

Universitatea “Politehnica” din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Studiul structurilor de tip cluster implementate în MariaDB

Proiect de diplomă

prezentat ca cerință parțială pentru obținerea titlului de

Inginer în domeniul Calculatoare și Tehnologia Informației

programul de studii de licență *Ingineria Informației(CTI-INF)*

Conducător științific

Absolvent

Ș.l.dr.ing. Valentin Pupezescu

Săcuiu Adrian


2015

Pagină lăsată intenționat goală.

Universitatea "Politehnica" din București
Facultatea de Electronică, Telecomunicații și Tehnologie Informației
Departamentul Electronică Aplicată și Ingineria Informației

Aprobat Director de Departament :

Prof.Dr.Ing.Sever Pașca



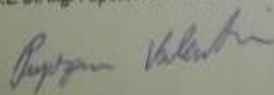
TEMA PROIECTULUI DE DIPLOMĂ

a studentului Săculu Adrian, grupa 443A

1. Titlul temei: **Studiul structurilor de tip cluster implementate în MariaDB.**
2. Contribuția practică, originală a studentului va consta în:
 - Proiectarea și dezvoltarea unei aplicații web de gestionare și assignare a asseturilor dintr-o firmă din domeniul IT;
 - Proiectarea și implementarea bazei de date utilizate de aplicație în sistemul de gestiune al bazei de date (SGBD) MariaDB;
 - Configurarea sistemelor distribuite astfel încât să se poată instala clusterul MariaDB;
 - Crearea unui modul în cadrul aplicației web care va permite testarea clusterului;
 - Se vor studia avantajele și dezavantajele arhitecturii implementate față de cazul centralizat.
3. Proiectul se bazează pe cunoștințe dobândite în principal la următoarele discipline: Programare Obiect-Orientată, Tehnologii de Programare în Internet, Proiectarea Bazelor de Date
4. Realizarea practică/ proiectul rămân în proprietatea: UPB
5. Proprietatea intelectuală asupra proiectului aparține: UPB
6. Locul de desfășurare a activității: UPB, ETTI
7. Data eliberării temei: 10 octombrie 2014

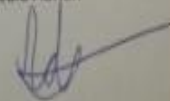
CONDUCĂTOR LUCRARE:

S.L. Dr. Ing. Pupezescu Valentin



STUDENT:

Săculu Adrian



Din cauza fișierului corupt cu imaginea Anexei 1 scanată, am atașat o poză a acesteia făcută cu aparatul foto. Voi remedia această problemă în cel mai scurt timp.

Pagină lăsată intenționat goală.

Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul "*Studiul structurilor de tip cluster implementate în MariaDB*", prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității "Politehnica" din București ca cerință parțială pentru obținerea titlului de *Inginer* în domeniul *Calculatoare și Tehnologia Informației*, programul de studii *Ingineria Informației(CTI - INF)* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurărilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, 20.06.2015

Absolvent
Adrian Săcuu



(semnătura în original)

Cuprins

LISTA FIGURILOR ȘI A TABELELOR	10
LISTA ACRONIMELOR	11
Introducere	12
Capitolul 1. Noțiuni teoretice	13
1.1. Baze de date	13
1.1.1. Clasificare.....	14
1.1.2. Arhitectura unui sistem de baze de date	16
1.1.3. Obiective principale ale SGBD-urilor	16
1.1.4. Funcțiile SGBD-urilor	17
1.2. Baze de date relaționale	19
1.2.1. Prezentare generală.....	20
1.2.2. Terminologii.....	21
1.2.3. Relații sau tabele.....	21
1.2.4. Relații de bază și derivate.....	22
1.2.5. Domenii.....	22
1.2.6. Constrângeri.....	22
1.2.7. Operații relaționale.....	23
1.3. Baze de date distribuite.....	25
1.3.1. Definiții.....	25
1.3.2. Avantaje.....	25
1.3.3. Funcții adiționale.....	26

1.4. Baze de date NoSQL.....	27
1.4.1. Tipuri de baze de date NoSQL.....	27
1.4.2. Avantaje.....	27
1.4.3. NoSQL vs SQL.....	29
Capitolul 2. Conceptul de MariaDB Cluster.....	30
2.1. Conceptul de clustering.....	30
2.2. Arhitecturile unui cluster.....	30
2.2.1. Shared-Nothing.....	31
2.2.2. Shared-Disk.....	32
2.3. MariaDB Galera.....	34
2.3.1 Avantaje.....	34
2.3.2. Arhitectura.....	35
2.3.3. Nivele de izolare.....	36
2.4. Funcționarea unui cluster MariaDB Galera.....	39
2.5. Limitări.....	39
Capitolul 3. Tehnologii utilizate.....	40
3.1. Limbaje.....	40
3.1.1. HTML.....	40
3.1.2. Java EE	41
3.1.3. Javascript.....	42
3.1.4. CSS.....	43
3.2. Framework-uri.....	44
3.2.1. Spring.....	44
3.2.2. Hibernate.....	46
3.3. Librarii.....	47

3.3.1. JQuery.....	47
3.4. Instrumente de asamblare a aplicației	48
3.4.1. Maven	48
3.5. Medii de dezvoltare	48
3.5.1. Eclipse	48
Capitolul 4. Descrierea aplicației	50
4.1. Instalare programelor necesare aplicației	51
4.1.1. Instalare MariaDB Galera Server	51
4.1.2. Instalare Java	51
4.1.3. Instalare Eclipse.....	53
4.2. Prezentarea aplicației.....	54
4.2.1. Baza de date.....	54
4.2.2. Modulul de înregistrare.....	55
4.2.3. Modulul de autentificare.....	57
4.2.4. Modulul de profilul meu.....	58
4.2.4. Funcționalitățile angajatului.....	59
4.2.4.1. Modulul de Asset-uri.....	59
4.2.4.2. Modulul de Plângeri.....	60
4.2.4.3. Modulul de Cereri.....	60
4.2.4.4. Modulul de Tranzacții.....	60
4.2.5. Funcționalitățile administratorului.....	61
4.2.5.1. Modulul de Users.....	61
4.2.5.2. Modulul de Asset-uri.....	61
4.2.5.3. Modulul de Plângeri.....	62
4.2.5.4. Modulul de Cereri	62
4.2.5.5. Modulul de Tranzacții.....	63

4.2.6. Securitatea aplicației.....	63
CONCLUZII.....	67
Bibliografie.....	68
ANEXE.....	70
Anexa 1	70
Anexa 2	76
Anexa 3	78
Anexa 4	81
Anexa 5	85

LISTA FIGURILOR ȘI A TABELELOR

Tabel 1.1 Terminologii	21
Tabel 1.2 NoSQL vs SQL	29
Figura 2.1 Shared-Nothing (partiționarea datelor între noduri)	31
Figura 2.2 Configurație Master – Slave	32
Figura 2.3 Arbore de Slave	32
Figura 2.4 Cluster Shared- Disk (toate nodurile au acces la toate datele)	33
Figura 2.5 Funcționarea unui cluster MariaDB Galera	39
Figura 4.1 Schema bazei de date	55
Figura 4.2. Pagina de start	56
Figura 4.3. Diagrama de secvență pentru înregistrarea încheiată cu succes	57
Figura 4.4. Diagrama de secvență pentru autentificarea încheiată cu succes	58
Figura 4.5 Pagina de My Profile	59
Figura 4.6 Pagina de detalii a asset-ului	60
Figura 4.7 Pagina de asset-uri a administratorului	61
Figura 4.8 Formularul de adăugare a noului asset	62
Figura 4.9 Pagina administratorului de vizionare a cererii	63
Figura 4.10 Bucată din codul paginii de start	65
Figura 4.11 Pagina de eroare pentru acces interzis	66

Lista acronimelor

CSS - Cascading Style Sheet

DB - Database

PK - Primary Key

FK - Foreign Key

HTML - HyperText Markup Language

XML - eXtended Markup Language

JS - JavaScript

JSP - Java Server Pages

JSON - JavaScript Object Notation

MAP - Abstract data structure

MySQL - Open source relational database management system

MariaDB - Open source relational database management system

NoSQL - Not Only SQL

SGBD - Sistem de Gestiune a Bazelor de Date

SGBDR - Sistem de Gestiune a Bazelor de Date Relationale

SGBDD - Sistem de Gestiune a Bazelor de Date Distribuite

SQL - Structured Query Language

VARCHAR - Variable Length Character String

TAGLIB - Librarie de tag-uri(etichete)

MVC - Model-View-Controller

Introducere

Am ales această temă deoarece consider că folosirea unui cluster de baze de date reprezintă un avantaj uriaș față de o bază de date simplă. Să luăm exemplul unei aplicații online, care este accesată de un număr mare de utilizatori. În cazul în care serverul pe care se află baza de date prezintă probleme și nu mai poate funcționa, atunci și aplicația nu va mai putea rula, ceea ce poate duce la pierderi importante.

Folosind un cluster de baze de date, această problemă ar fi eliminată deoarece la defectarea unui nod, imediat ar intra automat în funcțiune alt server, menținând aplicația activă.

Clusterul este format din mai multe noduri de baze de date, configurația fiind de minim 3 servere. Acestea conțin aceleași date, iar la modificarea uneia dintre bazele de date, celelalte preiau informațiile și le actualizează.

Obiective

Obiectivul principal al acestui proiect este de a crea o aplicație web de gestionare și asignare a asset-urilor într-o companie IT, care folosește un cluster de baze de date MariaDB pentru stocarea tuturor datelor și informațiilor.

Contribuție personală

Menționez că aplicația a fost creată de la zero, integral de către mine, aceasta incluzând crearea și configurarea clusterului MariaDB, design-ul aplicației și toate funcționalitățile acesteia.

Tehnologii folosite

Tehnologiile folosite pentru realizarea acestui proiect sunt următoarele: MariaDB Galera, HTML, JSP, JavaScript, JQuery, CSS, Java EE, Spring, Hibernate, Maven, Eclipse.

Capitolul 1. Noțiuni teoretice

1.1. Baze de date

Bazele de date și sistemele de baze de date sunt o componentă esențială a vieții în societatea modernă: majoritatea dintre noi se lovesc în fiecare zi de diferite activități care implică interacțiune cu o bază de date. De exemplu, dacă mergem la bancă să retragem sau să depunem numerar, dacă facem o rezervare la un hotel sau pentru un bilet de avion, dacă accesăm o bibliotecă online pentru a căuta o carte, sau dacă cumpărăm ceva online – cum ar fi o carte, o jucărie sau un calculator, sunt șanse ca aceste activități să implice pe cineva sau pur și simplu un program pe calculator care să acceseze o bază de date. Chiar și atunci când facem cumpărături într-un supermarket, baza de date care conține inventarul produselor se actualizează frecvent” [7]

O bază de date este o colecție de date ce se aseamănă. Prin date, ne referim la diferite informații care sunt salvate și care au un sens implicit. De exemplu, să considerăm numele, numerele de telefon și adresele persoanelor pe care le știm. Se poate să fi salvat aceste informații într-o agendă sau pe un hard disk, folosind calculatorul personal. Această colecție de date, ce se aseamănă și care au un sens implicit, se numește bază de date. [7]

Definiția precedentă a unei baze de date este foarte generală; ca de exemplu, considerăm colecția de cuvinte care formează această lucrare drept o bază de date. Totuși, utilizarea cuvântului *bază de date* este de obicei mult mai limitată. O bază de date are următoarele caracteristici:

- O bază de date reprezintă un aspect din viața reală, de cele mai multe ori denumit *mini lume* sau *universul discuției*. Modificări ale *mini lumii* sunt reflectate în baza de date.
- O bază de date este o colecție de date logică și coerentă, care are o semnificație inerentă. Dacă am lua la întâmplare o varietate de date, aceasta nu s-ar putea denumi bază de date.
- O bază de date este gândită, construită și populată cu date, cu un scop anume. Are destinat un anumit grup de utilizatori și câteva aplicații preconceptuate de care aceștia sunt interesați.

O bază de date poate avea orice dimensiune și poate fi oricât de complexă. De exemplu, lista numelor și adreselor la care ne-am referit în rândurile de mai sus este formată din câteva sute de arhive, fiecare având câte o structură simplă. Pe de altă parte, catalogul electronic al unei librării poate conține jumătate de milion de cărți, toate organizate pe diferite categorii, cum ar fi pe numele autorului, pe subiect, pe titlul cărții. [7]

Un sistem de gestionare a bazelor de date (SGBD) este o colecție de programe care permite utilizatorilor să creeze și să mențină o bază de date. SGBD-ul este un sistem de software cu un scop general care facilitează procesele de definire, construire, manipulare și partajarea datelor din baza de date între mai mulți utilizatori și diferite aplicații. Pentru a defini o bază de date este nevoie să se specifice tipul de date, structurile și limitările informațiilor ce se vor adăuga în baza de date. Definiția unei baze de date sau informația descriptivă este stocată de către SGBD sub forma unui catalog de baze de date sau a unui dicționar; este denumit meta-data.

Manipularea unei baze de date include diferite funcții cum ar fi interogarea bazei de date în scopul preluării de informații, actualizarea bazei de date pentru a reflecta schimbări în mini lume și generarea de rapoarte. Partajarea resurselor unei baze de date permite mai multor utilizatori/ programe să le acceseze în același timp. [7]

1.1.1. Clasificare

Se folosesc diferite criterii în clasificarea SGBD-urilor. Primul este modelul de date folosit, pe care este bazat SGBD-ul. Modelul principal de date utilizat în SGBD-urile actuale este modelul relațional de date. Modelul de date obiect a fost implementat în câteva sisteme comerciale, dar nu a ajuns să aibă o utilizare răspândită. Diferite aplicații mai vechi încă funcționează pe sistemele de baze de date ce folosesc modele de date ierarhice și de rețea. Rețeaua modelului de date a fost folosit de mai mulți importatori, iar rezultatul produselor, cum ar fi IDMS (Cullinet -acum Computer Associates), DMS 1100 (Univac – acum Unisys), IMAGE (Hewlett-Packard), VAXDBMS (Digital - atunci Compaq și acum HP), și SUPRA (Cincom) încă au o continuare și grupul lor de utilizatori au propriile lor organizații. Dacă e să adăugăm la acestea sistemul de fișiere VSAM al lui IBM, putem afirma cu încredere faptul că un procent destul de mare al informațiilor computerizate din toată lumea se află în așa numitele „sisteme vechi de bază de date”. [7]

SGBD-urile relaționale se dezvoltă în continuare, respectiv acestea încorporează multe din conceptele dezvoltate în bazele de date obiectuale. Aceasta a condus la o nouă clasă de SGBD-uri, denumite SGBD-uri obiect relaționale. Putem clasifica SGBD-urile în funcție de modelul de date: relaționale, obiectuale, obiect relaționale, ierarhice, rețele, și altele.

Mai mult, alte SGBD-uri se bazează pe modelul XML (eXtended Markup Language). XML este un model de date structurat (ierarhic) sub forma unui copac. Acestea au fost denumite SGBD-uri XML native. Diferite SGBD-uri relaționale au adăugat interfețe XML, respectiv memorie la produsele lor. [7]

Al doilea criteriu folosit pentru clasificarea SGBD-urilor este numărul de utilizatori acceptați de sistem. Sistemele ce folosesc un singur utilizator acceptă doar unul singur și funcționează cel mai des împreună cu PC-uri. Sistemele ce folosesc mai mulți utilizatori, și care includ majoritatea SGBD – urilor, funcționează cu mai mulți utilizatori în același timp.

Al treilea criteriu este numărul de site-uri la care baza de date este distribuită. Un SGBD este centralizat dacă informațiile sunt cuprinse pe un singur site de pe calculator. Un SGBD centralizat poate accepta mai mulți utilizatori, dar SGBD- ul împreună cu baza de date se află doar pe un singur computer. Un SGBD poate avea baza de date împreună cu software- ul SGBD distribuite pe mai multe calculatoare și conectate la o rețea de calculatoare. SGBD – urile omogene utilizează același software SGBD pe toate calculatoarele, în timp ce SGBD – urile eterogene pot utiliza diferite software – uri SGBD pe fiecare calculator. Este posibilă și dezvoltarea unui *middleware software* cu scopul accesării diferitelor date pre existente sub SGBD – urile eterogene. Aceasta duce la un SGBD federalizat (sau un multi sistem de baze de date), la care SGBD – urile participante sunt cuplate și au un anumit nivel de autonomie locală. Multe SGBD – uri utilizează arhitectura client – server.

