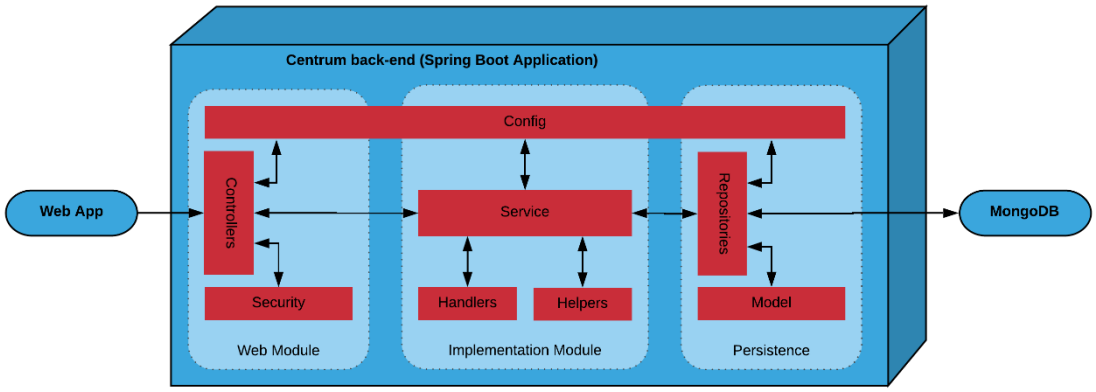

Figura 5.3 – Arhitectura componentei de agregare

Avantajul principal al acestui stil arhitectural este posibilitatea schimbării logicii oricărui nivel fără a afecta restul nivelelor. Astfel, schimbarea bazei de date și a logicii de stocare și accesare a datelor nu va necesita schimbări al nivelului de prezentare și al nivelului logic.

Un dezavantaj al acestui model este creșterea semnificativă a complexității de urmărire a fluxului de date prin aplicație. O soluție a acestei probleme este separarea nivelelor în unități funcționale independente.

Un alt dezavantaj este reprezentat de complexitatea de configurare a sistemului. Din fericire, acest dezavantaj este nulificat prin distribuirea componentei de configurare a aplicației la nivelul întregului sistem, acțiune posibilă datorită framework-ului Spring.

5.1.3.1. Modulul Web

Acesta este nivelul cel mai apropiat de client, având scopul asigurării disponibilității datelor pentru a fi accesate, precum și direcționarea acțiunilor efectuate la nivelul aplicației către serviciile din nivelul logic corespunzătoare acelor acțiuni.

Pentru implementarea interfeței programabile a aplicației s-a folosit librăria Spring MVC, parte a pachetului de librării Spring. Cu ajutorul acesteia s-au definit controlere pentru endpoint-urile prin care aplicația web interacționează cu componenta de agregare, expunând funcționalitatea acesteia prin intermediul metodelor HTTP și a stilului

arhitectural REST. În continuare va fi prezentată diagrama arhitecturală a modulului nivelului de prezentare.

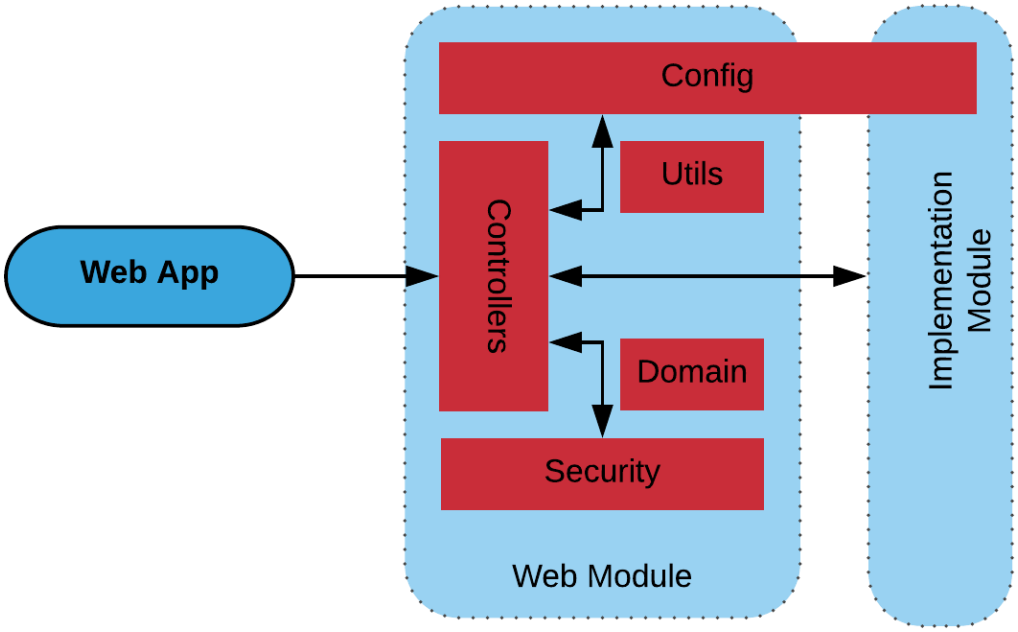

Figura 5.4 – Arhitectura nivelului de prezentare

După cum se poate observa, punctul de intrare în aplicație este reprezentat de controlerele prezente în modulul Web al aplicației. Aceste controlere sunt strâns legate de configurarea aplicației, fiind configurate folosind declarări conținute în clasele de configurare ale aplicației, reprezentate în diagrama de mai sus ca entitatea ”Config”. Totodată, punctul de intrare în aplicație este cuplat puternic și de elementele de securitate ale aplicației. Acestea sunt reprezentate ca fiind entitatea ”Security” și se ocupă cu administrarea traficului către aplicație, din punct de vedere al autentificării și autorizării. Entitățile descrise în diagrama arhitecturală sunt reprezentate de pachetele cu același nume din codul Java al aplicației.

De asemenea, diagrama de mai sus evidențiază apariția claselor de configurare pe mai multe nivele, acestea fiind prezente atât în modulul Web, cât și în modulul de implementare.

Totuși, este evidențiată și o separare pe nivele business, modulul Web fiind responsabil doar cu furnizarea accesului la aplicație. Astfel, procesarea cererilor care au ca punct de intrare modulul Web sunt trimise mai departe nivelului logic al sistemului propus.

5.1.3.2. Modulul Implementation

Nivelul logic reprezintă creierul aplicației. Acesta se ocupă de procesarea intensă a datelor și acțiunilor direcționate către acesta de nivelul de prezentare. În cazul în care este necesar accesul datelor aplicației, acest nivel se va conecta la nivelul de persistență.

Acest modul al aplicației realizează sarcini complexe, fiind necesară o vedere arhitecturală pentru evidențierea fluxului de date prin pachetele nivelului.

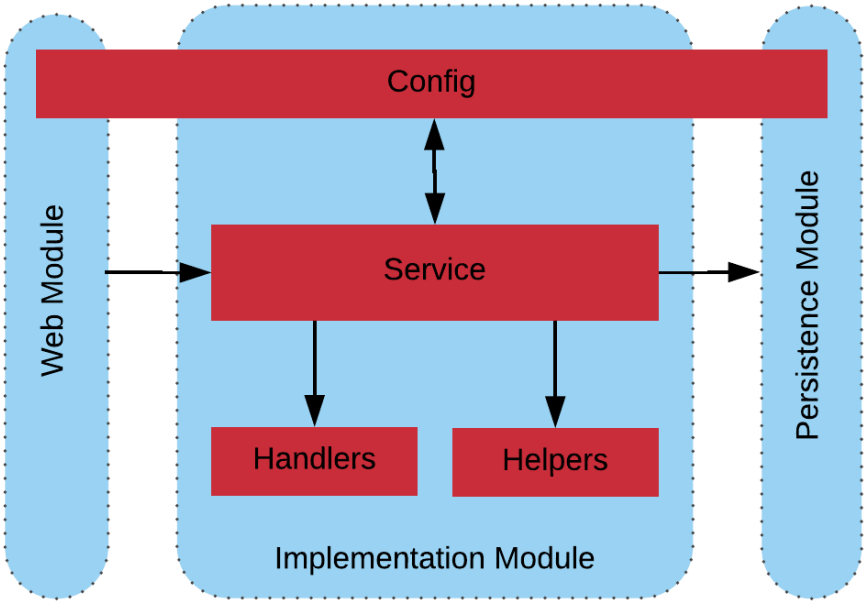

Figura 5.5 – Arhitectura nivelului logic

Diagrama de mai sus evidențiază cuplarea acestui nivel de nivelele adiacente lui, acesta acționând asemenea unui broker între API și persistență. Totodată, se poate observa apariția componentei ”Config”, aceasta realizând, prin intermediul caracteristicilor specifice pachetului de librării Spring, interconectarea și configurarea tuturor elementelor prezente în aplicației.

De asemenea, se pot observa faptul că entitatea ”Service” este principala unitate de procesare a nivelului, aceasta delegând sarcini specifice entităților auxiliare ”Handlers” și ”Helpers”. Diferența dintre cele două entități auxiliare constă în tipul de acțiuni pe care acestea le duc la bun sfârșit. Astfel, entitatea ”Handlers” are rolul de a trata cererile efectuate către serverele furnizorilor de servicii de poștă electronică, iar entitatea ”Helpers” are rolul de a ”ajuta” la buna completare a cererilor prin oferirea diverselor utilități pentru transformarea răspunsurilor primite de la serverele amintite anterior și prin gestionarea fluxurilor de date auxiliare principalului caz de utilizare al aplicației.

5.1.3.3. Modulul Persistence

Acesta este stratul cel mai de jos al aplicației, fiind responsabil cu încapsularea mecanismelor de stocare a datelor și cu expunerea datelor către nivelul superior acestuia. Acest strat trebuie să expună nivelului logic interfețe programabile conținând metode de acces și management al datelor stocate, fără a divulga acestuia implementarea din spatele acelor metode. Această ascundere a implementării și a mecanismelor de stocare este necesară pentru a nu crea dependințe între nivelele superioare și sistemul de management al datelor. Pentru o mai bună vizualizare a abstractizării din punct de vedere arhitectural a bazei de date a fost creată diagrama următoare:

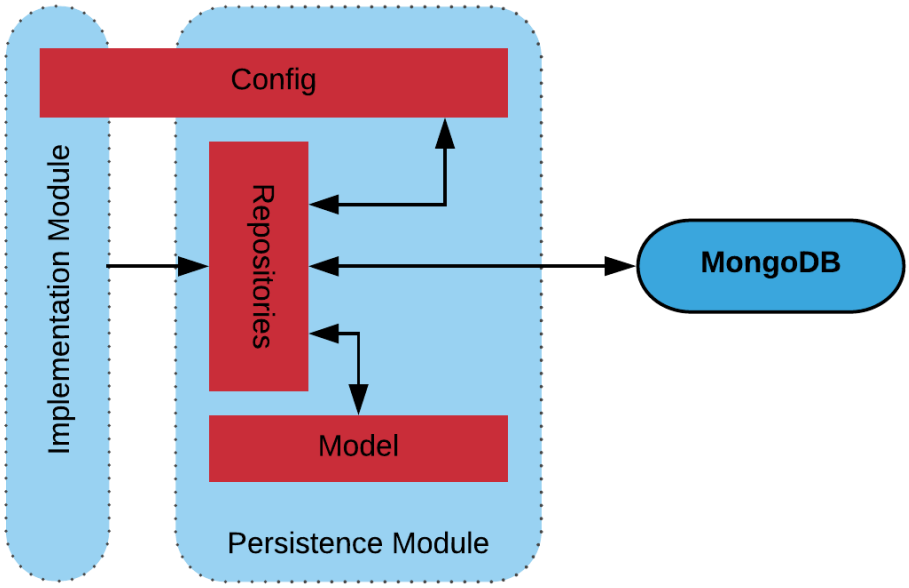

Figura 5.6 – Arhitectura nivelului de persistență a datelor

În figura aceasta se poate observa decuplarea bazei de date de restul modulelor, modului de persistență a datelor fiind singurul punct de acces al aplicației către baza de date. Astfel, nivelul superior apelează entitatea ”Repositories” a acestui modul, ea având în compoziția sa repertorii prin intermediul cărora se pot trimite operații către baza de date.