Al patrulea criteriu este costul. Este dificilă propunerea unei clasificări a SGBD – urilor bazată pe cost. În ziua de azi există surse gratuite, cum ar fi MySQL sau PostgreSQL, care sunt susținute de furnizori terți. Principalele produse SGBDR (Sisteme de gestiune a bazelor de date relaționale) sunt disponibile ca și versiuni de 30 de zile, respectiv versiuni personale, și care ar costa sub \$100 având un nivel de calitate nu foarte ridicat. Sistemele foarte mari se vând sub formă modulară, având componente ce suportă distribuirea, copierea, procesarea paralelă, capabilitatea mobilă, ș.a.m.d, dar și un număr mare de parametrii care trebuie definiți pentru configurare. [7]

În plus, sunt vândute sub formă de licențe – licențele de calculatoare permit utilizarea nelimitată a sistemelor de baze de date. Un alt tip de licență limitează numărul de utilizatori concurenți sau numărul de locuri al utilizatorilor într-o locație. Anumite versiuni de sisteme cu un singur utilizator, cum ar fi Microsoft Access sunt vândute per copie sau sunt incluse în configurația desktop – ului sau a calculatorului. În plus, stocarea datelor dar și ajutorul pentru diferite tipuri de date în plus, sunt disponibile la un extra cost. Instalarea, respectiv întreținerea anuală a sistemelor de baze de date de mari dimensiuni poate duce la costuri de milioane de dolari. [7]

SGBD – urile se mai pot clasifica și pe baza modalităților de tipuri de acces pentru depozitarea documentelor. O bine cunoscută familie de SGBD se bazează pe structuri de fișiere inversate. În concluzie, SGBD – urile pot fi cu scop general sau cu scop special. Dacă performanța reprezintă o caracteristică importantă, SGBD – ul cu scop special poate fi creat pentru o anumită aplicație. Un astfel de sistem nu poate fi folosit pentru alte aplicații fără a suferi schimbări majore. Au existat multe rezervări de zbor, respectiv sisteme de telefonie create în trecut și care sunt SGBD – uri cu scop special. Acestea se regăsesc în categoria sistemelor de procesare de tranzacții online, care suportă un număr mare de tranzacții concurente fără a suferi întârzieri multe. [7]

În continuare, vom descrie principalul criteriu d clasificare a SGBD – urilor: modelul de date. Modelul de bază de date relaționale reprezintă baza de date ca și o colecție de mese, unde fiecare masă poate fi depozitată ca un document separat. Cele mai multe baze de date relaționale utilizează un limbaj de interogare la nivel înalt, denumit SQL.

Modelul de date obiect definește o bază de date în ceea ce privește obiectele, caracteristicile și activitățile lor. Obiectele care au aceeași structură, respectiv comportament aparțin unei anumite categorii, iar aceste categorii sunt organizate pe diferite ierarhii. Activitățile fiecărei categorii sunt

specificate în funcție de anumite proceduri predefinite, numite metode. SGBD – urile relaționale și – au extins modelele cu scopul de a încorpora anumite concepte de baze de date obiect și alte capabilități; aceste sisteme se numesc sisteme obiect relaționale sau relațional extinse. [7]

Ne referim la modelul XML ca și un standard pentru schimbul de date online, respectiv a fost utilizat ca o bază pentru implementarea diferitor prototipuri de sisteme native XML. XML folosește structuri ierarhice și combină conceptele bazelor de date cu concepte din documentele modelelor de reprezentare. Datele sunt reprezentate ca elemente, respectiv acestea pot fi utilizate pentru a crea structuri ierarhice complexe. Acest model se aseamănă foarte mult cu modelul de obiecte dar folosește terminologii diferite.

Modelul ierarhic reprezintă datele ca o structură ierarhică. Fiecare ierarhie reprezintă un număr de înregistrări similare. Nu există un limbaj standard pentru modelul ierarhic. [7]

1.1.2. Arhitectura unui sistem de baze de date

Arhitectura pachetelor de SGBD a evoluat constant, începând de la sistemele monolitice, unde întregul sistem de software SGDB era un sistem bine integrat, până la modernul pachet SGDB ce sunt prezentate modular pe partea de design, împreună cu un sistem arhitectural de tip client/server. Această evoluție reflectă întocmai trend-urile în domeniul calculatoarelor, unde calculatoarele centralizate, de dimensiuni mari, sunt înlocuite treptat de sute de stații de lucru distribuite în mai multe părți regionale și de asemenea, calculatoare personale conectate prin intermediul rețelelor de comunicare, la numeroase tipuri de server- Web servers, database servers, file servers, application servers, s.a.m.d. [7]

Design-ul unui sistem SGBD depinde, în mare măsură, de arhitectura lui. Acesta poate să fie centralizat, decentralizat sau ierarhic. Arhitectura unui sistem SGBD poate fi clasificată în funcție de nivelul de accesare, ce poate fi pe un singur nivel sau pe numeroase niveluri. În cazul în care ne referim la o arhitectură a sistemului SGBD pe „n-niveluri”, aceasta împarte întregul sistem într-un circuit de „n-module”. Aceste module sunt independente, însă asociate între ele și pot fi modificate, schimbate sau înlocuite independent de către utilizator.

În cazul arhitecturii sistemului SGBD de la primul nivel, sistemul SGBD reprezintă singura entitate accesată de către utilizator. Orice modificare efectuată la acest nivel va fi înregistrată direct în sistemul SGBD. Sistemul nu furnizează instrumente practice pentru utilizatori. Designerii bazei de date și programatorii înclină spre utilizarea unei arhitecturi pe un singur nivel.

În cazul arhitecturii sistemului SGBD de la doilea nivel, este obligatoriu ca sistemul să aibă o aplicație prin intermediul căreia sistemul SGBD poate fi accesat. Programatorii folosesc acest tip de arhitectură de nivel 2 în momentul în care accesează SGBD-ul prin intermediul unei aplicații. În acest caz, este în întregime independent față de baza de date în ceea ce privește design-ul, programarea și utilizarea.

În cazul arhitecturii sistemului SGBD de la treilea nivel, aceasta separă nivelurile sale unul față de celălalt, având la bază complexitatea utilizatorilor și modul de utilizare a datelor prezente deja în

baza de date. Acest tip de arhitectura este cel mai folosit de catre utilizatori, in vederea proiectării unui sistem SGBD. [7]

Nivelul Bazei de Date - La acest nivel, baza de date se afla in legătura cu interogarea limbilor de proces. De asemenea, sunt regăsite relațiile ce definesc datele si constrângerile acestora la acest nivel.

Nivelul Aplicației - La acest nivel se afla server-ul de aplicație si programele ce accesează baza de date. Pentru un utilizator, acest nivel reprezintă o viziune abstracta a bazei de date.

Utilizatorii nu sunt conștienți de existenta unei baze de date, mai departe de acest punct. Pe de alta parte, la nivelul bazei de date nu este conștient de existenta vreunui utilizator, mai departe de nivelul aplicației. [7]

Prin urmare, nivelul aplicației se poziționează la mijlocul relației dintre baza de date si utilizator. De asemenea, nivelul aplicației se comporta ca un mediator intre utilizator si baza de date.

Nivelul utilizatorului - Utilizatorii reprezintă actorii principali ce operează in cadrul acestui nivel si aceștia nu sunt conștienți de existenta bazei de date, mai departe de acest nivel. In cadrul acestui nivel, numeroase interpretări ale bazei de date pot fii furnizate de către aplicației. Toate interpretările sunt generate de către aplicație, ce se regăsește la nivelul aplicației.

Arhitectura bazei de date pe mai multe niveluri este extrem de modificabila, ca urmare a faptului că aproximativ toate componentele sale sunt independente si pot fii modificate intr-o maniera independenta. [7]

1.1.4. Funcțiile SGBD-urilor

Sunt numeroase funcții pe care SGBD-urile le efectuează in vederea asigurării integrității si consistenței datelor in baza de date. Cele 10 funcții in cadrul SGBD-ului sunt: administrarea definițiilor de date , administrarea stocării de date, transformarea si prezentarea datelor, administrarea securității, controlul de acces pentru utilizatori multipli, administrarea recuperării de informații si fișierul de rezerva ce păstrează copia de rezerva a datelor, administrarea integrității datelor, limbile de acces a bazei de date si interfața de programare a aplicațiilor, interfața de comunicare a bazelor de date si administrarea tranzacțiilor.

1. Administrarea definițiilor de date

Dicționarul de date este locul unde SGBD stochează definițiilor elementelor si relațiile dintre date(metadata). SGBD-ul utilizează aceasta funcție pentru a identifica structurile componentelor de date si relațiile dintre acestea. Când programele accesează date intr-o baza de date, acestea parcurg

practic sistemul SGBD. Aceasta funcție înlătură dependența și structura de date cu abstractizarea datelor. Ca o consecință, aceasta ușurează munca utilizatorului. În cele mai multe cazuri, dicționarul de date este ascuns de utilizator și este folosit de către administratorii bazei de date și programatori. [8]

2. Administrarea stocării de date

Această funcție particulară este utilizată pentru stocarea datelor și alte modalități de introducere a datelor, definiții de raportare, reguli de validare a datelor, cod procedural, și structuri ce pot manevra formatori de tip video și foto. Utilizatorii nu sunt nevoiți să știe cum datele și informațiile sunt stocate sau manipulate. În alcătuirea acestei structuri se poate regăsi și un proces nou numit „performance tuning”. Acest proces este asociat cu eficiența unei baze de date, în strânsă legătură cu procesul de stocare și viteza de acces la baza de date. [8]

3. Transformarea și prezentarea datelor

Această funcție a fost dezvoltată pentru a transforma orice informație nou introdusă în structuri de date cerute. Prin utilizarea acestei funcții, SGBD-ul poate determina diferența dintre următoarele tipurile de date: tip logic sau tip “physical”. [8]

4. Administrarea securității

Aceasta reprezintă una dintre cele mai importante funcții în cadrul unui SGBD. Administrarea securității stabilește reguli ce determină exact utilizatorii ce pot avea acces la baza de date sau nu. Utilizatorii primesc un nume de utilizator și o parolă sau câteodată, pot avea acces la baza de date prin autentificare biometrică (de exemplu scanarea amprente sau scanarea retinei). Însă, autentificarea biometrică tinde să fie mai costisitoare în comparație cu celelalte metode de autentificare.

De asemenea, această funcție restricționează tipul de date pe care un utilizator le poate accesa sau administra. [8]

5. Controlul de acces pentru utilizatori multipli

Integritatea și consistența datelor reprezintă baza acestei funcții. Accesul de control pentru utilizatorii multipli este un instrument extrem de folositor în cadrul sistemului SGBD, oferă posibilitatea unui număr de utilizatori să acceseze simultan baza de date, neexistând riscul de a fi afectată integritatea bazei de date. [8]

6. Administrarea recuperării de informații și fișierului de rezerva ce păstrează copia de rezerva a datelor

Fișierul de rezerva ce conține copia de rezerva a datelor și procesul de recuperare a datelor este adus la lumina de fiecare dată când există o posibilă amenințare la adresa distrugerii bazei de date. De exemplu, dacă apare o pană de curent, administrarea recuperării de date implică timpul necesar de recuperare a bazei de date după ce pană de curent a intervenit. Fișierul de rezerva ce păstrează copia de

rezerva a datelor se refera la siguranța și integritatea datelor; de exemplu, crearea unui fișier de rezerva ce va stoca toate fișierele de tip mp3 pe un disc portabil. [8]

7. Administrarea integrității datelor

SGBD-ul impune aceste reguli cu scopul de a eradica posibile amenințări informatice, de exemplu redundanța datelor. Redundanța datelor apare atunci când datele sunt depozitate inutil în mai multe locuri. De asemenea, tot prin intermediul sistemului SGBD, se poate urmări și maximizarea consistenței datelor. În cadrul acestei acțiuni, se asigură faptul că baza de date returnează același răspuns, într-o manieră corectă, de fiecare dată când aceeași întrebare este adresată sistemului. [8]

8. Limbaje de acces al bazelor de date și interfețe de programare de aplicații

Limbajul de interogare este un limbaj non procedural. Un exemplu este SQL (structured query language). SQL este cel mai comun limbaj de interogare promovat de cei mai mulți furnizori de SGBD – uri. [8]

9. Interfețe de comunicare între baze de date

Aceasta se referă la cum SGBD – ul poate accepta diferite cereri de utilizatori finali prin diferite medii de rețea. Un exemplu în acest sens poate fi ușor raportat la internet. Un SGBD poate oferi acces la baza de date prin accesul la internet cu ajutorul motoarelor de căutare (Mozilla Firefox, Internet Explorer, Netscape). [8]

10. Administrarea tranzacției

SGBD trebuie să furnizeze o metodă care va garanta faptul că toate actualizările într-o anumită tranzacție sunt sau nu finalizate. Toate tranzacțiile trebuie să urmărească următorul model, denumit proprietățile ACID:

A – Atomicitate: Definiște faptul că o tranzacție este o unitate indivizibilă care se derulează ca un întreg și nu una câte una.

C – Consistență: Tranzacția trebuie să modifice baza de date de la o stare constantă la o altă stare constantă.

I – Izolare: Tranzacțiile trebuie îndeplinite în mod independent una de cealaltă. O parte din tranzacția în derulare nu trebuie să fie văzută de cealaltă tranzacție.

D – Durabilitate: O tranzacție completată cu succes este înregistrată pentru totdeauna în baza de date și nu trebuie să fie pierdută datorită anumitor erori. [8]

1.2. Baze de date relaționale

O **bază de date relațională** este o bază de date digitală a cărei organizare se bazează pe modelul de date relaționale. Acest model organizează datele în unul sau mai multe tabele (sau "relații") cu rânduri și coloane, cu o cheie unică pentru fiecare rând. În general, fiecare tip de entitate descris într-o bază de date are propriul tabel, rândurile reprezintă situații ale acelei entități, iar coloanele reprezentând valori atribuite entității. Deoarece fiecare rând dintr-un tabel are propria sa cheie unică, rândurile din tabele pot fi legate de rânduri din alte tabele prin stocarea cheii unice la rândul la care ar trebui să fie legată (în acest caz, aceasta cheie unică este numită "cheie străină "). Codd a arătat că relațiile dintre datele de complexitate arbitrară pot fi reprezentate folosind acest set simplu de concepte. [11]

Înainte de apariția acestui model, bazele de date au fost de obicei de formă ierarhică și se tindea spre a fi organizate folosind un mixt unic de indici, lanțuri și indicatori. Simplitatea modelului relațional a făcut ca în curând să devină tipul predominant de baze de date.

Diferitele sisteme software utilizate pentru a menține bazele de date relaționale sunt cunoscute ca Sisteme de Gestionare al Bazelor de Date Relaționale (SGBDR).

Practic toate sistemele de baze de date relaționale folosesc SQL (Structured Query Language) ca limbaj pentru interogarea și menținerea bazei de date. [11]

1.2.1. Prezentare generală

Fiecare bază de date este o colecție de tabele, care sunt numite „relații”, de unde și numele de "bază de date relațională". Fiecare tabel este o reprezentare metaforică a unei entități sau obiect care este reprezentat într-o formă tabelară, de coloane și rânduri. Coloanele sunt domeniile unei înregistrări sau atributele unei entități. Rândurile conțin valorile sau instanțele datelor; de asemenea, acestea sunt denumite înregistrări sau tupluri.

Există legături atât între coloanele unui tabel, cât și între tabele. Aceste relații iau trei forme logice: unu-la-unu, unu-la-mai multe sau mai multe-la-mai multe. Cele mai multe baze de date relaționale sunt proiectate astfel încât să existe o singură valoare în fiecare celulă (o intersecție dintre o coloană și un rând); în această formă, într-un tabel există doar relații unu-la-unu. Fiecare tabel este numit în funcție de datele pe care le conține, cum ar fi oameni sau adrese. [11]

Pentru ca un sistem de gestiune de baze de date (SGBD) să funcționeze eficient și precis, trebuie să aibă operațiuni ACID. O parte din acest proces implică faptul că, în mod constant, există posibilitatea de a selecta sau de a modifica unul și doar un singur rând dintr-un tabel. Prin urmare, cele mai multe implementări fizice au un sistem atribuit, o cheie primară unică pentru fiecare tabel. Când un rând nou este scris în tabel, sistemul generează și scrie noua valoare unică pentru cheia primară (PK); aceasta este cheia pe care sistemul o utilizează inițial pentru accesarea tabelului. Performanța sistemului este optimizat pentru PK. Alte chei, mult mai explicabile, pot fi de asemenea identificate și definite drept chei alternative (AK). De multe ori, mai multe coloane pot fi necesare pentru a forma un AK (aceasta este unul din motivele pentru care o singură coloană formează de obicei un PK). Atât PK și AK au capacitatea de a identifica în mod unic un rând dintr-un tabel. Tehnologie suplimentară poate

fi aplicată pentru a asigura un ID unic în lume, un identificator global unic; acestea sunt utilizate atunci când există un sistem mai larg cerințe. [10]

Cheile principale dintr-o bază de date sunt folosite pentru a defini relațiile dintre tabele. Atunci când PK migrează la un alt tabel, acesta devine o cheie străină în celălalt tabel. Intrucât fiecare celulă poate conține doar o valoare, iar PK migrează într-o entitate dintr-un alt tabel, acest format de model poate reprezenta fie o relație de tip unu-la-unu, fie unu-la-mai multe. Cele mai multe modele de date relaționale rezolvă relații de tip mai multe-la-mai multe, prin crearea unui tabel suplimentar ce conține PK din tabelele celorlalte entități - această relație devine o entitate; apoi tabelul de rezoluție este numit în mod corespunzător și de obicei i se atribuie propria PK, în timp ce cele două FK sunt combinate pentru a forma un AK. Migrația PK la alte tabele este doilea motiv major pentru care sistemele desemnate de numere întregi sunt utilizate în mod normal ca PK; de obicei, nu prezintă nici eficiență, nici claritate migrarea a mai multor alte tipuri de coloane. [10]

Cea mai mare parte din programarea SGBDR este realizată folosind proceduri de stocare. De multe ori pot fi folosite proceduri pentru a reduce semnificativ cantitatea de informație transferată în interiorul și în afara unui sistem. Pentru creșterea siguranței, proiectarea sistemului poate acorda acces numai la procedurile de stocare și nu direct la tabele. Procedurile fundamentale de stocare conțin informațiile necesare pentru a insera noi date și a le actualiza pe cele existente. Procedurile mai complexe pot fi create pentru a implementa noi norme și o logică suplimentară legate de prelucrarea sau selectarea datelor. [10]

1.2.2. Terminologii

Termeni SQL	Terminologie privind bazele de date relaționale	Descriere
Rând	Tuplu sau înregistrare	Un set de date reprezintă un singur element
Coloană	Atribut sau domeniu	Un element etichetat de un tuplu, de exemplu, "Adresă" sau "Data nașterii"
Tabel	Relație sau relație de variabile de bază	Un set de tupluri care împărtășesc aceleași atribute; un set de coloane și rânduri
Vizualizare sau ansamblu de rezultate	Relație variabilă derivată	Orice set de tupluri; un raport din SGBDR, ca răspuns la o interogare.

Tabel 1.1 Terminologii [10]

1.2.3. Relații sau tabele

O *relație* este definită ca un set de tupluri care au aceleași atribute. Un tuplu reprezintă de obicei un obiect și informația despre acel obiect. Obiectele sunt în general obiecte fizice sau concepte.

O relație este de obicei descrisă ca un tabel, organizat în rânduri și coloane. Toate datele care fac referire la un atribut sunt în același domeniu și sunt în conformitate cu aceleași constrângeri. [10]

Modelul relațional specifică că tuplurile dintr-o relație nu au o ordine specifică și că tuplurile, la rândul lor, nu impun o anumită ordine atributelor. Aplicațiile accesează bazele de date specificând interogări care folosesc operațiuni precum „select” pentru a identifica tuplurile, „proiect” pentru a identifica atributele sau „join” pentru a realiza relații. Relațiile pot fi modificate folosind adăugare, ștergere și operatori de actualizare. Noile tupluri pot furniza valori explicite sau pot fi derivate din interogări. În mod similar, interogările identifică tuplurile pentru actualizare sau ștergere.

Prin definiție, tuplurile sunt unice. Dacă tuplul conține un candidat sau o cheie primară atunci în mod evident este unic; cu toate acestea, o cheie primară nu trebuie să fie definită de un rând sau o înregistrare pentru a fi un tuplu. Definiția unui tuplu impune ca aceasta să fie unic, dar nu are nevoie de o cheie primară pentru a defini acest lucru. Deoarece un tuplu este unic, atributele sale, prin definiție, reprezintă o super-cheie. [10]

1.2.4. Relații de bază și derivate

Într-o bază de date relațională, toate datele sunt stocate și accesate prin intermediul relațiilor. Relațiile care stochează datele sunt numite "relații de bază", iar în implementări sunt numite "tabele". Alte relații nu stochează date, dar sunt calculate prin aplicarea operațiunilor relaționale altor relații. Uneori aceste relații sunt numite "relații derivate". În implementări acestea sunt numite "opinii" sau "interogări". Relațiile derivate sunt oportune în sensul că acestea acționează ca o singură relație, chiar dacă pot prelua informații din mai multe relații. De asemenea, relațiile derivate poate fi folosit ca strat de abstractizare. [10]

1.2.5. Domenii

Un domeniu descrie un set de valori posibile pentru un atribut dat și poate fi considerat o restricție a valorii atributului. Matematic, atașarea unui domeniu unui atribut înseamnă că orice valoare a atributului trebuie să fie un element al setului specificat. Șirul de caractere "ABC", de exemplu, nu este în domeniul numerelor întregi, dar valoarea întregă 123 este. Un alt exemplu este domeniul ce descrie valorile posibile pentru câmpul "Sex" ca ("Bărbat" femeie "). Astfel, câmpul "Sex" nu va accepta valori de intrare, cum ar fi (0.1) sau (M, F).

1.2.6. Constrângeri

O cheie primară specifică în mod unic o tuplu într-un tabel. Pentru ca un atribut să fie o bună cheie primară, acesta nu trebuie să se repete. În timp ce atributele fizice (attribute utilizate pentru a descrie datele ce au fost introduse) sunt uneori chei primare bune, cheile surogat sunt folosite adesea în locul lor. O cheie surogat este un atribut alocat în mod artificial unui obiect pentru a-l putea identifica în mod unic (de exemplu, într-un tabel cu elevii unei școli, poate fi atribuit fiecărui elev un anumit ID pentru a-i diferenția). Cheia surogat nu are nicio semnificație intrinsecă (inerentă), ci mai degrabă este

un mijloc util dat de abilitatea acestuia de a identifica în mod unic un tuplu. Un alt fenomen comun, mai ales în ceea ce privește cardinalitatea N:M este cheia compozit. O cheie compozit este o cheie formată din două sau mai multe atribute dintr-un tabel care (împreună) identifică în mod unic o înregistrare. (De exemplu, într-o bază de date ce cuprinde studenții, profesorii și orele: orele ar putea fi identificate în mod unic printr-o cheie compozit reprezentând numărul sălii și interval orar alocat, deoarece nici o altă oră nu ar putea avea exact aceeași combinație de atribute). De fapt, utilizarea unei astfel de cheie compozit poate reprezenta o formă de verificare a datelor, deși o formă slabă. [10]

O cheie externă este un domeniu dintr-un tabel relațional care se potrivește coloanei cheii primare dintr-un alt tabel. Cheia străină poate fi utilizată la tabele cu referințe încrucișate. Chei străine nu presupun valori unice în relațiile de afiliere. Cheile externe utilizează în mod eficient valorile atributelor în relația menționată pentru a restricționa domeniul unuia sau mai multor atribute în relația referință. O cheie externă poate fi descrisă în mod oficial ca: "Pentru toate tupluri din relația de referință, proiectate peste atributele la care fac trimitere, trebuie să existe un tuplu în relația de referință proiectat peste aceleași atribute, astfel încât valorile din fiecare atribut de referință să se potrivească cu valorile corespunzătoare în atributele la care fac trimitere. [10]

O procedură de stocare este cod executabil care este asociat cu baza de date, și în general stocat în baza de date. De obicei, procedurile de stocare colectează și personalizează operațiunile comune, cum ar fi introducerea unui tuplu într-o relație, colectarea informațiilor statistice despre patenurile de utilizare sau încapsularea calculelor și a metodologiei complexe. Frecvent, acestea sunt folosite ca o interfață de programare a aplicațiilor (API) pentru siguranță sau pentru simplitate. Implementările procedurilor de stocare în SQL SGBDR permit adesea dezvoltatorilor să profite de extensiile procedurale (de multe ori specifice furnizorilor) în ceea ce privește sintaxa standard SQL declarativă. Procedurile stocate nu sunt parte a modelului baze de date relationale, dar toate implementările comerciale le include. [10]

Un index este o modalitate de a oferi acces mai rapid la date. Indicii pot fi creați prin orice combinație de atribute dintr-o relație. Interogările care fac filtrarea folosind acele atribute pot găsi în mod aleatoriu tupluri potrivite folosind indicele, fără să fie necesar să se verifice fiecare tuplu, la rândul său. Acest lucru este analog cu folosirea cuprinsului unei cărți pentru a merge direct la pagina în care se găsește informația pe care o cauți, astfel încât să nu fie necesar să citești întreaga carte pentru a găsi informația căutată. În mod curent, bazele de date relaționale furnizează tehnici multiple de indexare, fiecare dintre aceste tehnici fiind optimă pentru o anumită combinație privind distribuția datelor, dimensiunea relației și modelul tipic de acces. De obicei, indicii sunt puși în aplicare prin dezvoltări de tip B+, R-trees și bitmap. Indicii nu sunt de obicei considerați parte a bazei de date, ci sunt considerați un detaliu de implementare, chiar dacă indicii sunt susținuți de obicei de către același grup care susține și celelalte părți ale bazei de date. Trebuie precizat faptul că utilizarea indicilor eficienți atât prin chei primare, cât și chei străine pot îmbunătăți în mod semnificativ performanța interogării. Acest lucru se datorează faptului că indicii B-tree conduc la proporționale ca timp cu $\log(n)$, unde n este numărul rândului dintr-un tabel și indicii zvon conduc la interogări în timp constant (nu depind de dimensiune, atâta timp cât partea relevantă a indicelui se potrivește în memorie). [10]

1.2.7. Operații relaționale

Interogările efectuate în baza de date relațională și derivările în baza de date sunt exprimate în calcul relațional sau în relații algebrice. În algebra originală relațională, Codd a introdus opt operatori relaționali, grupați în două grupe de patru operatori fiecare. Primii patru operatori au fost bazați pe bazați operațiunile matematice tradiționale:

- Operatorul reuniune combină tuplurile din două relații și elimină toate tupluri duplicate din rezultat. Operatorul reuniune relațional este echivalent cu operatorul SQL UNION.
- Operatorul intersecție indică un set de tupluri pe care două relații le au în comun. Intersecție este implementată în SQL sub forma operatorului INTERSECT.
- Operatorul diferență acționează pe două relații și indică un set de tupluri din prima relație care nu există în a doua relație. Operatorul diferență este implementat în SQL sub forma operatorului EXCEPT sau MINUS.
- Produsul cartezian a două relații este o îmbinare care nu este limitată de nici un criteriu, rezultând în unirea fiecărui tuplu al primei relații cu fiecare tuplu a celei de-a doua relație. Produsul cartezian este implementat în SQL prin intermediul operatorului CROSS JOIN. [10]

Restul operatorilor propuși de Codd implică operațiuni speciale specifice bazelor de date relaționale:

- Selecția sau restricția, operațiune ce preia tupluri dintr-o relație, limitând rezultatele la acelea care îndeplinesc un criteriu specific, de exemplu, un subansamblu al seriei teoretice. Echivalentul în SQL este SELECT cu o clauză WHERE.
- Operațiunea de proiecție - extrage doar atributele specificate dintr-un tuplu sau un set de tupluri.
- Operațiunea de îmbinare definită pentru bazele de date relaționale face referire adesea la o îmbinare naturală. În acest tip de alăturare, două relații sunt legate prin atributele lor comune. În SQL, INNER JOIN împiedică apariția produsului cartezian atunci când există două tabele într-o interogare. Pentru fiecare tabel adăugat la o interogare SQL, un INNER JOIN suplimentar este adăugat pentru a preveni produsul cartezian. Astfel, pentru N tabele într-o interogare SQL, trebuie să existe N-1 operatori INNER JOIN pentru a preveni un produs cartezian.
- Operația de împărțire relațională este o operațiune ceva mai complexă, care în esență implică folosirea tuplurilor unei relații (dividendul) pentru a împărți a doua relație (divizorul). Operatorul de împărțire relațională este efectiv opusul operatorului produsului cartezian (de unde și numele). [10]

Au fost introduși sau propuși ulterior și alți operatori în afara celor opt originali introduși de Codd, printre alții, inclusiv operatorii de comparație relațională și extensii care oferă suport pentru a clusteriza și ierarhiza datele.

1.3. Baze de date distribuite

1.3.1. Definiții

Putem defini o bază de date de distribuție (DDB) ca o colecție de mai multe baze de date interdependente, logic distribuite, într-o rețea de calculatoare și un sistem de gestiune de baze de date distribuite (SGBDD) ca un sistem software care gestionează o bază de date distribuită făcând transparentă distribuția pentru utilizator. [7]

Pentru ca o bază de date să fie numită distribuită, trebuie să fie îndeplinite cel puțin următoarele condiții:

- **Conectarea nodurilor bazei de date în rețeaua de calculatoare.** Există mai multe computere, numite **situri** sau **noduri**. Aceste situri trebuie să fie conectate printr-o **rețea de comunicație** adiacentă pentru a transmite date și comenzi între situri;
- **Asocieri logice ale bazelor de date conectate.** Este esențial ca informațiile din bazele de date să fie conectate în mod logic;
- **Lipsa restricțiilor omogene între nodurile conectate.** Nu este necesar ca toate nodurile să fie identice din punct de vedere al datelor, hardware și software; [7]

1.3.2. Avantaje

Unele dintre avantajele importante sunt enumerate mai jos:

1. Facilități îmbunătățite și flexibilitate în dezvoltarea aplicației. Dezvoltarea și menținerea aplicațiilor pe siturile distribuite geografic ale unei organizații este facilitată datorită transparenței distribuției bazei de date și controlului.
2. Fiabilitate ridicată și disponibilitate. Acest lucru este obținut prin izolarea erorilor în locul lor de origine, fără a afecta celelalte baze de date conectate la rețea. În cazul în care datele și software-ul SGBDD sunt distribuite pe mai multe situri, unul dintre situri poate eșua, în timp ce celelalte situri vor continua să funcționeze. Numai datele și software-ul care se află în situl în care a avut loc eroarea nu poate fi accesat. Acest lucru îmbunătățește atât fiabilitatea, cât și disponibilitatea. O îmbunătățire suplimentară este realizată prin replicarea în mod judicios a datelor și software-ul în mai multe situri. Într-un sistem centralizat, o eroare într-un singur sit face ca întregul sistem să nu mai fie disponibil tuturor utilizatorilor. Într-o bază de date distribuită, chiar dacă unele date nu pot fi accesate, utilizatorii pot avea în continuare posibilitatea de a accesa alte părți ale bazei de date. În cazul în care

datele din situl care a eşuat au fost replicate într-un alt sit înainte de eroare, atunci utilizatorul nu va fi afectat deloc.

3. Performanță îmbunătățită. Un SGBD distribuit fragmentează datele prin menținerea acestora mai aproape de locul unde este nevoie cel mai mult de ele. Localizarea datelor reduce disputa pentru CPU și I / O și în mod simultan reduce întârzierile de acces apărute în zona de rețelele largi. Dacă o bază de date mare este distribuit pe mai multe situri, atunci baze de date mai mici există în fiecare sit. Prin urmare, interogările locale și tranzacțiile care accesează datele dintr-un singur sit au o performanță mai bună, determinată de bazele de date locale mai mici. În plus, fiecare sit are un număr mai mic de tranzacții în executare, decât dacă toate tranzacțiile ar fi transmise către o singură bază de date centralizată. În plus, paralelismul inter și intra-interogări poate fi realizat prin executarea mai multor interogări în diferitele situri, sau prin splitarea unei interogări într-un număr de subinterogări care se execută în paralel. Acest fapt contribuie la îmbunătățirea performanței.