De asemenea, se poate observa prezența entității ”Model”, aceasta fiind strâns cuplată de entitatea amintită anterior. Acest lucru este necesar deoarece repertoriile amintite anterior au nevoie de un domeniu definit pentru a face translația dintre obiectele procesate la nivelul aplicației și documentele corespunzătoare acestora.

Totodată, se repetă apariția entității ”Config”, aceasta având același rol ca și în modulele anterioare. Trebuie însă precizat că, entitatea de configurare a aplicației înglobează atât clasele de configurare și fișierele de proprietăți prezente în aplicație, cât și componente de injecție și inversare a dependințelor, disponibile prin intermediul suitei de librării Spring.

5.2. Componente cheie

Pentru o mai bună înțelegere a modului de funcționare a aplicație, secțiunea aceasta va intra în detalii în ceea ce privește modul de funcționare al fiecărei componentă majoră al sistemului propus.

5.2.1. Aplicația web

Vom începe prin a descrie în detaliu aplicația web a sistemului. Aceasta a fost dezvoltat folosind tehnologii de ultimă generație precum Angular 6, RxJS și Typescript. Structura unui proiect Angular este diferită față de un proiect scris într-un limbaj orientat pe obiect clasic, deși Typescript este de asemenea orientat pe obiect. Această diferență este dată de modul de organizare a unui proiect Angular, acesta fiind centrat pe conceptul de

componente. Aceste componente reprezintă baza aplicației, fiind folosite pentru construirea modulară a paginilor web.

Totodată, Angular suportă declararea serviciilor, acestea reprezentând o grupare de funcții utilizabile din cadrul diverselor componente. Se poate face o analogie între serviciile Angular și serviciile din nivelul business al unei aplicații Java precum componenta de agregare a sistemului propus.

Pentru a evidenția mai ușor caracteristicile de mai sus, va fi utilizată următoarea diagramă.

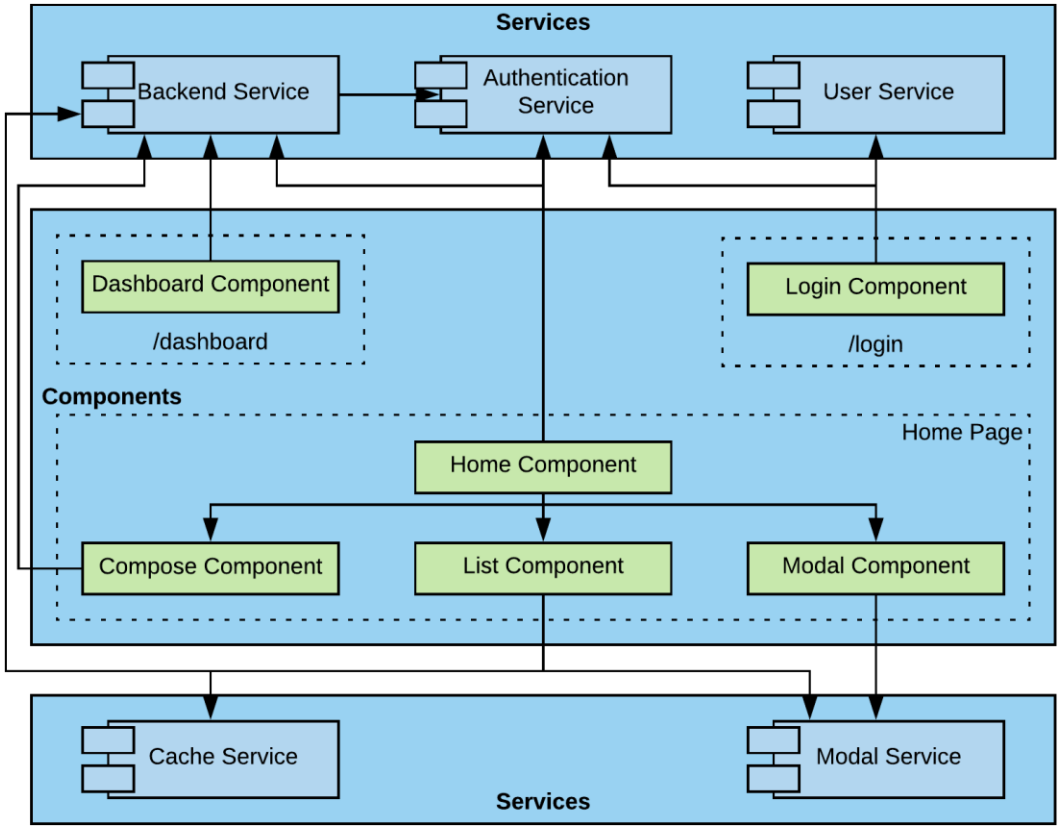

Figura 5.7 – Diagrama componentelor aplicației web

După cum se poate observa, Angular utilizează componentele pentru a construi pagini web. O componentă poate reprezenta orice, de la o căsuță de căutare, până la un panou de control. În cazul aplicației descrise în această lucrare, pagina web a panoului de control și cea a autentificării utilizatorului sunt realizate individual de componentele **Dashboard Component**, respectiv **Login Component**.

Pentru a simplifica accesul la elementul agregator, s-au creat serviciile **Backend Service**, **Cache Service**, **Authentication Service** și **User Service**. Acestea au rolul de a trimite cereri către componenta de agregare și de a efectual eventualele procesări ale datelor primite. Totodată, în completarea serviciilor descrise anterior, a fost dezvoltat serviciul **Modal Service**, acesta având rolul de a administra componenta Modal și de a asigura buna funcționare a acesteia în cadrul paginii principale a aplicației. În continuare, vom detalia rolul serviciilor ce comunică cu agregatorul.

- **Authentication Service** furnizează servicii de determinare a stării utilizatorului din punct de vedere al autentificării, având în compoziția sa metode construite cu acest scop, precum **login()**, **isLoggedIn()**, **logout()** și **getToken()**.
- **Backend Service** implementează mecanisme de cerere și procesare a utilizatorilor sistemului și a emailurilor acestora. Acest serviciu este crucial sistemului, fiind utilizat atât în pagina principală a acestuia, cât și în componenta Dashboard, cu scopul precizat anterior.
- **Cache Service** are rolul de a permite componentei List să folosească un mecanism de caching al emailurilor, cu scopul de a reduce timpul inițial de încărcare a paginii principale.
- **User Service** asigură componentei Login posibilitatea interogării bazei de date în ceea ce privește utilizatorii aplicației, prin intermediul componentei de agregare, fiind capabilă de a realiza diverse operații precum crearea, ștergerea și actualizarea utilizatorilor.

Revenind la componentele aplicației, putem observa modul în care pagina principala este construită prin intermediul componentei Home și a subcomponentelor acesteia.

Componenta Home are rolul principal de agregare a serviciilor și a componentelor utilizate în pagina principală a aplicației. Principala subcomponentă a paginii este reprezentată de List, aceasta având rolul afișării listei de emailuri agregate, dând utilizatorului posibilitatea vizualizării și interacțiunii cu emailurile respective.

Completând funcționalitatea subcomponentei anterioare, Modal Component ajută la afișarea detaliată a emailurilor și facilitează compunerea răspunsurilor prin afișarea unor modale în momentul anumitor acțiuni precum selectarea unui email sau apăsarea butonului ”Reply”.

Ultima subcomponentă a paginii principale este reprezentată de Compose. Aceasta are rolul de a furniza utilizatorului un mod de a compune emailuri noi și de a le trimite către o listă de destinatari. Această componentă este utilizată și de componenta Modal, aceasta afișând utilizatorului componenta Compose în cazul în care acesta alege să compună un răspuns sau un forward pentru unul din emailurile primite de acesta.

Totodată, aplicația web utilizează diferite module din cadrul suitei Angular, precum:

- **Router** pentru redirecționarea traficului prin aplicație.
- **HttpModule** pentru realizarea apelurilor HTTP.
- **BrowserAnimationsModule** pentru animarea diverselor componente ale interfeței utilizator.
- **ViewEncapsulation** pentru a afișa, în mod securizat, emailurile în format HTML.

De asemenea, pentru afișare corectă a emailurilor a fost necesar un efort substanțial, întrucât emailurile HTML pot conține cod care să fie executat, expunându-l sistemul unor atacuri. Pentru a elimina acest risc, se poate folosi ”igienizarea” codului HTML al emailurilor, însă această opțiune va elimina din corpul emailurilor elemente precum imaginile, animațiile și fonturile personalizate. În urma unei cercetări amănunțite, a fost implementat un mecanism de izolare a emailurilor prin intermediul unei abilități de încapsulare a unui subarbore de elemente DOM, numită **Shadow DOM**. Astfel, codul din interiorul unui email este executat în interiorul acelui subarbore, fiind capabil de a îl modifica, însă acest lucru nu reprezintă o problemă, întrucât subarborele respectiv este

izolat și nu poate afecta alte componente ale sistemului. Prin urmare, în cazul unui atac, codul se va executa doar în cadrul elementelor ce compun emailul care conține codul atacator, rezultând în blocarea atacului.

5.2.2. Componenta de agregare

În continuare, va fi descrisă componenta de agregare a sistemului propus. Aceasta este reprezentată de o aplicație Java organizată în trei nivele, acestea urmând să fie detaliate în cele ce urmează, începând cu punctul de intrare în aplicație, reprezentat de componenta responsabilă cu expunerea punctelor de conectare către aplicația web descrisă la punctul anterior. Detalierea va fi facilitată de includerea diagramelor și exemplificarea pe baza acestora a modului de organizare internă și funcționare a modulelor. Pentru început, va fi detaliat pachetul controller, parte din modulul web al aplicației, acesta găzduind controlere ce administrează punctele de intrare în aplicație.