4. Dezvoltarea mai ușoară. Într-un mediu distribuit, este permisă extinderea sistemului în ceea ce privește adăugarea de noi date, crescând dimensiunea bazei de date, iar adăugarea mai multor procesoare este mult mai ușoară. [7]

1.3.3. Funcții adiționale

Distribuția conduce la complexitate crescută în ceea ce privește designul și punerea în aplicare a sistemului. Pentru a obține avantajele potențiale enumerate anterior, software-ul SGBDD trebuie să poată furniza următoarele funcții, în plus față de cele ale unui SGBD centralizat:

- **Urmărirea distribuției datelor.** Capacitatea de a urmări distribuția datelor, fragmentarea și replicarea prin extinderea catalogului SGBDD.
- **Procesarea distribuită a interogării.** Capacitatea de a accesa situri izolate și de a transmite interogări și date între diferitele situri printr-o rețea de comunicații.
- **Gestiunea tranzacțiilor distribuite.** Capacitatea de a elabora strategii de execuție pentru interogările și tranzacțiile care accesează date din mai multe situri și de a sincroniza accesul la datele distribuite, menținând integritatea bazei de date globale.
- **Gestiunea multiplicării datelor.** Capacitatea de a decide pe care copie al unui element din datele multiplicat să îl acceseze și abilitatea de a menține coerența copiilor obținute prin multiplicarea unui element.
- **Recuperarea bazei de date distribuite.** Capacitatea de recuperare după accidente individuale ale sitului și de noile tipuri de erori, cum ar fi eroarea legăturilor de comunicare.
- **Securitate.** Tranzacțiile distribuite trebuie să fie executate cu buna gestionare a securității datelor și autorizarea / dreptul de acces ale utilizatorilor.

- **Gestiunea registrului (catalogului) distribuit.** Un director conține informații (metadate) despre datele incluse în baza de date. Directorul poate fi global pentru întregul DDB sau local pentru fiecare sit în parte. Plasarea și distribuirea directorul reprezintă probleme de proiectare și de tehnică. [7]

Aceste funcții cresc prin ele însele complexitatea unui SGBDD în raport cu un SGBD centralizat. Înainte de a putea înțelege pe deplin avantajele potențiale ale distribuției, trebuie să găsim soluții satisfăcătoare pentru aceste controverse și probleme de proiectare. Incluzând toate aceste aspecte suplimentare, funcționalitatea este dificil de realizat, iar găsirea unei soluții optime este un pas înainte. [7]

1.4. Baze de date NoSQL

NoSQL cuprinde o varietate largă de tehnici de baze de date care au fost dezvoltate ca răspuns la creșterea volumului de informații stocate despre beneficiari, obiecte și produse, a frecvenței cu care aceste date sunt accesate, performanța și procesarea necesităților. Pe de altă parte, bazele de date relaționale nu au fost concepute pentru a face față provocărilor în ceea ce privește dimensiunea și agilitatea cu care se confruntă aplicațiile moderne, nici nu au fost create pentru a profita de stocarea ieftină și puterea de procesare disponibile astăzi.

1.4.1. Tipuri de baze de date NoSQL

- **Bazele de date document** cuplează fiecare cheie cu o structură de date complexă, cunoscută sub denumirea de document. Documentele pot conține mai multe perechi de cheie-valoare diferite sau perechi de cheie-matrice sau chiar documente în serie.
- **Depozite de grafice** sunt utilizate pentru a stoca informații despre rețele, cum ar fi conexiunile sociale. Magazinele de grafice includ Neo4J și HyperGraphDB.
- **Depozite cheie-valoare** sunt cele mai simple baze de date NoSQL. Fiecare element unic din baza de date este stocat ca un nume atribut (sau "cheie"), împreună cu valoarea sa. Exemple de magazine cheie-valoare sunt Riak și Voldemort. Unele magazine cheie-valoare, cum ar fi Redis, permite fiecărei valori să aibă o formă, cum ar fi "întreg", ceea ce adaugă funcționalitate.
- **Depozite Wide-column**, cum ar fi Cassandra și HBase sunt optimizate pentru interogări asupra seturilor de date de mari dimensiuni și magazinelor de date coloane la un loc, în loc de rânduri. [12]

1.4.2. Avantaje

În comparație cu bazele de date relaționale, bazele de date NoSQL sunt mai accesibile și oferă performanțe superioare, iar modelul lor de date abordează o serie de aspecte pe care modelul relațional nu este conceput pentru a le aborda:

- volume mari de date structurate, semi-structurate și nestructurate
- sprinturi agile, repetare rapidă
- programare orientată-obiect, care este ușor de utilizat și flexibilă
- arhitectura eficientă, orientată spre exterior, în locul arhitecturii scumpe, monolite. [12]

Scheme dinamice

Bazele de date relaționale cer ca schemele să fie definite înainte de a putea adăuga date. De exemplu, ați putea dori să stocați date despre clienții dumneavoastră, cum ar fi numerele de telefon, numele și prenumele, adresa, localitatea și țara- o bază de date SQL trebuie să știe în prealabil ceea ce stocați.

Acest lucru nu se potrivește bine cu abordările de dezvoltare rapide întrucât de fiecare dată când completezi noi caracteristici, de multe ori, schema bazei trebuie să se schimbe. Așadar dacă decideți în privința iterațiilor în dezvoltare că doriți să stocați și elementele preferate ale clienților în plus față adresele și numerele de telefon, va trebui să adăugați acea coloană la baza de date și apoi migrați întreaga bază de date pe noua schemă. [12]

Dacă baza de date este mare, acesta este un proces foarte lent ceea ce implică nefuncționalități semnificative. Dacă schimbați frecvent datele aplicațiile stocate- pentru a face iterări rapide – aceste perioade de oprire poate fi, de asemenea, frecvente. De asemenea, nu există nicio modalitate de a accesa în mod eficient date care sunt complet nestructurate sau necunoscute în prealabil, folosind o bază de date relațională.

Bazele de date NoSQL sunt construite pentru a permite inserția de date fără o schemă predefinită. Acest fapt determină ușurința de a face schimbări semnificative în aplicații în timp real, fără preocupări privind întreruperile - ceea ce înseamnă că dezvoltarea este mai rapidă, integrarea codurilor oferă mai multă încredere și este nevoie de un timp mai scurt pentru administrarea bazelor de date. [12]

1.4.3. NoSQL vs SQL

	Baze de date SQL	Baze de date NOSQL
Tipuri	Un singur tip (baze de date SQL) cu variații minore	Mai multe tipuri diferite, incluzând depozite de cheie-valoare, baze de date de documente, depozite coloane largi și baze de date grafice.
Dezvoltarea cronologică	Dezvoltate în 1970 cu scopul de a face față primului val de aplicații de stocare a datelor.	Dezvoltate în anii 2000 pentru a face față cu neajunsurilor bazelor de date SQL, în special cu privire la dimensiunea, replicarea și stocarea datelor nestructurate.
Exemple	MySQL, Postgres, Oracle Database	MariaDB, MongoDB, Cassandra, HBase, Neo4j
Scheme	Structura și tipul datelor sunt stabilite în prealabil. Pentru a stoca informații despre o nouă caracteristică a datelor, întreaga bază de date trebuie să fie modificată, timp în care baza de date trebuie să fie deconectată.	De obicei dinamice. Se pot adăuga înregistrări privind noi informații din mers și spre deosebire de rândurile tabelor SQL, dacă este necesar, pot fi stocate împreună date diferite. Pentru anumite baze de date (de exemplu, stocările de coloane largi), este oarecum mai important să adăugi noi domenii în mod dinamic.
Scalare	Verticală, ceea ce înseamnă că un singur server trebuie să fie făcut din ce în ce mai puternic pentru a face față cerințelor în creștere. Este posibilă răspândirea bazei de date SQL de-a lungul a mai multor servere, dar, în general, este necesară tehnică suplimentară semnificativă.	Orizontală, ceea ce înseamnă că pentru a adăuga obiecte, un administrator de baze de date poate pur și simplu să adăuge mai multe servere sau instanțe cloud. Baza de date extinde automat datele peste toate serverele, după cum este necesar.
Dezvoltarea Modelului	Mix de surse deschise (de exemplu, Postgres, MySQL) și surse închise (de exemplu, Oracle Database)	Open-source
Susținerea operațiunilor	Da, actualizările pot fi configurate pentru a se completa în întregime sau deloc	În anumite circumstanțe și la anumite niveluri (de exemplu, nivelul document vs. nivelul bază de date)
Manipularea datelor	Limbaj specific, folosind Select, Insert, și situații de actualizare, de exemplu, SELECT fields FROM table WHERE...	Prin API-uri obiect-orientate.

Tabel 1.2 NoSQL vs SQL [12]

Capitolul 2. Conceptul MariaDB Cluster

MariaDB se folosește de MariaDB Galera Cluster pentru implementarea clusterului. Acesta va fi discutat în detaliu la punctul 2.2. În continuare vom afla ce reprezintă un cluster și care sunt arhitecturile acestuia.

2.1. Conceptul de clustering

Utilizatorii implementează, adesea, bazele de date în configurații cluster pentru a îndeplini cerințele de business precum scalabilitatea, performanța, disponibilitate ridicată și recuperarea în cazul unui eșec. Când se implementează un cluster, arhitectura SGBD care stă la bază, este un important factor de luat în considerare. Calculul Cluster este „un grup de calculatoare interconectate, ce lucrează împreună, așa că din multe se formează un singur calculator”. În limbajul folosit în bazele de date, clustering înseamnă că o aplicație vede o singură bază de date, în timp ce în spate stau de fapt două sau mai multe calculatoare ce procesează datele. Pe lângă asigurarea scalabilității, clusterelor bazei de date pot să furnizeze beneficii adiționale precum încărcarea cumpănată și disponibilitate ridicată, dar aceste lucruri nu sunt moștenite în toate clusterelor bazei de date. [13]

2.2. Arhitecturile unui cluster

Cele două arhitecturi SGBD predominante sunt shared-nothing și shared-disk.

2.2.1. Shared - Nothing

Shared - nothing lucrează pe principiul că fiecare nod din cluster are dreptul de proprietate exclusivă a datelor din nodul respectiv. Fiecare nod nu distribuie date cu alte noduri din cluster, de aici și termenul de shared-nothing. Când te muți dintr-un singur server în servere multiple, în clusterul shared-nothing trebuie să împarți datele serverelor. Acest proces de împărțire a datelor la servere, precum este prezentat în figură de dedesubt, este numit partiționare. Datele pot fi partiționate vertical sau orizontal. [13]

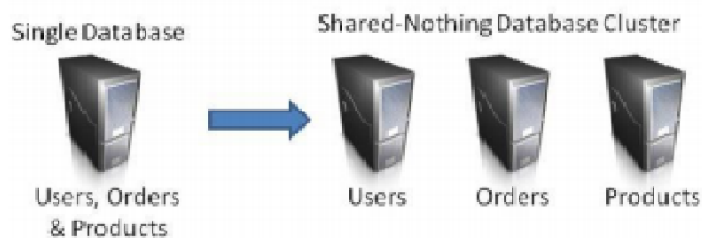


Figura 2.1 Shared-Nothing (partiționarea datelor între noduri) [13]

Cererile de date sunt apoi procesate printr-o tabelă de rutare ce rutează fiecare cerere către server/nod ce are respectivele date. De exemplu, Serverul 1 de deasupra poate să aibă informație despre useri, în timp ce Server 2 poate să aibă informație despre comenzi. Dacă aplicația dumneavoastră face o cerere ce implică ambele servere, de exemplu cererea unei liste de useri și informație despre comenzi pentru comenzile ce au avut loc luna anterioară, trebuie să implicați ambele servere. Baza de date ar putea rezolva această cerere prin citirea listei de comenzi ce au avut loc luna anterioară și apoi prin trimiterea listei de la Server 2 la Server 1 pentru a adăuga informație despre useri. Acest proces de trimitere a datelor de la un server la altul este denumit data shipping. La nucleul său, o bază de date de tip shared-nothing este o bază de date independentă cu facilități adăugate de data shipping. [13]

Fail-over și configurația Master- Slave.

Dacă un server va cădea, toate aplicațiile care necesită acces la serverul căzut, vor cădea deasemenea și ele. Din acest motiv, fiecare nod are nevoie de o copie identică care să poată ține locul în cazul în care nodul primar va cădea. Acest nod este numit și slave. În configurația master-slave, master-ul poate face operația de citire și updatare, în timp ce slave-ul poate asigura doar accesul de citire la copia locală a datelor slave-ului. [13]

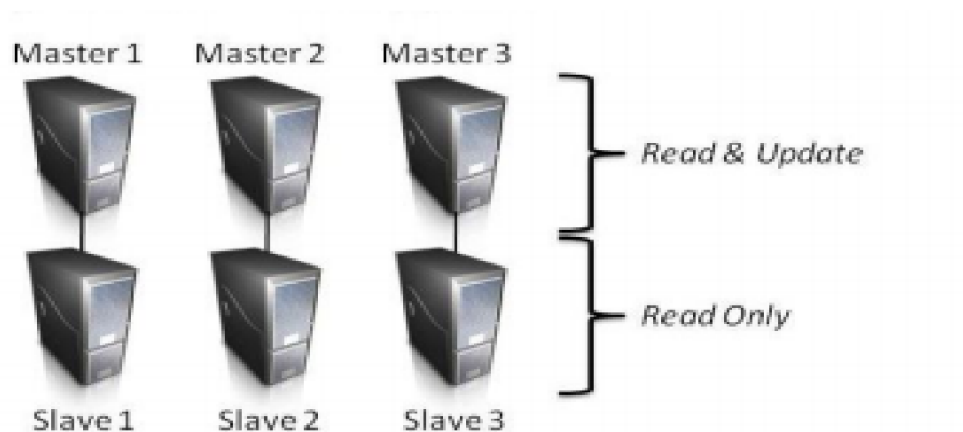


Figura 2.2 Configurație Master – Slave [13]

Pentru aplicațiile cu o rată ridicată de citiri vs. update-uri, slave-urile pot fi configurate într-o configurație de tip arbore, unde datele de la master sunt replicate în arbore, și apoi slave-ul descarcă accesul pentru citire. [13]

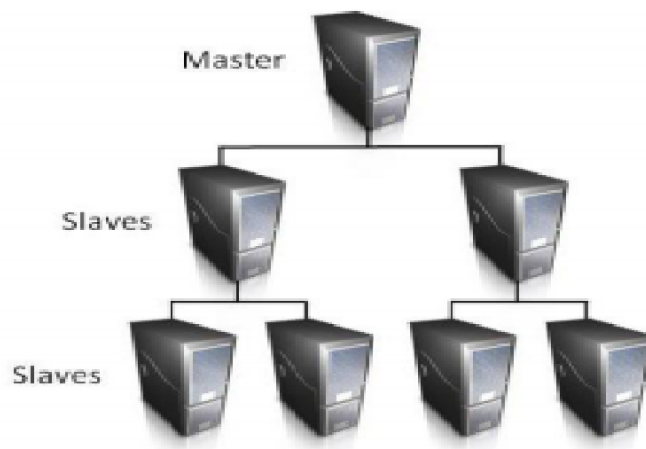


Figura 2.3 Arbore de Slave [13]

Nu se poate face nimic pentru a împiedica un shared-disk de la a utiliza un arbore similar, doar cu acces la citire, de tip slave, dar slave-urile sunt în general asociate cu shared-nothing.

Exemple de Shared –Nothing SGBD: Oracle 11g, IBM DB/2 (fără să fie de tip cadru principal), Sybase ASE, MySQL (InnoDB, MyISAM, Falcon...toate, cu excepția ScaleDB, fiind motoare de stocare), Microsoft SQL, Server, etc.

2.2.2. Shared – Disk

Shared – disk, deasemenea cunoscut ca și shared- everything, funcționează pe principiul că dumneavoastră aveți o listă a disk-urilor, tipică unei rețele de tip Storage Area Network (SAN) sau Network Attached Storage (NAS), aceste tipuri de rețele ținând toate datele în baze de date. Fiecare server sau nod în cluster se comportă ca o singură colecție de date în timp-real. [13]

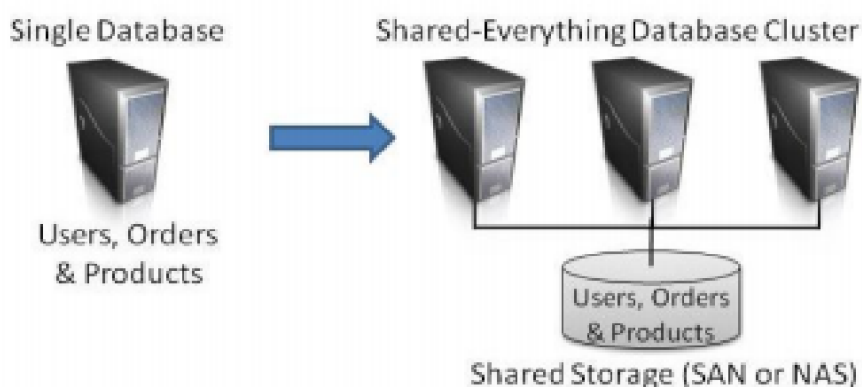


Figura 2.4 Cluster Shared- Disk (toate nodurile au acces la toate datele) [13]

Într-o arhitectură de tip shared-disk fiecare nod poate îndeplini orice cerere, deoarece toate nodurile au acces la toate datele. De aceea, în loc de a merge la un nod specific unei anumite date, shared-disk poate ruta pur și simplu cererea la următorul nod disponibil. Deoarece fiecare nod poate adresa cerere oricărei baze de date, încărcarea este preponderentă de-a lungul nodurilor din cluster. [13]

Fail-Over, Încărcarea Preponderată și configurarea modului Master-Master:

În timp ce modul shared-nothing are numai un singur nod ce poate update orice parte din date în baza de date, modul shared-disk activează orice nod să updateze în baza de date. Pentru acest motiv, shared-disk este numit arhitectură master-master. Deasemenea, el utilizează la maxim serverul din cluster. Această arhitectură asigură, deasemenea, o moștenire fail-over, în timp ce fiecare nod acționează ca un back-up la fiecare alt nod. În sfârșit, arhitectura shared-disk asigură o moștenire load-balancing, de vreme ce fiecare cerere la baza de date poate fi trimisă în orice server disponibil.

Totuși nimic nu vine pe gratis. Pentru a activa flexibilitatea, nodurile comunică între ele pentru a-și coordona activitatea, în special: blocarea, statutul și bufferingul. Acest schimb de mesaje internodal crează un anumit grad de depășire, dar dumneavoastră trebuie să controlați această depășire împotriva impactului datelor/funcției de transport găsite în clusterelor shared-nothing. [13]

Exemple de Shared-Disk SGBD: ScaleDB (MySQL), Oracle RAC, Sybase ASE CE, IBM IMS și DB/2 (mainframe).

Care arhitectură este mai bună?: Din păcate, o așa simplă întrebare nu are un simplu răspuns. Fiecare arhitectură are părți bune și părți mai puțin bune. În multe cazuri, comparând aceste două arhitecturi este precum a compara transmisiile automobilelor. Shared disk este analog cu transmisia automată, deoarece automatizează mult din complexitatea de întreținere și set-up, acestea scăzând din costul total al proprietarului (TCO). Shared-nothing este analog cu transmisia manuală deoarece asigură un control mai granular în schimburi pentru un efort manual crescut din partea proprietarului. Așa cum șoferii tind să fie pasionați de transmisia lor preferată, DBAs tind să fie pasionate în privința arhitecturii preferate de baze de date. [14]

2.3. MariaDB Galera

Clusterul MariaDB Galera este un cluster bază de date de tip sincron multi-master, bazat pe replicarea sincronă și MySQL/InnoDB a lui Oracle. Când clusterul Galera este în folosință, dumneavoastră puteți citi și scrie direct în orice nod și puteți pierde orice nod individual fără întreruperi în operații și fără să fie nevoie să vă descurcați cu proceduri complexe de fail-over. [14]

La un nivel înalt, clusterul Galera este alcătuit dintr-un server de baza de date - ce este MySQL, MariaDB sau Percona XtraDB. Serverul folosește apoi Plugin de replicare Galera. Ca să fim mai specifici API-ul plugin-ului de replicare MySQL a fost extins pentru a asigura toate informațiile și legăturile necesare pentru o adevărată replicare sincronă multi-master. Acest API extins este denumit API-ul de replicare Write-Set, sau API-ul wsrep.

Prin API-ul wsrep, clusterul Galera asigură o certificare pentru replicarea de bază. O tranzacție pentru replicare, și anume write-set: nu numai că ea conține rândurile bazei de date pentru replicare, dar deasemenea, include informații despre toate barierele impuse de baza de date, în timpul tranziției. Fiecare nod certifică apoi replicarea write-set împotriva altor write-set-uri ce se află în coadă pentru aplicare. Write-set-ul este apoi aplicat dacă nu se găsesc niciun conflict de barieră. În acest punct tranziția este considerată dusă la bun sfârșit, după care fiecare nod continuă să facă aplicările în tabela de spații.

Această abordare este deasemenea numită și replicarea virtuală sincronă, dat fiind că, deși este logic sincron, scrierea actuală și ducerea la bun sfârșit în tabela de spațiu are loc independent și cele nesincrone au loc pe fiecare nod. [14]

2.3.1 Avantaje

Clusterul Galera asigură o îmbunătățire semnificativă de disponibilitate ridicată pentru MySQL ecosystem. Diferitele căi de atingere a disponibilității de nivel ridicat au asigurat, tipic, doar unele

dintre caracteristicile disponibile prin clusterul Galera, făcând alegerea unei soluții de disponibilitate cu nivel ridicat un exercițiu de trade-off.

Următoarele caracteristici sunt disponibile prin clusterul Galera:

- Un adevărat Multi- Master de citire și scriere în orice moment și la orice nod;
- Replicarea sincronă No Slave Lag: nicio dată nu este pierdută la o cădere a unui nod;
- O cuplare strânsă a tuturor nodurilor ce mențin aceeași stare. Nicio deviere de date între noduri este permisă;
- Multi-threaded Slave pentru o mai bună performanță. Pentru orice volum de lucru;
- Nicio operație Failover Master-Slave sau folosire a VIP;
- Hot Standby No downtime în timpul failover-ului (de vreme ce nu este nici un failover);
- Un automat Node Provisioning. Nu este nevoie de un back-up manual pentru baza de date și nu este nevoie de o copie a ei într-un alt nod;
- Suportă InnoDB;
- Transparență la aplicațiile cerute (sau minimum): nici o schimbare la aplicație;
- Nicio necesitate de despărțire a procesului de Citire și Scriere.

Ca rezultat al unei soluții de disponibilitate ridicat, ce este și robustă în termeni și în integritatea datelor dar și de o performanță ridicată cu failover-uri instantanee. [14]

Implementarea de Cloud cu clusterul Galera

Un beneficiu adițional al clusterului Galera este un bun suport de cloud. Modul automat de asigurare a nodurilor face ca operațiile de scale-in și scale-out să fie fără probleme. Clusterul Galera s-a dovedit a se descurca destul de bine în cloud, precum atunci când folosește instanțare multiplă de noduri mici de-a lungul centrelor de date – zone AWS de exemplu, sau chiar deasupra rețelei Wider Area. [14]

2.3.2 Arhitectura

Api-ul de replicare

Sistemele sincrone de replicare folosesc replicarea eager. Nodurile din cluster se sincronizează cu celelate noduri prin actualizarea replicărilor în timpul tranziției. Ceea ce înseamnă că atunci când o tranziție este dusă la bun sfârșit, toate nodurile au aceeași valoare. Procesul are loc folosind replicarea write-set prin grupul de comunicații. [14]

Arhitectura internă a clusterului Galera se învârtă în jurul a patru componente:

- Sistemul de gestiune al bazei de date (SGBD) - Serverul bazei de date ce rulează pe un nod individual. Clusterul Galera poate folosi MySQL, MariaDB sau Percona XtraDB.
- API-ul wsrep – Interfața și responsabilitățile pentru serverul bazei de date și furnizorul replicării. Este alcătuit din:

- Cârligul wsrep: Integrarea cu motorul serverului bazei de date pentru replicarea write-set.
- dloperm(): Funcția ce face furnizorul wsrep disponibil în cârligul wsrep.

- Plugin-ul de replicare Galera: Plugin-ul ce activează funcționalitatea serviciului de replicare.
- Plugin-urile Group Communication: Diverse sisteme de grup de comunicare disponibile clusterului Galera. De exemplu Gcomm și Spread. [14]

Api-ul wsrep

Api-ul wsrep folosește un model de replicare ce consideră că serverul bazei de date are o anumită stare. Starea se referă la conținutul bazei de date. Când o bază de date este în folosință, clienții modifică conținutul bazei de date, acest lucru schimbând starea. Api-ul wsrep reprezintă schimbările în starea bazei de date ca o serie de schimbări atomice, sau tranziții.

În clusterul bazei de date, toate nodurile au aceeași stare întotdeauna. Ele se sincronizează între ele prin replicare și prin aplicarea schimbărilor de stare în aceeași ordine serială. [14]

Dintr-o perspectivă mai tehnică, clusterul Galera suportă schimbări de stare în următoarele procese:

1. La un nod din cluster, o schimbare de stare are loc în baza de date;
2. În baza de date, cârligul wsrep translatează schimbările în write-set;
3. dlopem() face disponibil furnizorul funcției wsrep în cârligul wsrep;
4. Plugin-ul de replicare Galera suportă certificare și replicare write-set în cluster.

Pentru fiecare nod în cluster, procesul aplicației are loc la tranziții de prioritate ridicată. [14]

Tranziția Globală ID

Pentru a ține starea identică de-a lungul clusterului, Api-ul wsrep folosește Tranziția Globală ID, sau GTID. Aceasta presupune a identifica schimbările de stare și a identifica starea curentă în relație cu ultima schimbare de stare.

Tranziția Globală ID este alcătuită din următoarele componente:

- Starea UUID : Un identificator unic pentru stare și pentru secvența de schimbări ce urmează;
- Ordinal Sequence Number: Secvența formată din un integr signed de 64-bits folosit pentru a arăta poziția schimbării în secvență.

Tranziția Globală ID vă permite să comparați starea aplicației și a stabili ordinea schimbărilor de stare. Puteți să o folosiți pentru a determina dacă o schimbare a fost sau nu aplicată și dacă schimbarea este aplicabilă la toate stările date. [14]

Plugin-ul de replicare Galera

Plugin-ul de replicare Galera implementează API-ul wsrep operează precum furnizorul wsrep. Dintr-o perspectivă mai tehnică, plugin-ul de replicare Galera este alcătuit din următoarele componente:

- Certification Layer: Acest strat pregătește write-set-urile și face verificările de certificare asupra lor, asigurând faptul că ele pot fi aplicate.
- Replication Layer: Acest strat gestionează protocolul de replicare și asigură capacitatea totală de comandă.
- Group Communication Framework.: Acest strat asigură plugin-ul de arhitectură pentru diverse sisteme de grupuri de comunicări ce se conectează la clusterul Galera. [14]

Group Communication Plugins(Plugin-urile grupului de comunicații)

Framework-ul grupului de comunicații asigură o arhitectură de plug-in pentru diverse sisteme Gcomm.

Clusterul Galera este construit deasupra unui strat de sistem de grup proprietar de comunicație, care implementează un sincron QoS virtual. Un sincron virtual unește datele livrate și serviciile membre clusterului, asigurând un formalism clar pentru semanticele de livrare a mesajului.

În timp ce un sincron virtual garantează consistența, el nu garantează un sincron temporal, ce este necesar pentru operații fine de multi-master. Pentru a se apropia de acest lucru clusterul galera implementează în al său control de debit runtime-configurabil temporal. Controlul de debit menține nodurile sincronizate la o fracțiune de secundă.

În plus, Framework-ul grupului de comunicație asigură, deasemenea, o comandă totală de mesaje pentru surse multiple. Folosește acest lucru pentru a genera Global Transaction ID's într-un cluster multi-master.

La nivelul de transport, Clusterul Galera este un graf neorientat simetric. Toate nodurile bazei de date se conectează unul la celălalt peste conexiunea TCP. În mod implicit TCP este folosit atât pentru replicarea mesajului cât și pentru serviciile membre clusterului, dar dumneavoastră puteți deasemenea să folosiți multicast-ul UDP pentru replicările într-un LAN. [14]

2.3.3 Nivele de izolare

Clusterul Galera se ocupă cu tranzițiile în izolare. Aceste nivele izolate garantează că nodurile procesează tranziții într-un mod fiabil.

Izolarea garantează că tranzițiile concomitent funcționale nu interferează una cu cealaltă. Din acest motiv asigură, deasemenea și consistența datelor. Dacă tranzițiile nu au fost izolate, o tranziție poate modifica datele pe care alte tranziții le citesc, ceea ce ar putea duce la inconsistența datelor. Clusterul Galera implică patru nivele, care sunt într-o ordine crescătoare: [14]

- CITIRE NEÎNREGISTRATE
- CITIRE ÎNREGISTRATE
- CITIRE REPETABILĂ
- SERIALIZABILITATE

CITIRE ÎNREGISTRATE

Aici tranzițiile pot vedea schimbările făcute asupra datelor de către alte tranziții, care nu au fost încă implicate. Cu alte cuvinte, tranzițiile pot citi datele, ce eventual nu mai pot exista, dat fiind că alte tranziții pot întotdeauna să reinițializeze schimbările, fără implicare. Acest proces este cunoscut ca și „citirea murdară”.Efectiv CITIREA – NEIMPLICATA nu are deloc o izolare reală. [14]

CITIRE NEÎNREGISTRATE

Aici „citirile murdare” nu sunt posibile. Schimbările neimplicate rămân invizibile altor tranziții până când tranziția se implică.Totuși, la acest nivel de izolare, interogările SELECT folosesc propriile instantanee de date implicate. Acestea sunt datele implicate înainte în interogarea SELECT executată. Ca un rezultat, interogările SELECT când rulează de multe ori în aceeași tranziție, pot da diferite seturi de rezultate. Acest lucru este numit și „citirea nerepetabilă”. [14]

CITIREA REPETABILĂ

Aici citirile nerepetabile nu sunt posibile. Instantaneele luate pentru interogarea SELECT sunt luate prima dată când interogarea SELECT rulează în timpul tranziției. Instantaneul rămâne în folosire pe întreaga tranziție pentru interogarea SELECT. Întotdeauna returnează același set de rezultate. Acest nivel nu duce la schimbări de cont la date de către alte tranziții, indiferent dacă au fost sau nu comise. În acest fel, citirile rămân repetabile. [14]

SERIALIZAREA

Aici toate înregistrările accesate în tranziții sunt blocate. Resursele se blochează într-un mod ce, deasemenea vă împiedică de la anexarea înregistrării la tabela de tranziții ce a operat înainte. SERIALIZAREA previne un fenomen cunoscut ca și „citirea fantomă”. „Citirile fantomă” au loc, când într-o tranziție două interogări identice se execut, și rezultatele celei de-a doua interogări se întorc diferite față de prima.

Clusterul Galera folosește izolarea tranziției atât local cât și la nivelul clusterului. [14]

Izolare locală a tranziției

Izolarea tranziției are loc la fiecare nod la nivelul local al serverului bazei de date. Funcționează la fel ca și cu motorul de stocare nativ InnoDB.Toate cele patru nivele sunt disponibile. Setarea de bază pentru izolarea locală a tranziției este de tip Citire –Repetabilă. [14]

Izolarea Tranziției Clusterului

Izolarea tranziției are loc deasemenea, la nivelul clusterului, între tranzițiile ce sunt procesate pe diferite noduri, Clusterul Galera implementează un nivel de tranziție denumit IZOLARE –

INSTANTANEE. Nivelul Izolării- Instantanee are loc între CITIREA –REPETABILĂ și SERIALIZABILITATE. [14]

Motivul pentru aceasta este că nu este niciun suport în izolarea tranziției în nivelul de SERIALIZABILITATE pentru cazul folosirii a multi-master, și nici în STATEMENT și nici în formatele ROW. Acest lucru este datorat faptului că Plugin-ul de replicare Galera nu transportă un read-set de tranziție. Deasemenea, datorită faptului că izolarea tranziției la nivelul de SERIALIZABILITATE este vulnerabilă la conflicte multi-master. Au loc blocări în citire și orice scriere replicată la un rând de citire blocat cauzează abandonarea tranziției. Este recomandat ca dumneavoastră să evitați folosirea SERIALIZABILITĂȚII în clusterul Galera. [14]

2.4 Funcționarea unui cluster MariaDB Galer

Principala concentrare este consistența datelor. Tranzițiile sunt aplicate fie pe fiecare nod sau pe niciunul. Așadar, bazele de date rămân sincronizate, asigurând ca ele să fie configurate corespunzător și sincronizate la început.