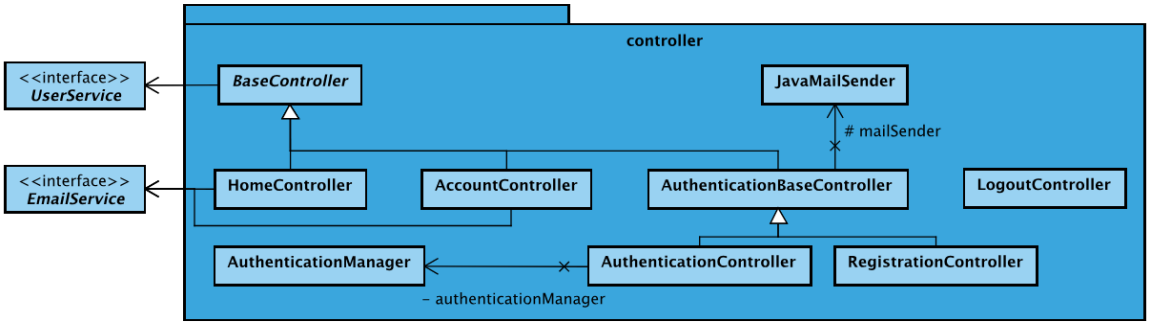

Figura 5.8 – Diagrama de clase a pachetului controller

Pentru o mai bună organizare a controlerelor aplicației, acestea au fost organizate în funcție de scopul deservit, comportamentul comun fiind extras într-un controler numit **BaseController**. Acesta conține o instanță a serviciului de administrare a utilizatorilor, numit **UserService**, și o metodă ce are scopul de a returna utilizatorul autentificat în sistem. Această metodă este foarte utilă, întrucât majoritatea operațiilor efectuate de sistemul propus au nevoie de cunoașterea utilizatorului în contextul căruia este efectuată operația. Unica entitate ce nu extinde clasa **BaseController** este **LogoutController**. Acesta este separat de restul entităților, întrucât nu este suficient de complex pentru a merita o extindere a celorlalte clase.

Controlerele aplicației au fost separate în funcție de aria acoperită de acestea, fiind numite într-un mod sugestiv în ceea ce privește funcționalitatea oferită. De la această regulă se exclud **BaseController** și **AuthenticationBaseController**, acestea fiind clase părinte responsabile cu extragerea comportamentul comun al claselor derivate. Astfel, pachetul controller, conține următoarele controlere:

- **HomeController**, responsabil cu administrarea operațiilor ce pot fi efectuate de pe pagina principală a aplicației, operații precum interogarea emailurilor și a folderelor unui utilizator.
- **AccountController**, suportând operațiuni de configurare a unui cont, precum adăugarea unui cont nou de email.
- **RegistrationController**, realizator al operațiilor de înregistrare a unui utilizator nou.

- **AuthenticationController**, responsabil cu procesarea cererilor de autentificare ale utilizatorilor.
- **LogoutController**, procesor al operației de ieșire din sistem.

Modulul web al aplicației conține, după cum se poate observa în diagramele arhitecturale de mai sus, alte pachete suplimentare pachetului controller. Alte pachete importante din modulul web sunt **Config**, responsabil cu configurarea aplicației utilizând Spring Framework, și **Security**, implementând furnizorul de servicii de autentificare Spring împreună cu clasele anexe acestuia.

Continuând, modulul implementation al aplicației este unul complex, având rolul de a procesa orice fel de interogare efectuată asupra aplicației. De asemenea, acest modul este și unitatea de legătură dintre componenta web a aplicației și baza de date. Pentru a evidenția mai bine aceste caracteristici principale ale modulului, vom continua prin descrierea detaliată a pachetului **service**.

Pachetul service conține principalele servicii ale aplicației și se poate spune că este nucleul modulului de implementare al sistemului propus. Astfel, întreaga logică a aplicației se procesează în aceste entități, anumite sarcini fiind delegate de către acestea către servicii specifice acelui tip de procesare. De exemplu, clasa **EmailServiceImpl** deleagă toate acțiunile de descărcare sau trimitere a emailurilor către clasele corespunzătoare din cadrul pachetului **handler**.

Unul dintre serviciile principale ale aplicației este **UserService**. Implementarea acestei interfețe determină modul de procesare al acțiunilor ce administrează utilizatorii aplicației. Acest serviciu se află în strânsă legătură cu clasele **AccountController**, **RegistrationController** și **AuthenticationController**, realizând toate operațiile specifice acestor controlere.

Cel de-al doilea serviciu crucial bunei funcționări a aplicației este **EmailService**. Acesta procesează cererile specifice controlerului **HomeController**, fiind astfel un executor al acțiunilor efectuate de utilizator în pagina principală a aplicației. Acest serviciu este unul foarte complex, orchestrând acțiunile de descărcare, procesare și trimitere a emailurilor, acțiuni posibile prin intermediul claselor declarate la nivelului pachetului **handler**. În continuare, va fi descris acest pachet, evidențiind nevoile îndeplinite de acesta în cadrul sistemului propus.

Acest pachet poate fi împărțit în două fluxuri de date, unul pentru descărcarea emailurilor, iar cel de-al doilea pentru trimiterea emailurilor. Aceste fluxuri sunt reprezentate de componentele **RetrievingRequestHandler** și, respectiv, **SendingRequestHandler**.

De asemenea, pentru a facilita procesul de autentificare și pentru a asigura generalitatea componentelor de mai sus, au fost realizate clasele următoare:

- **PropertiesHandler**, clasă responsabilă cu încărcarea dinamica a diverselor proprietăți specifice unui server de email, permițând astfel utilizarea simultană a adreselor de email din mai multe surse.
- **SmtpAuthenticator**, entitate ce extinde clasa **Authenticator** a pachetului javax.mail cu scopul de a configura autentificarea în funcție de proprietățile încărcate dinamic de clasa descrisă anterior.

Aceste caracteristici pot fi observate în detaliu în diagrama de clase de mai jos, putând fi, totodată, observată și prezența pachetului **helper**. Astfel, vom continua prin a

descrie acest pachet, evidențiind impactul pe care acesta îl are la nivelul sistemului propus și, de asemenea, rolul jucat de acesta în cadrul serviciului **EmailService**.

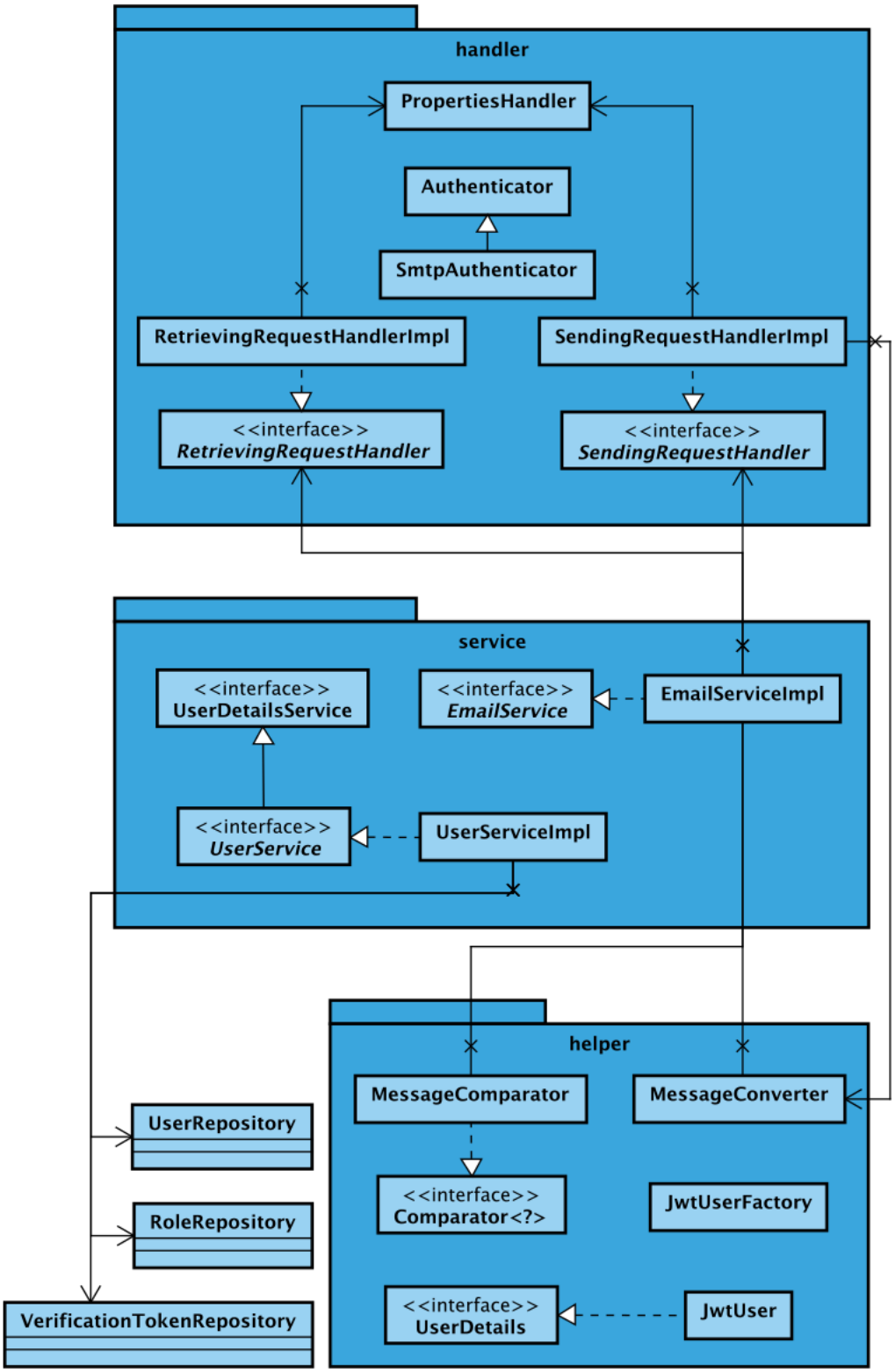

Figura 5.9 – Diagrama de clase a modulului implementation

Pachetul **helper** este compus din două clase principale, **MessageComparator** și **MessageConverter**. Cea dintâi are rolul de a compara două mesaje email după data la care acestea au fost primite, implementând interfața Comparator din pachetul java.util. Cea din urmă joacă un rol foarte important în descărcarea emailurilor, realizând conversia acestora din formatul **Message** oferit de javax.mail într-un format mult mai ușor de utilizat la nivelul agregatorului și al aplicației web.

Întrucât serverele de email opresc uneori conexiunile la acestea după un anumit timp, procesul de conversie al mesajelor include și un mecanism de reconectare la serverele furnizorilor de servicii de poștă electronică. Acest mecanism este declanșat de excepția **FolderClosedException** definită în pachetul javax.mail.

Ultimul modul al aplicației este modulul persistence, responsabil cu realizarea comunicării cu baza de date. În continuare, vom detalia rolul acestui ultim modul, evidențiind detaliile de implementare prin intermediul diagramei de mai jos.