Plugin-ul de replicare Galera diferă față de replicarea standard MariaDB, prin adresarea unor diferite probleme, incluzând conflictele de scriere multi-master, lag-ul de replicare și slave-urile încep fără sincronizare cu master-ul. [14]

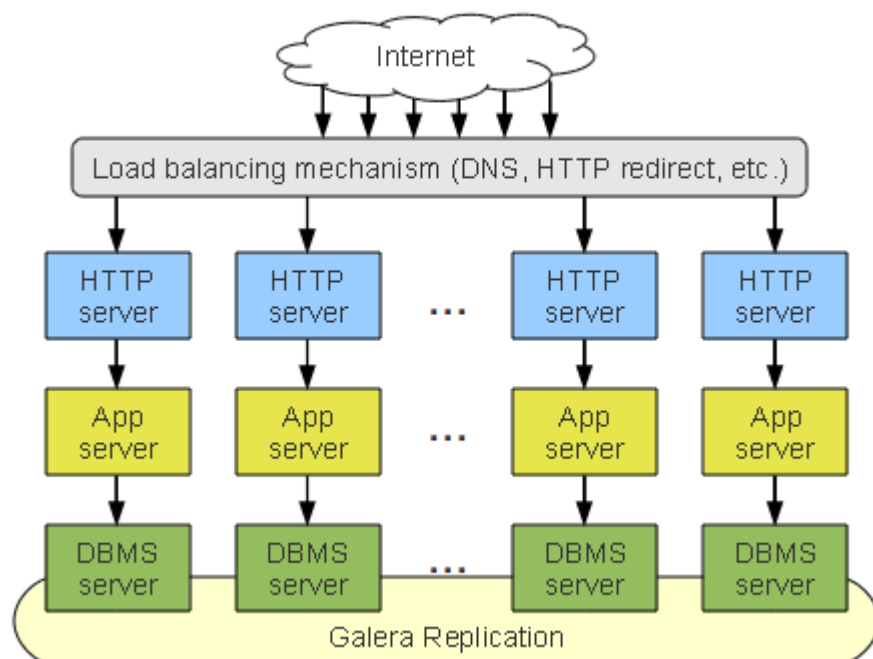


Figura 2.5 Funcționarea unui cluster MariaDB Galera [14]

Într-o instanță tipică a Clusterului Galera, aplicațiile pot scrie în orice nod din cluster și tranziția va avea loc. Apoi sunt aplicate la toate serverele, printr-o replicare certificată de bază. Replicarea

certificată de bază este o abordare alternativă către replicarea sincronă a bazei de date, folosind comunicație grp și comandând tehnici de tranziție.

Configurarea unui cluster MariaDB Galera este prezentată în capitolul 4, în descrierea aplicației. [14]

2.5. Limitări

Vom enumera câteva limitări importante ale clusterului MariaDB Galera:

- Windows nu este suportat;
- Replicarea curentă lucrează doar cu motorul de stocare InnoDB. Orice scrieri în tabelele de alte tipuri, incluzând tabele de system (MariaDB *), nu sunt replicate (această limitare exclude declarația DDL precum și CREATE USER, ce modifică implicit tabelele MariaDB *–acestea fiind replicate). Deasemenea este un suport experimental pentru MyISAM--vedeți replicarea wsrep a variabilei de sistem MyISAM!.
- Blocarea nesuportată explicit include LOCK TABLES, PUSH TABLES { lista explicită de tabele } cu READ_LOCK, (GET_LOCK(), RELEASE_lock(...)). Folosind tranziții corespunzător, ar trebui să fie capabil să depășească aceste limitări. Operatorii globali de locking precum FLUSH TABLES WITH READ LOCK sunt suportați.
- Toate tabelele ar trebui să aibă cheie primară (cheile primare multi-column sunt suportate). Operatorii nu sunt suportați pe tabelă fără o cheie primară. Deasemenea, rândurile în tabele fără o cheie primară pot apărea într-o ordine diferită pe noduri diferite.
- Jurnalul de interogare nu poate fi alocat tabelei. Dacă dumneavoastră activați jurnalul de interogare, trebuie să trimiteți mai departe jurnalul către un fișier: log_output=FILE.
- Tranzacțiile XA nu sunt suportate.
- Mărimea tranziției. În timp ce Galera nu limitează explicit mărimea tranziției, un write-set este procesat ca un singur buffer memory-resident. Ca un rezultat, tranziții extrem de largi (e.g. LOAD DATA) pot afecta în mod advers performanța nodului. Pentru a evita acest lucru variabilele de sistem wsrep_max_ws_rows și wsrep_max_ws_size limitează rândurile de tranziție la 128k și mărimea tranziției la 1GB, în mod implicit. Dacă e necesar, utilizatorii ar putea dori să mărească aceste limite. Versiuni viitoare vor adăuga suport pentru fragmentarea tranziției.”[15]

Capitolul 3 Tehnologii folosite

3.1 Limbaje

3.1.1 Limbajul HTML

Majoritatea aplicațiilor care citesc și scriu în fișiere folosesc un format specific. Aceste fișiere conțin informațiile necesare pentru a reconstrui documentul de fiecare dată când sunt deschise, care este conținutul documentului și metadatele despre document, cum ar fi autorul, ultima dată când a fost modificat, sau chiar o listă de schimbări ce permite navigarea între versiuni. [1]

HTML(HyperText Markup Language) este un limbaj ce descrie conținutul unei pagini web. Folosește elemente de sintaxă speciale denumite etichete (ce reprezintă elemente) pentru a încorpora text în interiorul lor, ce indică clienților cum ar trebui să interpreteze acea porțiune a documentului.

Un client este un program care accesează paginile web pentru un utilizator. Aici trebuie făcută o

distincție. Toate browser-ele pentru desktop (Internet Explorer, Opera, Firefox, Safari, Chrome) sau destinate smartphone-urilor (Opera Mini, WebKit) reprezintă clienți, dar nu toți clienții sunt browsere. Programele automate folosite de motoarele de căutare cum ar fi Google, sunt și ele de asemenea clienți, deși ele nu sunt controlate de un operator uman în mod direct. [1]

Browser-ele web citesc documentele HTML și le redau într-o formă vizuală sau auditivă. Browser-erele nu afișează direct etichetele HTML, în schimb le folosesc pentru a interpreta conținutul paginilor.

Elementele HTML reprezintă structurile de bază pentru construirea oricărei pagini web. Limbajul permite includerea de obiecte și imagini pentru a crea pagini interactive, și o metodă de a crea documente structurate, reflectând structura textului prin intermediul unor elemente specifice cum ar fi titluri, paragrafe, liste, link-uri, citate. Poate să includă și script-uri scrise în alte limbaje cum ar fi JavaScript, care pot modifica comportamentul paginii HTML.[2]

Elementele de bază în HTML sunt reprezentate de două etichete ce conțin o porțiune de text, și în majoritatea cazurilor pot conține și elemente copil. Există și excepții de la această regulă însă, unele elemente cum ar fi `img`, nu conțin nici text nici copii.

Elementele pot avea de asemenea și atribute, acestea putând să modifice comportamentul elementului și să adauge mai multe funcționalități. [1]

În cadrul limbajului HTML există două tipuri de elemente ce corespund unor structuri specifice: `block` și `inline`. [1]

Elementele de tip `block` corespund unui nivel superior și de obicei denotă structura documentului. Pot fi considerate ca începând de pe o linie nouă, separând conținutul de elementele anterioare. Elemente de tip `block` uzuale sunt listele, paragrafele și tabelele.

Elementele de tip `inline` sunt conținute în cele de tip `block` înglobând doar o mică porțiune de text, nu grupuri mari de text sau paragrafe. Un element `inline` nu va genera o nouă linie, fiind elemente care apar de obicei în cadrul unui paragraf. Elemente de tip `inline` des întâlnite sunt link-urile sau citatele. [1]

De asemenea limbajul HTML conține și caractere speciale. Acestea sunt `<`, `>`, `&`. Ele sunt folosite pentru a marca începutul sau sfârșitul elementelor HTML și nu caracterele mai mici, mai mare și ampersand. [1]

3.1.2 Java EE

Java EE este platforma Oracle enterprise. Această platformă oferă un API și un mediu pentru dezvoltarea și rularea aplicațiilor enterprise, ce includ servicii web și aplicații de rețea la scară largă, pe mai multe nivele, scalabile și securizate. Java EE extinde Java Platform Standard Edition, oferind API-uri pentru maparea relațiilor pe obiecte, arhitecturi distribuite și multinivel și servicii web. Platforma are un design bazat în principal pe componente modulare ce rulează într-un server de aplicație. Produsele Java EE sunt dezvoltate în principal folosind limbajul Java. Platforma încurajează convenția înaintea configurării, și folosirea de adnotări pentru configurare. Opțional se poate folosi XML pentru a supradefini adnotările sau pentru a devia de la configurarea standard. [3]

Java EE este definită pe baza specificațiilor.

Java EE include câteva specificații de API-uri, cum ar fi RMI, email, JMS, servicii web, XML care definesc modul lor de coordonare. Java EE are de asemenea specificații pentru componente unice pentru această platformă. Printre acestea se numără Enterprise JavaBeans, conectori, servlet - uri, JavaServer Pages și câteva tehnologii pentru servicii web. Acestea permit dezvoltarea unor aplicații enterprise care sunt scalabile și portabile, și care se pot integra ușor cu aplicații mai vechi. Un server de aplicație Java EE se va ocupa de tranzacții, securitate, scalabilitate, concurență și gestiunea componentelor care sunt încărcate, pentru a le permite dezvoltatorilor să se concentreze pe logica componentelor și nu pe infrastructură și integrare. [3]

API – urile de bază:

- javax.servlet.* - specificațiile pentru servlet. Include un set de specificații pentru gestiunea cererilor HTTP și specificațiile pentru JSP
- javax.websocket.* - specificațiile pentru conexiune pe baza unui websocket
- javax.faces.* - pachetul conține specificații de bază pentru JSF(JavaServer Faces). JSF este utilizat pentru a construi interfețe folosind componente
- javax.faces.component.* - definește partea de componente pentru JSF., fiind unul din pachetele de bază
- javax.el.* - definește clasele și interfețele folosite de Expression Language
- javax.enterprise.inject.* - definește adnotările pentru injectare din cadrul API – urilor Context and Dependency Injection
- javax.enterprise.context.* - definește adnotările de context din cadrul Context and Dependency Injection
- javax.ejb.* - pachetul conține clasele și interfețele care definesc contractele între EJB și client și cele dintre EJB și container
- javax.validation.* - conține adnotări și interfețe pentru validare declarativă
- javax.transaction.* - acest pachet conține contractele dintre clasele gestionate și JPA(Java Persistence API)
- javax.security.auth.message.* - oferă JTA(Java Transaction API) și conține clasele și interfețele necesare pentru a interacționa cu capacitățile de tranzacționare oferite de Java EE
- javax.enterprise.concurrent.* - conține clasele și interfețele necesare pentru a construi module de autentificare securizate
- javax.jms.* - oferă o modalitate generală prin care programele Java pot crea, trimite, primi și citi mesaje
- javax.batch.api.* - folosită în general pentru acțiuni ce pot necesita volume mari de date
- javax.resource.* - folosit pentru a conecta servere de aplicații cu sisteme de informație de nivel enterprise [3]

3.1.3 JavaScript

Javascript cunoscut și ca ECMAScript este un limbaj de programare dinamic. Este cel mai des folosit ca parte a browser-elor web ale căror implementări permit scripturilor să interacționeze cu utilizatorul, să controleze browser-ul, să comunice asincron și să modifice conținutul documentului. [4]

JavaScript este clasificat ca fiind un limbaj de scripting bazat pe prototipuri și cu tipuri dinamice. Aceasta combinație de caracteristici îl fac să fie un limbaj cu multiple paradigme, permițând

programarea orientată obiect, imperativă și funcțională.

Sintaxa de JavaScript este derivată din cea de C, semantică fiind influențată de limbajele de programare Self și Scheme.

JavaScript poate fi parte și a unor aplicații care nu sunt web, cum ar fi documente pdf și aplicații desktop. Pe partea de client JavaScript a fost implementat tradițional ca limbaj interpretat, deși browser-erele moderne pot efectua compilări just-in-time.

JavaScript a fost standardizat în standardul ECMAScript. [4]

JavaScript suportă majoritatea elementelor specifice programării structurale, sintaxa fiind asemănătoare cu cea a limbajului C (structuri alternative, repetitive, iterative). O diferență este scopul variabilelor. Inițial scopul unei variabile era în cadrul funcției în care a fost declarată. În ECMAScript 6 folosind cuvântul cheie `let` se poate limita scopul unei variabile la nivelul blocului în care a fost declarată. O altă diferență față de C este inserarea automată; în cazul în care aceasta nu a fost adăugată deja. [4]

Ca și în alte limbaje de scripting tipurile sunt asociate cu valori și nu cu variabile.

JavaScript este aproape în întregime orientat obiect. Obiectele în JavaScript pot fi considerate liste asociative, completate de prototipuri. Numele proprietăților sunt reprezentate de valori de tip șir de caractere. Proprietățile pot fi adăugate, modificate și șterse la rulare. Majoritatea proprietăților unui obiect pot fi enumerate folosind construcția `for ...in`. JavaScript conține și un număr de obiecte predefinite cum ar fi `Date` sau `Function`. [4]

JavaScript poate folosi funcția `eval` pentru a rula expresii sub formă de string-uri la momentul rulării.

Funcțiile în JavaScript reprezintă obiecte, având proprietăți și metode precum `.call()` sau `.bind()`. O funcție poate fi definită în interiorul altei funcții. Va fi creată de fiecare dată când este apelată funcția în care este definită. În plus fiecare funcție are asociat un scop, dar are acces și la elementele din scopul funcției părinte, chiar și după ce execuția acesteia s-a terminat. În JavaScript este permisă folosirea funcțiilor anonime. [4]

În JavaScript se folosesc prototipurile pentru moștenire. Este posibil să se folosească prototipurile pentru a simula multe din capacitățile claselor și ale moștenirii.

Funcțiile pot fi folosite drept constructori folosind cuvântul cheie `new`. În acest mod se crează o nouă instanță a prototipului, moștenind proprietățile de la constructor. JavaScript oferă prototipuri implementate cum ar fi `Array` sau `Object`. Este posibilă modificarea prototipului `Object`, dar în general este de evitat deoarece poate cauza modificarea comportamentului tuturor obiectelor care îl moștenesc.

Spre diferență de alte limbaje în JavaScript nu este nici o diferență între definirea unei funcții și definirea unei metode. Diferența apare doar când se invocă funcția ca metodă a unui obiect `this` va corespunde obiectului a cărui funcție a fost invocată. [4]

3.1.4 CSS

CSS(Cascading Style Sheet) este un limbaj folosit pentru a descrie aspectul și formatarea unui document scris într-un limbaj de etichete. Deși de obicei este folosit pentru a schimba stilul unei pagini web scrisă în HTML sau XHTML, el poate fi aplicat și pentru orice tip de XML. Împreună cu HTML și JavaScript este folosit pentru a crea interfețe web pentru aplicații web sau mobile. [5]

CSS este creat pentru a permite separarea conținutului unui document de prezentare. Această separare îmbunătățește accesul la conținut, oferă mai multă flexibilitate și control asupra prezentării, permite mai multor pagini HTML să utilizeze aceeași formatare prin utilizarea unui fișier extern cu extensia .css, și reduce complexitatea și repetitivitatea conținutului structurat. CSS face posibilă separarea de instrucțiunile HTML prin includerea elementelor proprii într-un fișier extern sau în secțiunea style a documentului. [5]

Această separare permite de asemenea prezentarea aceluiași conținut în diferite forme, cum ar fi pe ecran, vocal și pe dispozitive ce afișează în Braille. Poate fi folosit și pentru a afișa pagina web diferit în funcție de rezoluția ecranului sau dispozitivul pe care este vizualizată. Un alt avantaj al CSS este faptul că se poate modifica stilul unui sau mai multor documente, ușor și rapid prin editarea câtorva linii de cod în locul procesului laborios de a verifica linie cu linie documentul HTML. [5]

Specificațiile CSS conțin o schemă de priorități pentru a determina ce stil se aplică în cazul în care mai multe stiluri corespund aceluiași element.

Specificațiile de CSS sunt gestionate de World Wide Web Consortium(W3C) .Internet media type text/css este înregistrat pentru a fi folosit pentru CSS în RFC 2318. [5]

Sintaxa CSS este simplă și utilizează cuvinte din vocabularul limbii engleze pentru numele diferitelor proprietăți.

Conținutul este reprezentat de o listă de reguli. Fiecare regulă are unul sau mai mulți selectori și un bloc declarativ. [5]

În CSS selectorii sunt folosiți pentru a determina pe ce etichete se aplică, verificând structura elementelor și atributele lor din cadrul documentului. [5]

Selectorii se pot aplica pentru:

- toate elementele de un anumit tip
- elementele specificate pe baza unui atribut
- id: un identificator unic la nivelul documentului
- class: un identificator ce grupează mai multe elemente în cadrul documentului
- elementele pe baza poziției în arborele DOM [5]

Blocul declarativ este reprezentat de o listă de declarații delimitate de o pereche de acolade. Fiecare declarație conține o proprietate și o valoare.

Proprietățile sunt specificate în fiecare standard CSS. Fiecare proprietate are un set specific de valori.

Unul din scopurile CSS este acela de a oferi utilizatorilor un control mai mare asupra prezentării. În funcție de browser și de website utilizatorul poate alege dintre mai multe seturi oferite de designeri, sau le poate dezactiva pe toate, utilizând stilurile de bază specifice browser-ului. [5]

3.2 Framework-uri

3.2.1 Spring

În 1996 limbajul Java era încă la început și mulți dezvoltatori au ales să folosească acest limbaj datorită aplicațiilor web ce puteau fi dezvoltate cu ajutorul applet – urilor. Spre diferență de alte limbaje în Java se poate crea o aplicație separată în mai multe părți de dimensiuni mici. [6]

În decembrie acel an Sun Microsystems a publicat specificațiile pentru JavaBeans 1.0. Acestea defineau un model de componente pentru Java. Aceste specificații defineau un set de reguli ce permiteau obiectelor Java să fie ușor re folosibile în cadrul aplicațiilor complexe. [6]

Deși au fost intenționate pentru uz general, aplicațiile complexe necesită o serie de servicii, suport pentru tranzacții, securitate, etc care nu sunt oferite de specificațiile de JavaBeans. Astfel în 1998 Sun Microsystems a publicat specificațiile pentru EJB 1.0. Acestea includ și serviciile necesare dezvoltării aplicațiilor enterprise, dar le lipsește simplitatea bean – urilor originale.

În ziua de astăzi, dezvoltarea în Java s-a întors înspre origini. Noi tehnici de programare, cum ar fi programare orientată pe aspecte, și inversiunea controlului oferă JavaBeans multe din capacitățile rezervate până acum doar pentru EJB. [6]

Spring este un framework open-source, creat de către Rod Johnson și descris în cartea sa Expert One-on-One: Design and Development. Spring a fost creat pentru a adresa complexitatea dezvoltării aplicațiilor enterprise și permite utilizarea de JavaBeans pentru lucruri ce ar fi fost posibile doar folosind EJB. Dar utilitatea Spring – ului nu este limitată doar la partea de server-side. Orice aplicație poate beneficia prin folosirea Spring în termeni de simplitate, scalabilitate, testare și a legării slabe între componente. [6]

Misiunea fundamentală pentru Spring este aceea de a simplifica dezvoltarea în Java. Pentru a îndeplini acest obiectiv se folosește de patru strategii:

- obiecte simple și dezvoltare cât mai puțin invazivă
- legare slabă prin injectare de dependențe și interfețe pentru convenții
- programare declarativă prin folosirea de aspecte și convenții uzuale
- reducerea codului redundant prin folosirea aspectelor și a modelelor

Aproape toate lucrurile în Spring pot fi clasificate într-una din aceste strategii. [6]

Într-o aplicație Spring, contextul aplicației încarcă definițiile claselor și crează legăturile dintre ele. Contextul aplicației este responsabil în întregime de instanțierea obiectelor ce alcătuiesc aplicația și legăturile dintre ele. Spring are mai multe implementări pentru contextele de aplicație, care diferă doar prin modul de încărcare al obiectelor. [6]

Deși injectarea dependențelor face posibilă legarea slabă a componentelor, programarea orientată pe aspecte permite încorporarea funcționalității în componente folosite în cadrul aplicației.

Programarea orientată pe aspecte este o tehnică ce promovează separarea responsabilităților într-un sistem software. Sistemele sunt compuse din mai multe componente, fiecare componentă având o responsabilitate specifică. De multe ori aceste elemente au responsabilități în plus față de funcționalitățile de bază. Serviciile oferite de sistem cum ar fi securitatea, gestiunea tranzacțiilor interacționează cu componente care au cu totul alte responsabilități. Acestea sunt denumite cross-cutting concerns, deoarece interacționează cu numeroase componente din cadrul sistemului.

Adaugând aceste componente se introduc două nivele de complexitate în cadrul codului.

- codul care implementează componenta este duplicat în cadrul mai multor componente
- componentele conțin cod care nu are legătură cu funcționalitatea lor de bază

Programarea orientată pe aspecte face posibilă modularizarea acestor servicii și aplicarea lor declarativă asupra componentelor care ar trebui afectate. În acest mod se obțin componente coezive care se concentrează doar asupra funcționalității de bază, fără să țină seama de cum interacționează cu alte servicii. [6]

Spring 3 introduce Spring Expression Language(SpEL), o metodă succintă de a seta valorile obiectelor prin setteri sau constructorii, prin folosirea de expresii ce sunt evaluate la rulare. Folosind SpEL se pot obține rezultate, care folosind modul tradițional de inițializare al obiectelor ar fi imposibil.

Printre caracteristicile SpEL se numără:

- posibilitatea de a identifica obiecte pe baza unui id
- invocarea metodelor unui obiect și accesarea proprietăților
- operații matematice, relaționale și logice aplicate valorilor
- expresii regulate
- utilizarea colecțiilor [6]

Scopul final al SpEL este acela de a transforma o expresie într-o valoare după evaluare. Pe parcursul evaluării se pot folosi și alte valori. [6]

Spring oferă posibilitatea de a identifica tipul obiectelor automat în cazul în care acesta nu a fost specificat. Spring folosește patru metode de identificare a obiectelor:

- după nume – încearcă să compare toate proprietățile unui obiect pentru a verifica dacă definiția obiectului conține acele proprietăți
- după tip - compară obiectele pe baza valorilor ce sunt introduse ca parametrii
- constructor – caută un constructor care primește ca parametrii valorile specificate
- autodetecție – compară întâi după constructor, apoi dacă nu a găsit un răspuns încearcă după tip [6]

3.2.2 Hibernate

Hibernate este un framework Java folosit pentru a mapa un domeniu orientat obiect pe o bază

de date relațională.

Principala caracteristică a lui Hibernate este maparea unor clase Java pe tabele din baza de date (și conversia tipurilor de date Java în valori SQL). Hibernate oferă metode de a selecta datele din baza de date. Generează apeluri de SQL și elimină nevoia de a converti manual datele. Aplicațiile ce folosesc Hibernate sunt portabile de pe o bază de date pe alta cu condiția să fie suportată de Hibernate.

Maparea claselor pe tabele se realizează folosind un fișier de configurare XML sau prin adnotări Java. Când se folosește XML pentru configurare, Hibernate poate genera cod sursă pentru clasele mapate. Acest lucru nu se mai întâmplă dacă se folosesc adnotări. Schema bazei de date este menținută folosindu-se XML sau adnotări. [16]

Facilități pentru a mapa relații de tipul one-to-many sau many-to-many sunt oferite. Hibernate oferă de asemenea posibilitatea de a mapa relații reflexive, când un obiect are o relație de tipul one-to-many cu instanțe de același tip. [16]

Hibernate poate mapa și alte valori decât cele standard. Acest lucru face posibile următoarele:

- suprascrierea tipurilor care ar fi mapate automat de către Hibernate
- posibilitatea de a mapa obiecte de tip enum pe coloane
- maparea unei singure proprietăți pe mai multe coloane

Hibernate oferă un limbaj inspirat de SQL denumit HQL (Hibernate Query Language), pentru a rula comenzi asemănătoare cu cele de SQL asupra obiectelor mapate de Hibernate. [16]

Hibernate oferă o metodă transparentă de mapare a obiectelor pe bază de date. Singura necesitate este aceea ca obiectele să conțină un constructor fără argumente, nu neapărat public. De asemenea poate fi de dorit suprascrierea metodelor equals() și hashCode(). [16]

Colecțiile de obiecte sunt stocate în colecții java de tip List sau Set. Hibernate suportă Java Generics și poate fi configurat pentru lazy loading asupra colecțiilor de obiecte. Începând cu Hibernate 3 lazy loading este setarea standard. [16]

Obiectele legate între ele prin relații pot fi configurate să aplice modificările efectuate asupra lor în cascadă.

Componentele Hibernate:

- Hibernate ORM – software – ul de bază pentru maparea obiectelor pe baze de date relaționale
- Hibernate Annotations – adnotările folosite pentru transformările efectuate asupra datelor în timpul mapării
- Hibernate EntityManager – un wrapper ce oferă o soluție conform java Persistence API folosindu-se de funcționalitățile de bază ale Hibernate
- Hibernate Envers – auditarea și versionarea claselor de persistență
- Hibernate OGM – o extensie pentru lucrul cu baze de date NoSQL
- Hibernate Shards – partiționare orizontală pentru multiple baze de date [16]

3.3 Librării

3.3.1 JQuery

JQuery este o bibliotecă JavaScript folosită pentru a simplifica dezvoltarea scripturilor HTML. Este cea mai populară bibliotecă de JavaScript la momentul actual, fiind folosită în cadrul a mai mult de 60% din primele un milion cele mai vizitate site-uri. [17]

Sintaxa de JQuery este gândită pentru a ușura navigarea în document, selectarea unor elemente DOM, crearea de animații, gestiunea de evenimente și construirea de aplicații AJAX. JQuery oferă programatorilor posibilitatea de a extinde această librărie prin crearea de plugin-uri. [17]

Microsoft și Nokia au încorporat JQuery în platformele lor. Microsoft l-a inclus în Visual Studio pentru a fi folosit împreună cu framework-ul ASP.NET AJAX și în cadrul platformei MVC. Nokia l-a integrat în platforma Web Run-Time.

Jquery este la baza o bibliotecă pentru manipularea DOM. DOM este o structură arborescentă a tuturor elementelor dintr-o pagină web, iar JQuery simplifică căutarea, selectarea și manipularea acestor elemente. De exemplu JQuery poate fi folosit pentru a căuta un element cu o anumită proprietate, pentru a-i modifica unul sau mai multe atribute și pentru al face să interacționeze cu un eveniment. [17]

Jquery oferă o nouă paradigmă pentru gestiunea evenimentelor. Asignarea evenimentului și definirea funcției de callback se fac acum într-un singur pas în același loc, în cod. JQuery țintește la a încorpora funcționalități des folosite în JavaScript prin manipularea proprietăților CSS. [17]

Avantajele folosirii JQuery sunt

- încurajează separarea codului JavaScript de HTML – în JQuery este foarte simplă adăugarea de evenimente a unor elemente DOM folosind JavaScript, existând astfel încorporarea lor în atribute HTML. În acest fel face posibilă separarea completa a codului de JavaScript de HTML
- clariate – JQuery promovează claritatea folosind funcții ce pot fi înălțuite și cu nume scurte
- eliminarea incompatibilităților între browser-e – motoarele de JavaScript de pe diferite browser-e au anumite diferențe, însă JavaScript oferă o interfață care funcționează la fel pe orice browser
- extensibilitate - JQuery este foarte ușor de extins, noi elemente și metode putând fi adăugate și apoi refolosite ca un plugin [17]

JQuery oferă următoarele funcționalități

- selectarea elementelor DOM utilizând motorul de selectare multi-browser Sizzle
- manipularea elementelor DOM bazată pe selectori CSS care folosesc numele și atributele elementelor, cum ar fi id sau class, drept criterii de căutare a nodurilor DOM
- evenimente
- efecte și animații
- AJAX
- obiecte de tip deferred și promise pentru controlul operațiilor asincrone
- parsare de JSON
- extensibilitate prin plugin-uri

- utilități ca detectarea funcționalităților
- metode pentru compatibilitate (deși sunt prezente în toate browser-ele moderne, lipsesc din unele mai vechi)
- suport multi-browser [17]

3.4 Instrumente de asamblare a aplicației

3.4.1 Maven

Maven este un instrument pentru automatizarea asamblării, folosit în principal în proiectele Java. Maven adresează două aspecte ale construirii unui program software. În primul rând descrie cum este construit software-ul, iar în al doilea rând descrie dependențele. Spre diferență de alte instrumente ca Apache Ant, folosește convenții pentru proceduri și numai excepțiile trebuie scrise într-un XML. Fișierul XML conține informații despre cum a fost construit proiectul, dependențele față de module și componente externe, ordinea construirii proiectului și plugin-uri necesare. Conține setări predefinite pentru anumite procese, cum ar fi compilarea și împachetarea codului. Maven încarcă dinamic librării Java și le salvează într-un cache local.. Acesta poate fi completat cu propriile proiecte. [18]

Maven poate fi folosit și pentru construcția și gestiunea proiectelor scrise în C#, Ruby, Scala sau alte limbaje. Maven este găzduit de Apache Software Foundation, și a fost parte din proiectul Jakarta. [18]

Maven a fost construit folosind o arhitectură bazată pe plugin-uri care îi permite să utilizeze aplicații controlabile prin input-ul standard. În teorie acest lucru permite crearea de plugin-uri pentru a interacționa cu instrumente pentru alte limbaje, dar suportul pentru alte limbaje în afară de Java a fost minimal. [18]

POM (Project Object Model) oferă toată configurarea pentru un singur proiect, Configurarea de bază conține numele proiectului, cel care deține proiectul și dependențele către alte proiecte. Se pot configura faze individuale din construcția unui proiect, care sunt implementate sub formă de plugin-uri.

Proiectele de dimensiuni mari ar trebui împărțite în mai multe module, fiecare cu câte un POM. Se poate adăuga un POM rădăcină care poate compila toate modulele cu o singură comandă. POM-urile mostenesc configurația de la alte POM-uri. Toate POM-urile mostenesc Super POM. [18]

Cea mai mare parte a funcționalității Maven este în plugin-uri.

Există plugin-uri Maven pentru compilare, testare, gestiunea codului sursă, rularea unui server web, generarea de proiecte Eclipse, și multe altele. Plugin-urile sunt introduse și configurate în secțiunea <plugins> a pom.xml. Unele plugin-uri de bază sunt incluse automat în proiect și au setări ce nu trebuie modificate. [18]

O funcționalitate de bază în Maven este gestiunea dependențelor. Mecanismul de gestiune a dependențelor este organizat pe baza unui sistem de coordonate care identifică artefacte cum ar fi librării și module. Maven 2 Central Standard este setat standard pentru căutarea librăriilor, dar poate fi setat pentru a căuta și în alte zone. [18]

3.5 Medii de dezvoltare

3.5.1 Eclipse

Eclipse este un mediu de dezvoltare. Conține un spațiu de lucru de bază și un sistem extensibil de customizare bazat pe plugin-uri. A fost dezvoltat în principal în Java. Pe lângă Java în Eclipse se pot dezvolta și programe în alte limbaje: Ada, ABAP, C, C++, COBOL, Fortran, Haskell, JavaScript, Lasso, Lua, Natural, Perl, PHP, Prolog, Python, R, Ruby, Scala, Clojure, Groovy, Scheme și Erlang. [19]

Eclipse folosește plugin-uri pentru toate funcționalitățile. Mediul de rulare este Equinox, o implementare de platformă OSGI. [19]

Pe lângă posibilitatea de a extinde platforma Eclipse cu alte limbaje cum ar fi C sau Python, framework-ul de plugin-uri permite extinderea IDE-ului cu aplicații de rețea precum telnet, sau sisteme de gestiune a bazelor de date. [19]

Eclipse SDK, care este inclus automat include Eclipse Java development Tools (JDT) un compilator Java incremental și un model pentru fișierele sursă. [19]

Eclipse oferă Rich Client Platform care include următoarele elemente:

- Equinox OSGi – framework standard
- Platforma de bază – pornește Eclipse, rulează plugin-uri
- Standard Widget Tool(SWT) – set de widget-uri portabil
- JFace – ajută la implementarea modelui MVC pentru SWT
- Eclipse Workbench – editoare, perspective, wizard-uri [19]

Exemple de aplicații Rich Client Platform dezvoltate cu Eclipse sunt:

- IBM Notes 8 și 9
- Novell/NetIQ Designer
- Apache Directory Studio
- Remote Component Environment [19]

Eclipse oferă suport pentru dezvoltare pentru Tomcat, GlassFish și multe alte servere, fiind capabil să instaleze serverul necesar direct din IDE. Oferă funcționalitate de debugging , permițând utilizatorului să vizualizeze variabilele când aplicația rulează cu un server atașat. [19]

Eclipse conține și Web Tools Platform(WTP) folosit pentru dezvoltarea aplicațiilor web și a celor de Java EE. Include editoare text și grafice pentru numeroase limbaje, wizard-uri și aplicații standard pentru a simplifica dezvoltarea, instrumente și API-uri pentru rularea și testarea aplicațiilor. [19]

Capitolul 4. Descrierea aplicației

Aplicația reprezintă un site web de management și alocare a asset-urilor într-o firmă IT. În această aplicație angajații se pot loga și își pot vedea asset-urile alocate, pot face cereri pentru a primi noi asset-uri sau pot renunța la cele curente.