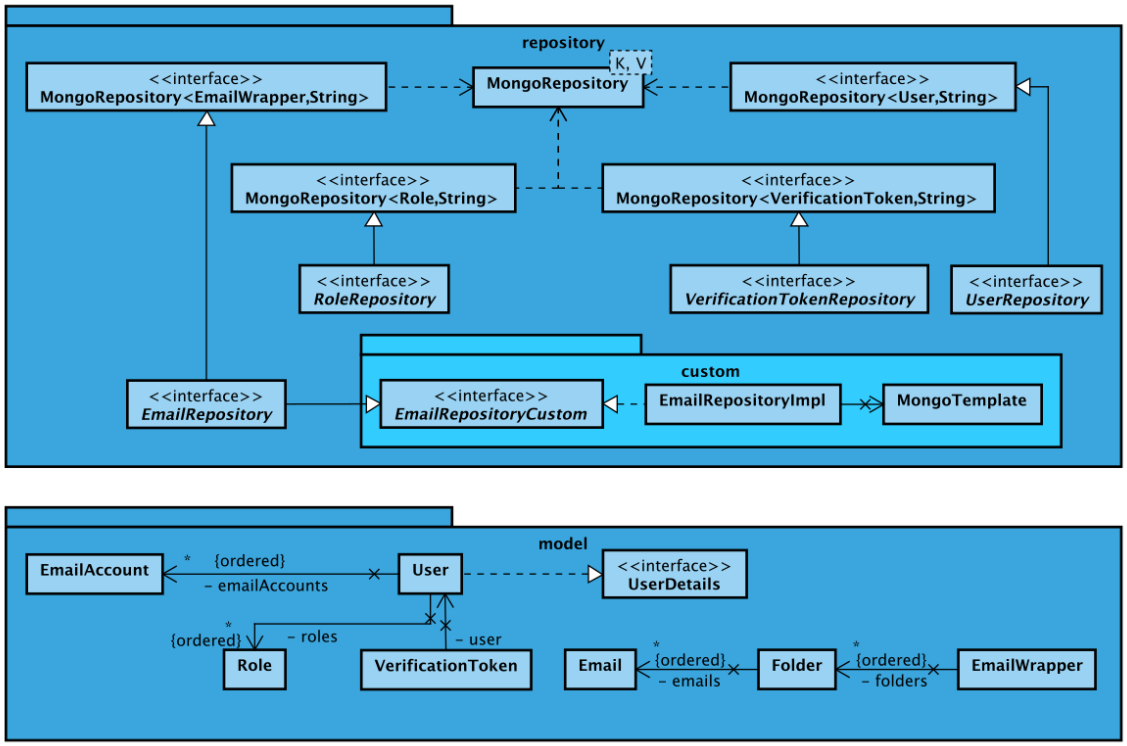

Figura 5.10 – Diagrama de clase a modulului persistence

După cum se poate observa, acest modul conține clase concrete ale obiectelor de tip DAO, numite **repository** în contextul librăriei Spring Data și fiind organizate într-un pachet cu același nume. Aceste clase sunt create utilizând interfețe ale librăriei Spring Data specifice bazei de date MongoDB, acestea furnizând o abstractizare a metodelor de acces și interogare a sistemului de stocare a datelor.

Cu toate acestea, a fost necesară crearea unor repertorii personalizate pentru a realiza stocarea emailurilor utilizatorilor. Aceste elemente personalizate au fost realizate creând o interfață și o implementare a acesteia, urmând ca interfața să fie extinsă, alături de interfața standard **MongoRepository**, la nivelul interfeței **EmailRepository**. Astfel, mecanismul de inversiune a dependințelor din cadrul suitei de librării Spring va injecta clasa concretă personalizată ca și implementare a interfeței **EmailRepository**, respectiv a interfeței

standard **MongoRepository**, făcând posibilă apelarea metodelor descrise în clasa personalizată.

Totodată, modulul persistence conține și clasele concrete ale obiectelor de tip DTO, acestea fiind organizate în pachetul **model**. Astfel, în diagrama de mai sus se poate observa modul de încapsulare a emailurilor, oferind un mod de separarea a acestora în funcție de contul de email din cadrul căruia acestea au fost descărcate. De asemenea, este evidențiată și implementarea de către clasa **User** a interfeței **UserDetails** furnizată de librăria Spring Security. Această implementare permite utilizarea obiectelor de tip **User** pentru autentificarea și autorizarea unui utilizator, autorizarea fiind determinată, la rândul ei, de încapsularea clasei **Role**, aceasta descriind rolul utilizatorului în sistem din punctul de vedere al autorizării.

5.2.3. Componenta de administrare a sistemului

Componente de administrare a sistemului este reprezentate de aplicația Rancher. Această componentă permite unui administrator să efectueze schimbări complexe ale arhitecturii sistemului, precum:

- Adăugarea și configurarea proxy-urilor
- Adăugarea și scalarea aplicațiilor instalate
- Distribuirea aplicațiilor pe multiple noduri
- Distribuirea aplicațiilor pe diferite noduri în funcție de afinitatea acestora
- Pornirea rapidă a unui servicii prin intermediul unui catalog personalizabil

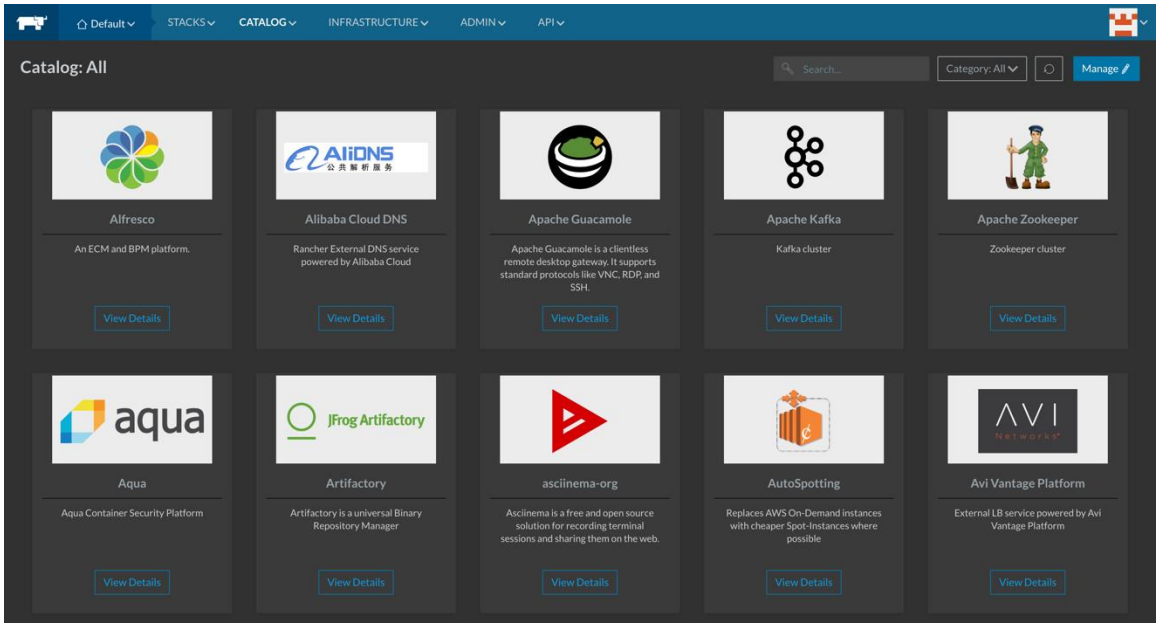

Figura 5.11 – Catalogul personalizabil al utilitarului Rancher

Principala caracteristică a acestei componente este reprezentată de catalogul personalizabil. În cazul sistemului propus, acesta a fost utilizat pentru a crea un model de instalare a aplicației utilizând fișiere de configurare **yml**. Astfel, instalarea aplicației Centrum se poate realiza prin simpla selectare a acesteia din intermediul catalogului și

apăsarea butonului ”Launch” (în cazul în care se dorește lansarea aplicației cu configurările implicite), după cum se poate observa în imaginea următoare:

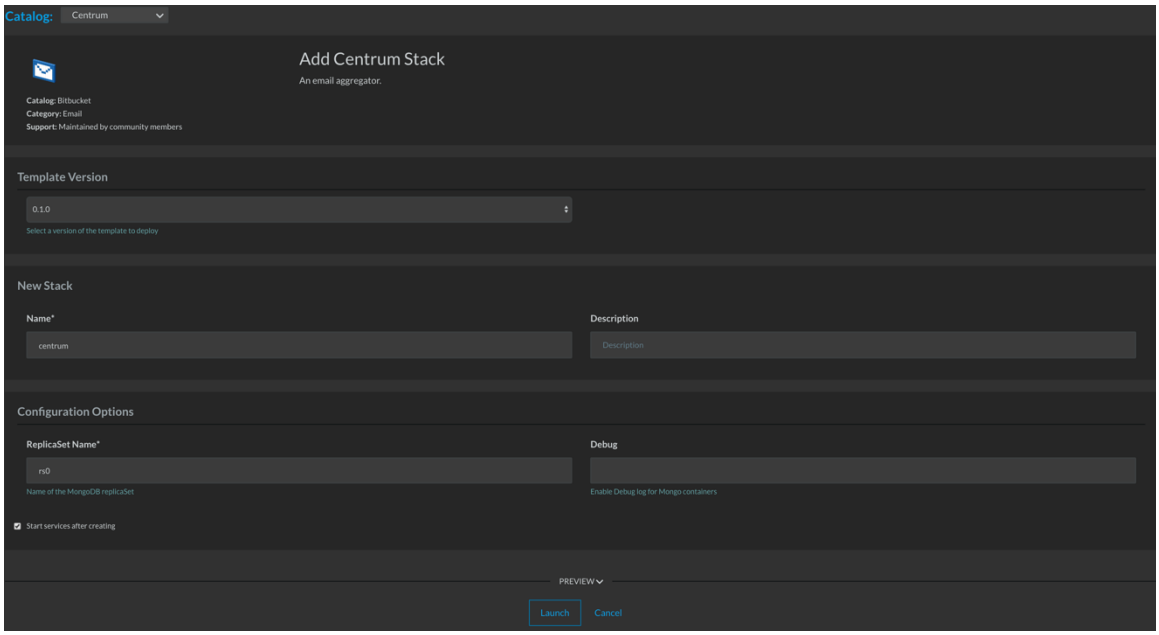

Figura 5.12 – Instalarea sistemului propus prin intermediul utilitarului Rancher

De asemenea, Rancher oferă un sistem de scalare a aplicație și un mecanism de asigurare a redundanței prin repornirea serviciilor care fost închise din cauza unei erori. Totodată, acesta suportă și opțiuni de actualizare a serviciilor lansate fără downtime, o caracteristică crucială pentru sistemele aflate în producție.

5.2.4. Componenta de monitorizare a sistemului

Asigurarea monitorizării unui sistem este un obiectiv crucial în implementarea acestuia. O bună monitorizare a unei aplicații este crucială în întreținerea sa, iar un sistem de alertare bine implementat ajută semnificativ la administrarea acesteia într-un mediu de producție.

O alertă reprezintă un semnal activat în cazul descoperirii unui comportament problematic la nivelul unei aplicații, rolul acesteia fiind aducerea anomaliei detectate în atenția administratorilor sistemului. Însă, pentru a putea fi capabili de detectarea unor anomali, trebuie definit normalul.

Pentru a defini atât normalul, cât și comportamentul defectuos al unui sistem software, este nevoie de metrici. Metricile unei aplicații reprezintă măsurători ale comportamentului aplicației. Acestea pot fi create din diverse surse, precum jurnalele aplicației sau jurnalele sistemului de operare.

Pentru agregarea, colectarea și transformarea acestor metrici, Centrum se folosește de un utilitar gratuit de procesare a metricilor, numit Prometheus. Colectarea datelor se face prin intermediul unor exportatori instalați pe fiecare nod al aplicației, datele colectate incluzând metrici precum nivelul de utilizare al procesorului, nivelul de utilizarea al memorie RAM și al spațiului de stocare, numărul de instrucțiuni de citire de pe disc, lungimea de bandă folosită pentru conexiunea la internet, etc.

Prometheus este un utilitar foarte flexibil, însă nu este dezvoltat cu scopul de a fi o unealtă de vizualizare a metricilor. Utilitarul oferă atât un mod de vizualizare a metricilor, cât și un sistem de alertare a administratorilor, dar acestea dificil de configurat și oferă un set restrâns de caracteristici. După cum se poate observa în figura următoare, vizualizarea datelor este net inferioară unor vizualizatoare specializate.

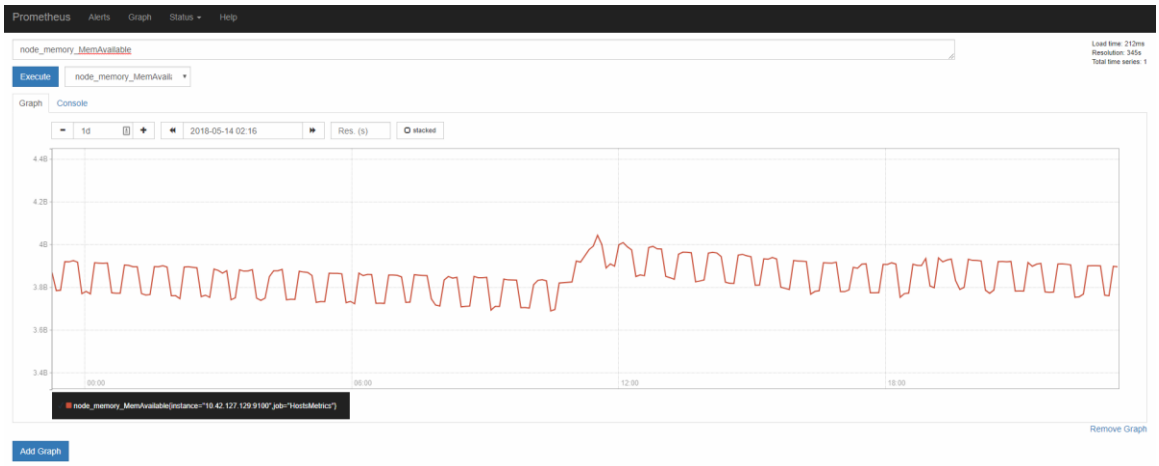

Figura 5.13 - Interfața grafică a componentei de monitorizare oferită de Prometheus

Astfel, pentru vizualizarea metricilor colectate se va folosi Grafana, un vizualizator open-source. Acest utilitar este compatibil cu agregatoare de date precum Graphite și InfluxDB, însă și cu Prometheus.

Figura 5.14 - Interfața grafică a unui panou de bord in Grafana

5.3. Structura bazei de date

Sistemul prezentat în această lucrare utilizează o bază de date NoSQL open-source, bazată pe documente. Acest mod de stocare al datelor are o influență puternică în ceea ce privește structura bazei de date, fiind posibilă apariția unor probleme majore în cazul unui design defectuos. Una dintre posibilele probleme este reprezentată de stocarea recursivă a unor documente, evidențiată de diagrama următoare:

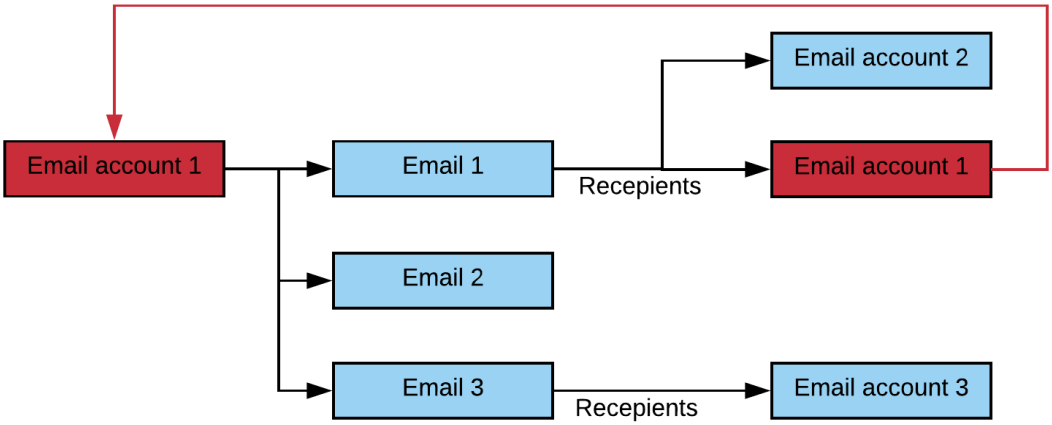

Figura 5.15 – Stocarea recursivă a documentelor

MongoDB are propriile mecanisme de corectare a acestei structuri, evitând interogările recursive, însă o structură de tipul celei de mai sus trebuie evitată, întrucât aceasta poate duce la deteriorarea performanței sistemului în cazul anumitor interogări.

Luând în considerare problema menționată anterior, precum și alte bune practici, structura bazei de date a luat o formă robustă și relativ ușor de interogat. În continuare vom detalia modul de stocare a datelor, evidențiind structura bazei de date prin intermediul diagramei relațională a entităților de mai jos.

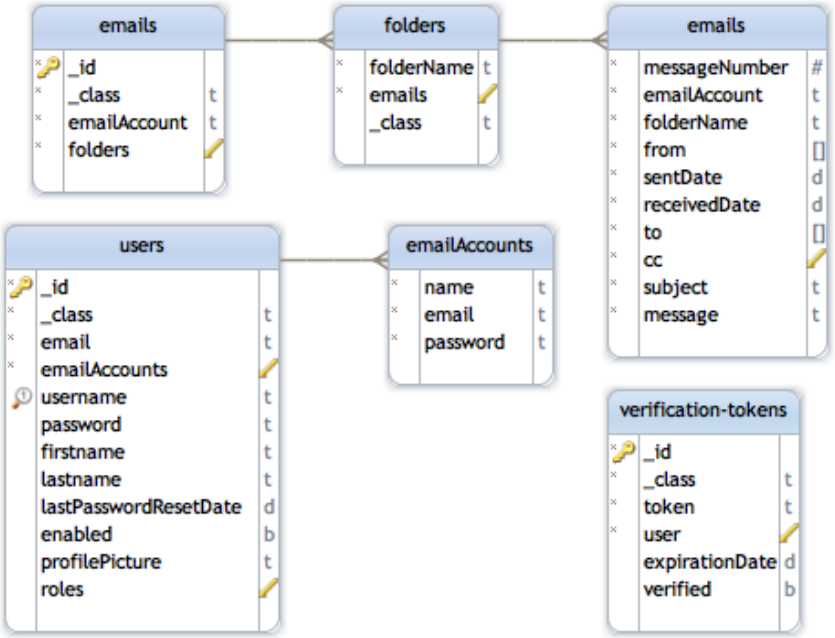

Figura 5.16 – Diagrama relațională a entităților

După cum se poate observa, entitățile descendente nu referențiază entitățile părinte, evitând astfel problema de mai sus. În continuare vom prezenta în detaliu fiecare entitate, explicând rolul său în cadrul sistemului.

Începând cu entitatea principală a aplicației, documentul **emails,** se poate observa o încapsulare a emailurilor în entități wrapper. Astfel, entitatea de bază **emails** este alcătuită din o adresă de email și o listă de foldere prezente în cadrul adresei precizate anterior, urmând ca această listă să fie formată la rândul său din numele folderului și o listă a tuturor emailurilor din folderul respectiv. Această compoziție permite izolarea emailurilor în cadrul aceluiași cont, din punct de vedere ierahic, fiind, de asemenea, posibilă clasificarea acestora în funcție de folderul din care fac parte.

Continuând, putem observa o structură compozițională și în cadrul entității **users,** fiecare utilizator având asociate, sub forma entității **emailAccounts,** conturile de email adăugate de acesta în sistem. În plus, utilizatorii sunt indentificați prin intermediul unui rol stocat sub forma unei entități constante numită **roles**.

În final, putem evidenția entitatea **verification-tokens**, aceasta fiind responsabilă cu stocarea unor identificatoare unice necesare activării conturilor noi înregistrate, un identificator fiind șir alfanumeric aleator trimis prin email și accesat de noul utilizator prin intermediul punctului de ieșire **/confirm**.

5.4. Deployment

Instalarea întregului sistem este o sarcină complexă, însă ajută enorm în simplificarea viitoarelor procese de administrare a aplicației. Pentru a asigura o instalare cât mai rapidă și mai corectă a sistemului, trebuie evidențiat modul de distribuție a componentelor pe containere, cât și tipul de aplicație server care va susține instalarea diverselor aplicații ale sistemului propus.

Începând cu modul de găzduire a componentelor sistemului, putem detalia faptul că acesta este găzduit în totalitate în containere Docker rulând pe hosturile fizice înregistrate în Rancher. Astfel, putem scoate în evidență o ierarhie a entităților ce fac parte din procesul de deployment, după cum urmează:

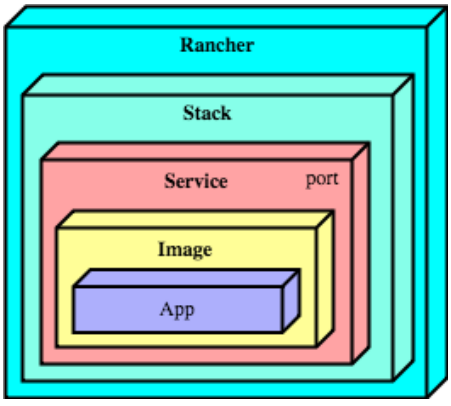

Figura 5.17 – Ierarhia entităților ce fac parte din procesul de deployment

După cum se poate observa, o aplicație rulează în cadrul unui container bazat pe o imagine Docker. Această imagine este mai apoi adăugată unui serviciu, parte a unui Stack din aplicația de administrare a infrastructurii. Rancher este la rândului lui găzduit de o mașină fizică, însă aplicațiile ce fac parte din sistem vor fi distribuite pe gazdele înregistrate în Rancher. Pentru a evidenția mai bine acest proces a fost construită următoarea diagramă:

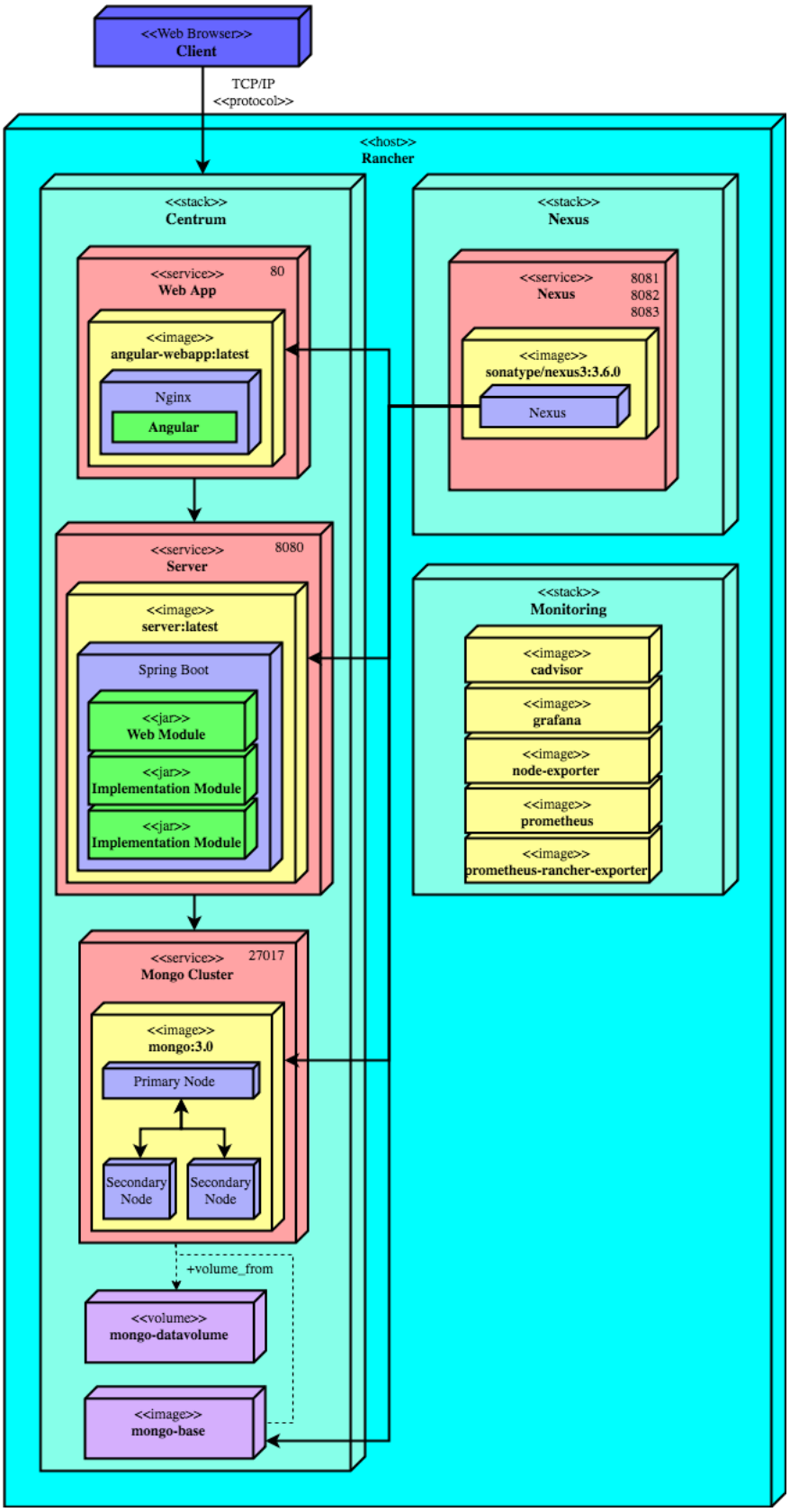

Figura 5.18 – Diagrama de deployment a sistemului

Imaginile Docker sunt stocate la nivelul repertoriului Nexus, fiind descărcate la crearea containerelor. De asemenea, se poate observa separarea dintre serviciu, imagine si application server. În cazul componentei de agregare, serviciul Server este creat în interiorul stackului Rancher, imaginea este rulată în cadrul acestui serviciu, această imagine conținând application serverul utilizat (Spring Boot în cazul descris), server care rulează o listă de fișiere jar generate pe baza codului Java.

Totodată, se evidențiază utilizarea seturilor de replicare MongoDB, cât și utilizarea volumelor Docker cu scopul persistenței datelor.

Capitolul 6. Testare şi Validare

6.1. Analiza performanței

În cadrul acestui capitol se dorește prezentarea principalelor tehnici de testare utilizate pentru validarea sistemului propus. Totuși, realizarea corectă a etapelor de testare nu oferă garanția unui sistem complet funcțional în orice condiții, acest proces fiind doar un pas către o stabilitatea mai bună a sistemului prin detectarea eventualelor neajunsuri și rezolvarea acestora.

Pentru a atinge performanța specificată în cerințele non-funcționale ale sistemului, aplicația trebuie testată de-a lungul întregului ciclu de dezvoltare pentru a putea urmării evoluția performanței sistemului în raport cu adăugarea de noi caracteristici sistemul. Astfel, s-a recurs la utilizarea unui program dezvoltat special pentru testarea rezistenței unei aplicații la un număr foarte mare de cereri pe secundă. Acest program se numește **Locust**, este scris în limbajul de programare **Python** și este complet open-source. De asemenea, fișierele de configurare ale cazurilor de testare vor fi scrise în **Python**, acesta fiind un limbaj de programare ușor de învățat și de utilizat.

Metodologia de testare este una simplă, dar eficientă. Utilizând **Locust**, au fost efectuate peste 100000 de cereri către două pagini ale aplicației, pagina principală și pagina de autentificare. Aceste interogări au fost făcute la o rată medie de 169.39 interogări/secundă, simulând un număr de 200 de utilizatori concurenți, iar rezultatele sunt evidențiate în următoarele grafice:

Figura 6.1 – Grafic al numărului de cereri efectuate pe secundă pe parcursul testării

Figura 6.2 – Grafic al timpului de răspuns mediu pe parcursul testării

Sistemul a funcționat pe întregul parcurs al perioadei de testare, fără erori, având o rată de succes a interogărilor de 100%. Pentru informații mai detaliate în ceea ce privește comportamentul sistemului, consultați tabelul următor:

Endpoint	# cereri	# erori	Timp de răspuns mediu (ms)	Timp de răspuns minim (ms)	Timp de răspuns maxim (ms)	cereri/s
/	50212	0	177	151	1177	84.89
/login	49987	0	177	152	1278	84.51
Total	100199	0	177	151	1278	169.39

Tabel 6.1 – Tabel comparativ al funcționării sistemului pe parcursul testării

În concluzie, se poate observa faptul că, în ciuda încărcării ridicate a sistemului, acesta continuă să funcționeze în parametrii normali specificați în cadrul cerințelor non-funcționale.

6.2. Optimizarea sistemului

În decursul dezvoltării sistemului au fost întâmpinate diferite piedici în atingerea obiectivelor non-funcționale ale acestuia. Una dintre aceste piedici a fost reprezentată de viteza foarte mică de descărcare a email-urilor de pe conturile Yahoo, rezultând într-o inițializare care putea dura peste 12 ore pentru 15000 de emailuri.

Această problemă a fost rezolvată în urma unor lungi sesiuni de testare si validare a diverselor soluții de optimizare a modului de descărcare al emailurilor. Ca urmare a acestor sesiuni de testare și validare, a fost posibilă crearea graficului de mai jos, acesta expunând timpii de descărcare a diverselor soluții încercate.

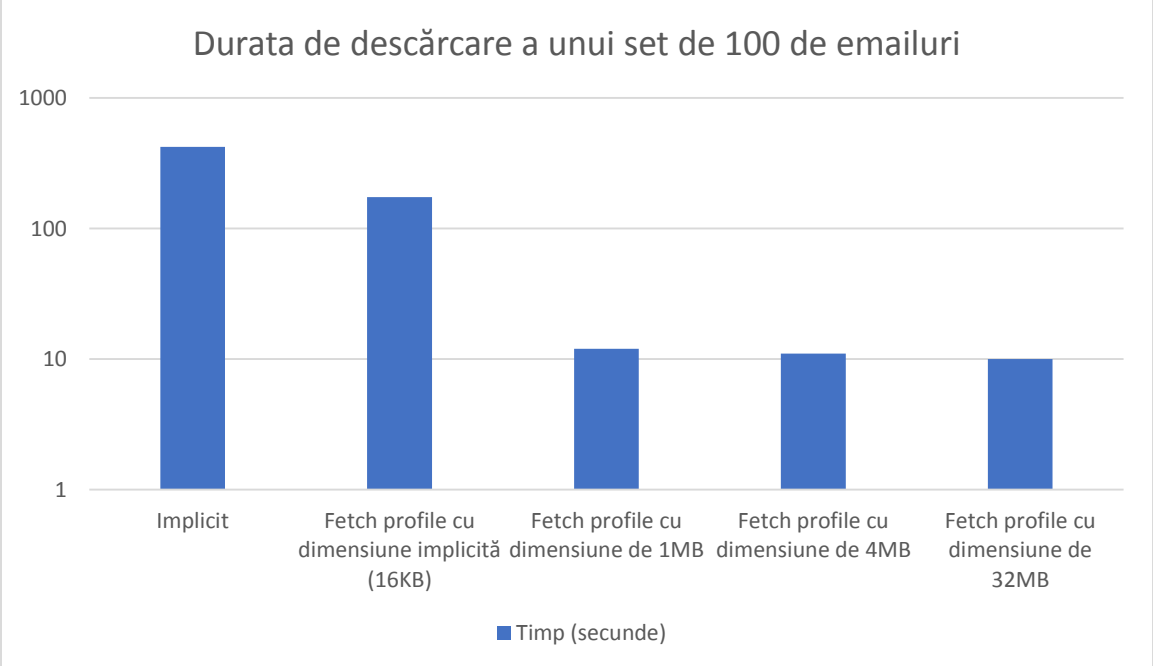

Figura 6.3 – Comparație în scală logaritmică a mecanismelor de descărcare a emailurilor

După cum se observă, cea mai eficientă soluție este ultima din tabel, reușind să descarce setul de emailuri în doar 9 secunde. Totuși, diferența este foarte mică între soluțiile ce constă în utilizarea fetchului cu dimensiuni mai mari de 1MB, cel mai mare câștig fiind în folosirea acestui mecanism personalizat în detrimentul celui implicit sau al celui fetch cu dimensiunea implicită de 16KB. Diferența între soluția implicită si soluția cea mai optimă este considerabilă, graficul de deasupra nefiind capabil să reflecte total această diferență, fiind nevoie de o scală logaritmică pentru a evidenția diferența dintre ultimele 3 categorii. Pentru a expune această diferență, vom utiliza un alt grafic, ascunzând din acesta soluțiile fetch profile cu dimensiune de 1MB sau 4MB și nefolosind o scală logaritmică de reprezentare a valorilor obținute.

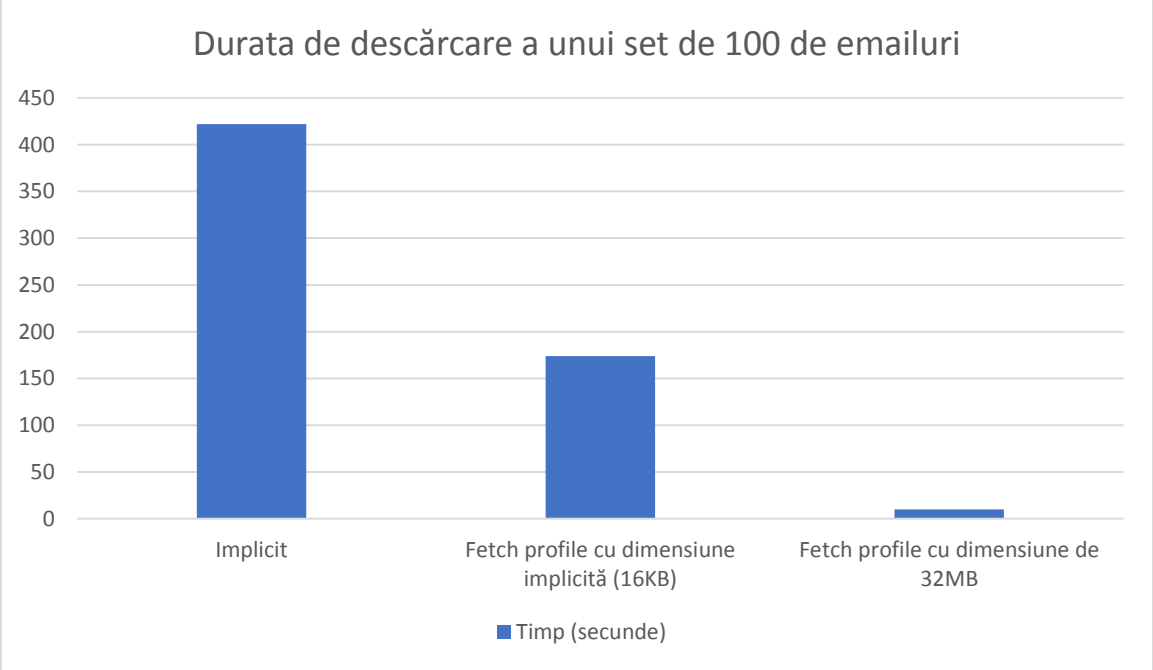

Figura 6.4 – Grafic comparativ în scală normală a principalelor mecanisme de descărcare a emailurilor

Prin urmare, se poate observa diferența majoră în viteza de descărcare, soluția finală fiind de 42 de ori mai rapidă decât soluția implicită și de 17 ori mai rapidă decât soluția fetch cu dimensiunea implicită de 16KB.

Din păcate, descărcarea emailurilor nu depinde doar de acest factor, serverele Yahoo răspunzând, în general, mai greu cererilor trimise de Centrum, comparativ cu serverele Google. Acest factor rezultă în o latență mai mare de descărcare a emailurilor găzduite de Yahoo, însă diferența nu este critică.

Capitolul 7. Manual de Instalare si Utilizare

În acest capitol sunt prezentați pașii necesari instalării și utilizării sistemului propus, împreună cu resursele necesare rulării acestuia.

7.1. Resurse necesare

Pentru a fi utilizate corespunzător, toate aplicațiile software au nevoie de anumite resurse hardware și software. În continuare, sunt definite resursele minime necesare funcționării componentelor principale ale sistemului descris, Centrum și Rancher, împreună cu câteva recomandări suplimentare.

Sistem de operare	Orice sistem de operare modern ce suportă una din următoarele versiuni de Docker: • 1.10.3 • 1.12.3 • 1.13.x • 17.03.x-ee • 17.03.x-ce Recomandate: RancherOS, Ubuntu, RHEL/CentOS 7
Memorie RAM	Minim 1GB
Procesor	Minim două nuclee sau un procesor fizic

Tabel 7.1 – Resursele necesare aplicației

Sistem de operare	Orice distribuție modernă de Linux ce suportă una din următoarele versiuni de Docker: • 1.10.3 • 1.12.3 • 1.13.x • 17.03.x-ee • 17.03.x-ce Recomandate: RancherOS, Ubuntu, RHEL/CentOS 7
Memorie RAM	Minim 1GB
Procesor	Minim două nuclee sau un procesor fizic
În cazul utilizării unei baze de date externe MySQL	MySQL max_connections > 150 Cerințe de configurare MySQL: • Rulare folosind Antelope cu opțiunea implicită ”COMPACT” • Rulare MySQL 5.7 cu Barracuda, unde opțiunea implicită ”ROW_FORMAT” este setată ca ”Dynamic”

Tabel 7.2 - Resursele necesare Rancher

7.2. Instalare

Instalarea sistemului realizat se poate efectua în mediul local, fiind însă necesară instalarea în prealabil a platformei Docker. Pentru instalarea și utilizarea platformei Docker, virtualizarea trebuie să fie activată în BIOS, iar procesorul trebuie să suporte SLAT (Second Level Address Translation). De asemenea, sistemul de operare trebuie să funcționeze pe 64 de biți, iar în cazul instalării pe sistemul de operare Windows 10 se recomandă instalarea aplicației Docker for Windows, însă aceasta necesită activarea suportului pentru Hyper-V.

7.2.1. Instalarea aplicației de agregare

Instalarea aplicației centrale a sistemului este foarte simplă. În urma instalării platformei Docker, aplicația poate fi instalată rapid și simplu prin rularea comenzii ***./gradlew redeploy*** pentru UNIX sau ***gradlew.bat redeploy*** pentru Windows în directorul ce conține agregatorul, acest proces fiind urmat de rularea comenzii ***npm start*** în cadrul directorului ce conține aplicația web. Pentru instalarea aplicației web este necesară instalarea în prealabil a platformei NodeJS.

7.2.2. Instalarea întregului sistem

Aplicația client este reprezentată de navigatorul web folosit de utilizator, nefiind necesare instalări sau configurări suplimentare, însă instalarea întregului sistemului reprezintă o sarcină inițial complexă.

În urma instalării Docker, se poate continua instalarea sistemului. Se va începe cu instalarea componentei de administrare a sistemului, Rancher. Instalarea acesteia este simplă, fiind necesară doar rularea în terminal a următoarei comenzi Docker: ”***docker run -d --restart=unless-stopped -p 8080:8080 rancher/server:v1.6.12***”. În funcție de sistemul de operare utilizat, această comandă poate cere drepturi de administrator.

În urma rulării comenzii de mai sus, Rancher va fi instalat și va putea fi accesat pe portul local 8080. Pentru administrarea și provizionarea de containere, Rancher are nevoie de cel puțin o gazdă. Aceasta poate fi adăugată din meniul Infrastructure → Hosts, urmând apoi instrucțiunile afișate.

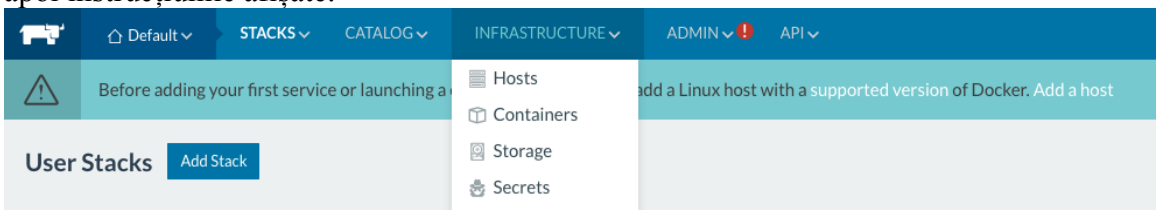

Figura 7.1 – Adăugarea hosturilor în Rancher

În final, se va putea observa noua gazdă în panoul accesibil prin meniul Hosts. Totodată, vor fi expuse o serie de metrici ale hostului utilizat alături de o listă a containerelor care rulează pe gazda respectivă, împreună cu starea fiecăruia. Pentru mai multe detalii legate de fiecare container, se poate accesa o vedere mai detaliată prin selectarea, prin intermediul mouse-ului, a containerului dorit.

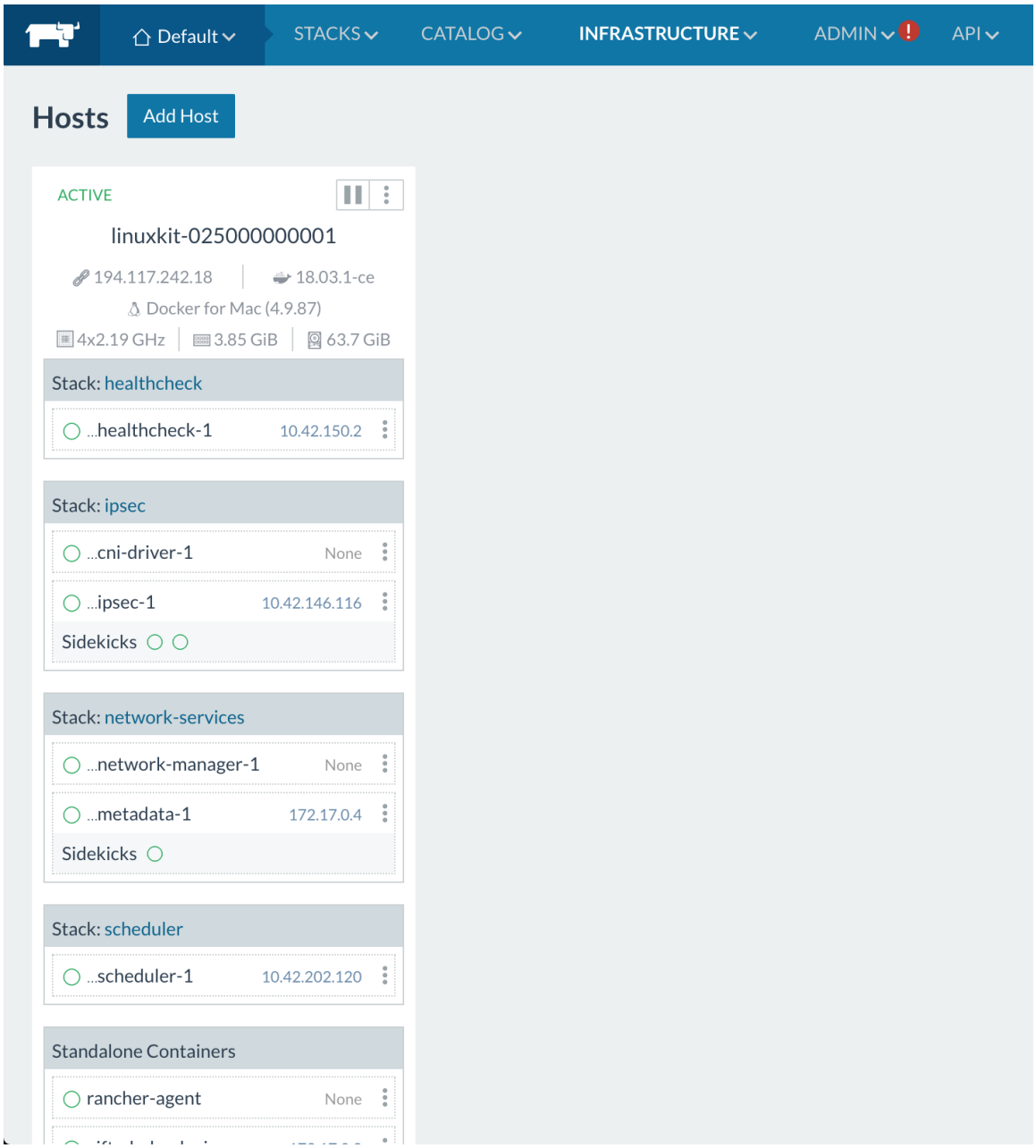

Figura 7.2 – Vizualizarea hosturilor adăugate în Rancher

Pentru instalarea sistemului prin intermediul catalogului oferit de Rancher va fi necesară utilizarea unui sistem de control al versiuni Git. Momentan, Rancher nu suportă repertorii private de Git, fiind necesară utilizarea unui repertoriu public sau trimiterea credențialelor către un repertoriu privat prin intermediul parametrilor URL, după cum urmează: ”***https://user:parola@repository-url.git***”.

Pentru utilizarea catalogului, se va crea repertoriul de Git precizat anterior, se vor încărca în acesta datele aflate în directorul **rancher-catalog** și se va înregistra în Rancher accesând opțiunea **Settings** din meniul **Admin** și adăugând un nou catalog în categoria **Catalog** a paginii **Settings**. În urma creării și înregistrării unui repertoriu Git în Rancher, se va putea observa noul catalog personalizat în meniul **Catalog**.

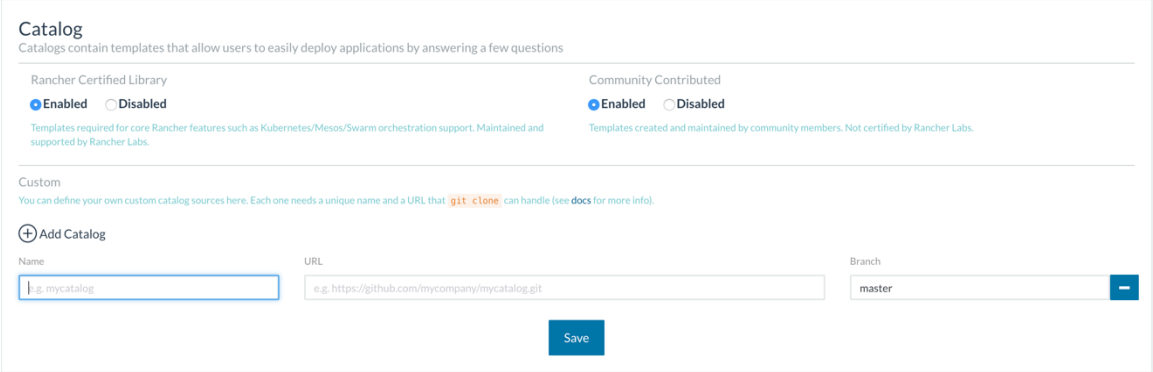

Figura 7.3 – Categoria Catalog a paginii Settings

Pentru instalarea aplicației va fi necesară utilizarea unui sistem de găzduire a imaginilor Docker. Pentru sistemul propus a fost folosit Nexus, un șablon al acestuia fiind deja parte al catalogului personalizat adăugat la pasul anterior.

Se va crea un serviciu Nexus folosind catalogul, după care va fi necesară construcția și încărcarea imaginilor Docker în serviciul Nexus. Construcție se poate realiza rulând comanda ***./gradlew build-images*** pentru UNIX sau ***gradlew.bat build-images*** pentru Windows. Încărcarea imaginilor se realizează utilizând comanda ***docker push <url-nexus>/<tag-imagine>***. Există posibilitatea ca această comandă să returneze o eroare în care se precizează faptul că repertoriul utilizat nu este sigur. În acest caz trebuie adăugat repertoriul în cadrul repertoriilor nesigure autorizate la nivelul preferințelor Docker.

În urma încărcării imaginilor și a adăugării catalogului, va fi posibilă instalarea one-click a aplicației folosind catalogul Rancher, după care este posibilă instalarea sistemului de monitorizare tot prin intermediul catalogului. Pentru o mai bună localizare a elementelor Centrum și Prometheus în catalog, se pot utiliza următoarele capturi de ecran ca și referință.

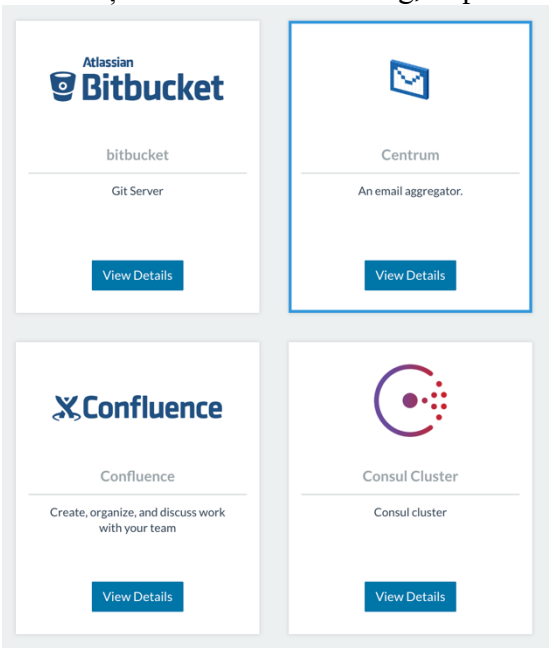

Figura 7.4 – Centrum în catalogul Rancher

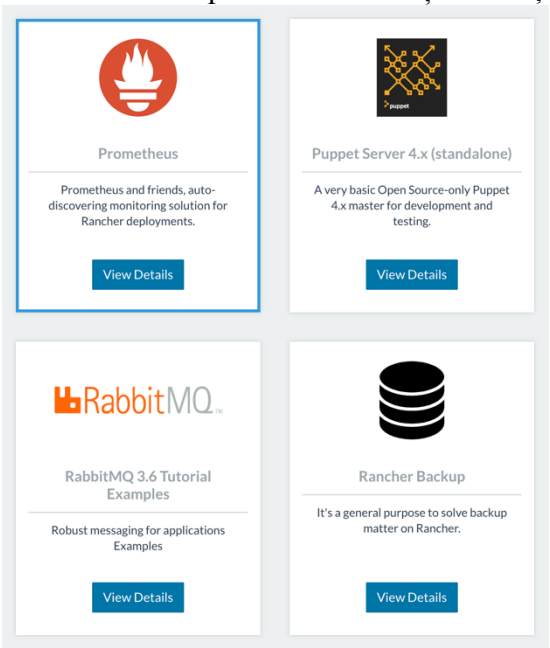

Figura 7.5 – Prometheus în catalogul Rancher

7.3. Utilizare

Utilizarea aplicației este foarte simplă datorită stilizării interfeței utilizator în raport cu standardele UX. Totodată, interfața utilizator este o interfață web, oferind utilizatorilor posibilitatea accesării aplicației folosind orice platformă care poate utiliza un navigator web. Pentru demonstrarea acestei capabilități a sistemului se poate observa următoarea figură ce afișează pagina principala a componentei Prometheus prin intermediul navigatorului w3m disponibil în cadrul liniei de comandă Linux.

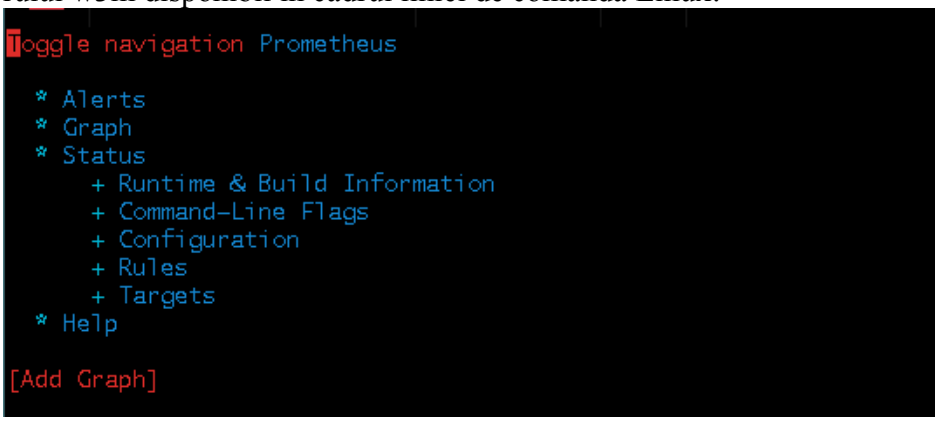

Figura 7.6 – Accesarea sistemului prin intermediul navigatorului w3m

Prin urmare, sistemul își atinge obiectivele în ceea ce privește utilizabilitatea acestuia, fiind un sistem ușor de utilizat și accesibil. Pentru detalierea modurilor de utilizare a sistemului, consultați cazurile de utilizare din cadrul capitolului 4 al acestei lucrări.

Capitolul 8. Concluzii

Acest capitol va prezenta realizările proiectului prezentat, evidențiind, totodată, posibilele dezvoltări și îmbunătățiri ulterioare.

8.1. Analiza rezultatelor obținute

Aplicația dezvoltată este foarte competitivă cu aplicațiile similare acesteia, atingându-și scopurile în această categorie. Obiectivele principale ale sistemului dezvoltat au fost atinse prin realizarea unei aplicații ușor de utilizat și de întreținut, folosind tehnologii de ultimă generație. Astfel, utilizatorii sistemului își vor putea administra simultan multiple conturi de email, iar administratorii aceluiași sistem vor putea gestiona corespunzător sistemul prin intermediul utilitarelor incluse.

8.2. Contribuții personale

Ca urmare a proiectării, dezvoltării și testării, s-a obținut un sistem complex, robust, modular, care este capabil de a îndeplini cerințele pentru care a fost creat, fiind, totodată, ușor de întreținut și administrat. Aceste caracteristici derivă din utilizarea eficace a unei arhitecturi bazată pe microservicii, sprijinită de o componentă de administrare a acesteia și de o componentă de monitorizare și alertare.

Sistemul dezvoltat este rezultatul studierii intensive a librăriilor și utilitarelor folosite. De asemenea, a fost necesară studierea și implementarea optimizărilor necesare funcționării eficiente a sistemului, acestea fiind detaliate în cadrul capitolului 6.

8.3. Dezvoltări ulterioare

Odată cu trecerea timpului, orice sistem informatic trebuie să evolueze pentru a se adapta schimbărilor pieței. Sistemul descris în această lucrare poate evolua masiv prin introducerea de noi caracteristici precum posibilitatea agregării calendarelor electronice, implementarea autentificării prin OAuth și suportarea diverselor acțiuni suplimentare de administrare a emailurilor precum marcarea acestora în diferite moduri sau instalarea unui filtru de spam.

Totodată, în funcție de evoluția grafică a aplicațiilor, Centrum trebuie să păstreze o interfață familiară și vizual plăcută pentru utilizatorii acestuia.

De asemenea, sistemul trebuie să se adapteze trendurilor din lumea software, fiind necesară o evoluție din punct de vedere al tehnologiilor folosite. Acest criteriu este ușor de urmat întrucât sistemul este ușor de modificat datorita modularității acestuia.

Bibliografie

[1] C. Posta, Microservices for Java Developers, Sebastopol, CA: O'Reilly Media, Inc., 2016.

[2] M. T. F. Martin L. Abbott, The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise (2nd Edition), Addison-Wesley Professional, 2015.

[3] W. W. S. F. N. R. Felix Gessert, NoSQL database systems: a survey and decision guidance, Berlin: Springer-Verlag, 2016.

[4] Dynatrace LLC, „Java enterprise performance," 2018. [Interactiv]. Available: https://www.dynatrace.com/resources/ebooks/javabook/.

[5] S. Ritter, „4 Reasons Why Java is Still #1 - Azul Systems Inc.," 12 January 2016. [Interactiv]. Available: https://www.azul.com/4-reasons-java-still-1/.

[6] B. Muschko, „Why Build Your Java Projects with Gradle Rather than Ant or Maven?," 8 Iulie 2014. [Interactiv]. Available: http://www.drdobbs.com/jvm/why-build-your-java-projects-with-gradle/240168608.

[7] Google, „Angular - What's Angular?," 2018. [Interactiv]. Available: https://angular.io/docs.

[8] D. Doomen, „Microservices: The State of the Union," 1 July 2016. [Interactiv]. Available: https://dzone.com/articles/microservices-the-state-of-the-union.

Anexa 1 (Glosar de Termeni)

Termen	Descriere
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface - Set de funcţii, metode sau chiar protocoale pentru stabilirea unui contract între diferite programe, pentru a realiza o comunicare.
CAP	Consistency, Availability, Partition-Tolerant
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
IP	Internet Protocol
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol - Protocol folosit pentru stabilirea regulilor de transmitere a mesajelor în cadrul World Wide Web
HTTPS	Hypertext Transfer Protocol Secure – Varianta securizată prin SSL a protocolului HTTP
JSON	Javascript Object Notation - Format folosit pentru schimbul de informaţii între diferite sisteme.
MVC	Model-View-Controller. Stil Arhitectural
NoSQL	Non SQL / Non-relational
PaaS	Platform-as-a-Service
REST	Representational State Transfer – Modalitate architecturală folosită pentru schimbul de informaţii între diferite sisteme prin intermediul protocolului HTTP
SaaS	Software-as-a-Service
SSL	Secure Socket Layer – Protocol criptografic pentru securizarea comunicării dintre sisteme
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UX	User Experience

Anexa 2 (Lista figurilor din lucrare)