Totodată, ei mai au acces și la următoarele operații:

- semnalarea problemelor apărute la utilizarea asseturilor;

- vizualizarea cererilor noi sau a celor deja procesate;
- vizualizarea tranzacțiilor în curs de desfășurare sau a celor încheiate;
- vizualizarea plângerilor noi sau a celor rezolvate;

Pentru procesarea cererilor, tranzacțiilor și a plângerilor depuse va fi nevoie de un administrator. Acesta va vedea toate acțiunile făcute de angajați și se va ocupa de rezolvarea lor.

Aplicația este construită pe arhitectura MVC (Model-View-Controller).

Model-view-controller este un model arhitectural utilizat în ingineria software. Succesul modelului se datorează izolării logicii de business față de considerentele interfeței cu utilizatorul, rezultând o aplicație unde aspectul vizual sau/și nivelele inferioare ale regulilor de business sunt mai ușor de modificat, fără a afecta alte nivele. [20]

- **Model**

Această parte a controlatorului manipulează operațiunile logice și de utilizare de informație (trimisă dinainte de către rangul său superior) pentru a rezulta de o formă ușor de înțeles. [20]

- **View**

Acestui membru al familiei îi corespunde reprezentarea grafică, sau mai bine zis, exprimarea ultimei forme a datelor: interfața grafică ce interacționează cu utilizatorul final. Rolul său este de a evidenția informația obținută până ce ea ajunge la controlator. [20]

- **Controller**

Cu acest element putem controla accesul la aplicația noastră. Pot fi fișiere, scripts sau programe, în general orice tip de informație permisă de interfață. În acest fel putem diversifica conținutul nostru de o formă dinamică și statică, în același timp. [20]

4.1. Instalarea programelor necesare aplicației

În continuare vom vedea de ce programe avem nevoie pentru a realiza și rula această aplicație. De asemenea, vor fi prezentate și instrucțiunile de creare a bazei de date și structura acesteia.

4.1.1. Instalare MariaDB Galera Server

Galera nu este suportată pe sistemul de operare Windows.

Pentru instalarea pe Linux (Ubuntu Release 14.04, versiunea 10.0), mai întâi trebuie să adăugăm cheile pentru magazia Galera rulând următoarele comenzi în terminal:

```
sudo apt-get install software-properties-common
```

```
sudo apt-key adv --recv-keys --keyserver hkp://keyserver.ubuntu.com:80  
0xc9cb082a1bb943db
```

Adăugăm magazia pentru Ubuntu:

```
sudo add-apt-repository 'deb http://mirror.stshosting.co.uk/mariadb/repo/10.0/ubuntu  
trusty main'
```

După ce cheile au fost importate și magazia a fost adăugată, vom instala programele MariaDB, Galera, și Rsync:

```
sudo apt-get update
```

```
sudo apt-get install -y rsync
```

```
sudo apt-get install mariadb-galera-server
```

4.1.2. Instalare Java

Pentru descărcarea acestui program se accesează link-ul următor:

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

În această pagină, cautați rubrica **Java SE Development Kit 8u45** (versiunea actuală a programului la momentul redactării acestui document), bifați **Accept License Agreement** și selectați din opțiuni versiunea sistemului dumneavoastră de operare.

Se deschide fișierul descărcat și urmați pașii indicați pentru a finaliza instalarea.

Pentru ca sistemul de operare să poată localiza fișierele executabile Java este nevoie să adăugăm calea acestuia în variabilele de mediu **JAVA_HOME** și **Path**.

WINDOWS:

- Intrați în Control Panel -> System -> Advanced system settings;
- In josul ferestrei, apăsați pe **Environment Variables**;
- Sub **System Variables**, apăsați pe **New**;

- Introduceți ca nume “**JAVA_HOME**”, iar ca valoare inserați calea către folderul de Java (ex: **C:\Program Files\Java\jdk1.8.0_45**) și apoi apăsați **OK**;
- Căutați variabila **Path** și, în meniul de edit, după ultima valoare din câmpul Value adăugați **;%JAVA_HOME%\bin**.
- Verificați că variabilele au fost setate corect introducand comanda “**java -version**” în terminal;

Linux:

- Editați fișierul de pornire (~/.bashrc);
- export JAVA_HOME = folderul de Java;
- export PATH = \$JAVA_HOME/bin:\$PATH;
- Salvați și închideți fișierul;
- Verificați că variabilele au fost setate corect introducand comanda “**% java -version**” în terminal;

4.1.3. Instalare Eclipse

Pentru descărcarea acestui program se accesează link-ul următor:
<http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplersr2>

Se selectează din partea dreaptă a paginii sistemul de operare, apoi se apasă pe butonul **Download** pentru a începe descărcarea programului. După ce descărcarea s-a finalizat, nu este nevoie de instalare, doar accesați fișierul executabil **eclipse.exe** pentru a deschide programul.

Pentru suportul de Spring și Maven este nevoie să instalăm extensia **Spring Tools** După deschiderea programului Eclipse, se accesează opțiunea **Help** din bara de instrumente de sus, apoi se selectează **Eclipse Marketplace**. Din fereastra care apare, se selectează meniul **Search** și în casuța de **Find** se introduce textul “Spring Tools Suite” și apăsați Enter. Din rezultatele apărute se caută “**Spring Tools Suite (STS) for Eclipse Kepler (4.3) 3.6.4. RELEASE**” (versiunea actuală la momentul redactării acestui document) și apoi **Install**. Urmăriți pașii indicați din fereastră pentru a finaliza instalarea.

4.2. Prezentarea aplicației

4.2.1. Baza de date

În continuare vom vizualiza structura bazei de date. Instrucțiunile pentru crearea acesteia sunt trecute în Anexa 1.

Baza de date este formată din 7 tabele: USERS, ASSETS, REQUESTS, TRANSACTIONS, COMPLAINTS, DEPARTMENTS și COUNTRY.

Asocierile dintre acestea sunt următoarele:

USERS 1:n ASSETS cu cheia străină în ASSETS cu numele ID_USER;
USERS 1:n REQUESTS cu cheia străină în REQUESTS cu numele ID_USER;
USERS 1:n TRANSACTIONS cu cheia străină în TRANSACTIONS cu numele ID_USER;
USERS 1:n COMPLAINTS cu cheia străină în COMPLAINTS cu numele ID_USER;
USERS n:1 DEPARTMENTS cu cheia străină în USERS cu numele ID_DEPARTMENT;

ASSETS 1:n REQUESTS cu cheia străină în REQUESTS cu numele ID_ASSET;
ASSETS 1:n TRANSACTIONS cu cheia străină în TRANSACTIONS cu numele ID_ASSET;
ASSETS 1:n COMPLAINTS cu cheia străină în COMPLAINTS cu numele ID_ASSET;
ASSETS n:1 USERS cu cheia străină în ASSETS cu numele ID_USER;

REQUESTS n:1 USERS cu cheia străină în REQUESTS cu numele ID_USER;
REQUESTS n:1 ASSETS cu cheia străină în REQUESTS cu numele ID_ASSET;

TRANSACTIONS n:1 USERS cu cheia străină în TRANSACTIONS cu numele ID_USER;
TRANSACTIONS n:1 ASSETS cu cheia străină în TRANSACTIONS cu numele ID_ASSET;

COMPLAINTS n:1 USERS cu cheia străină în COMPLAINTS cu numele ID_USER;
COMPLAINTS n:1 ASSETS cu cheia străină în COMPLAINTS cu numele ID_ASSET;

DEPARTMENTS 1:n USERS cu cheia străină în USERS cu numele ID_DEPARTMENT;

COUNTRY 1:n DEPARTMENTS cu cheia străină în DEPARTMENTS cu numele ID_COUNTRY;

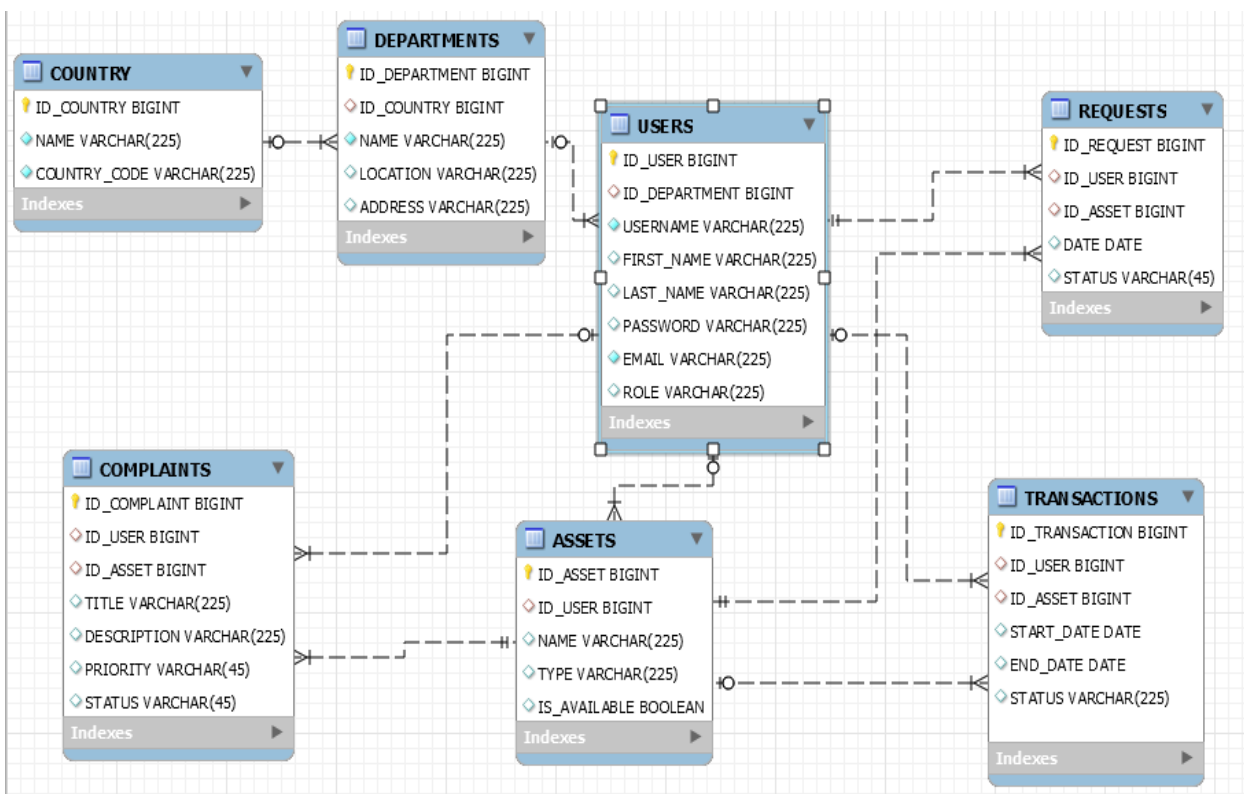


Figura 4.1 Schema bazei de date

4.2.2. Modulul de înregistrare

Pagina de start a aplicației, pentru utilizatorii care nu sunt autentificați, va conține două opțiuni: de logare sau înregistrare.

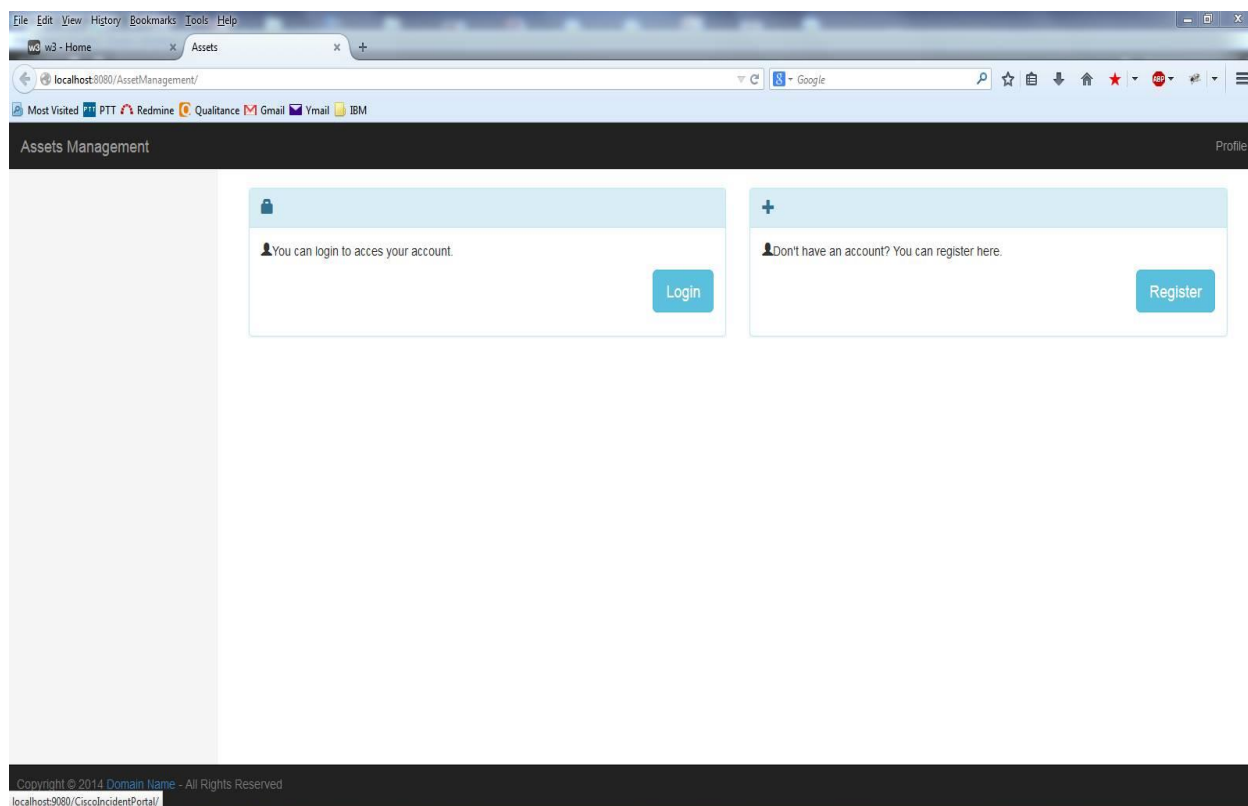


Figura 4.2. Pagina de start

Dacă utilizatorii selectează opțiunea de înregistrare, aceștia vor fi duși pe pagina de înregistrare. Aici ei vor completa câmpurile Username, Password, First Name, Last Name, Email, Department și vor avea și opțiunea de a adăuga și o poză de profil. Username, Password și Email sunt câmpuri obligatorii și trebuie neapărat completate pentru a finaliza înregistrarea.

Dacă mai există în baza de date încă un angajat cu același username sau email, atunci este afișat un mesaj de eroare și utilizatorul va fi întors în pagina de înregistrare.

Cu ajutorul taglib-ului de Spring, toate câmpurile vor fi mapate de un obiect de tip User, iar la acționarea butonului de Register, acesta este trimis în Back-End și prelucrat de un Controller de Spring în Java. Parola va fi criptată cu encriptarea BCrypt. În cazul în care înregistrarea s-a efectuat cu succes, atunci datele utilizatorului sunt salvate pe sesiunea de lucru și acesta este redirecționat către pagina de Profilul Meu (My Profile). În caz contrar, el este întors pe pagina de înregistrare însoțită de mesajul de eroare.

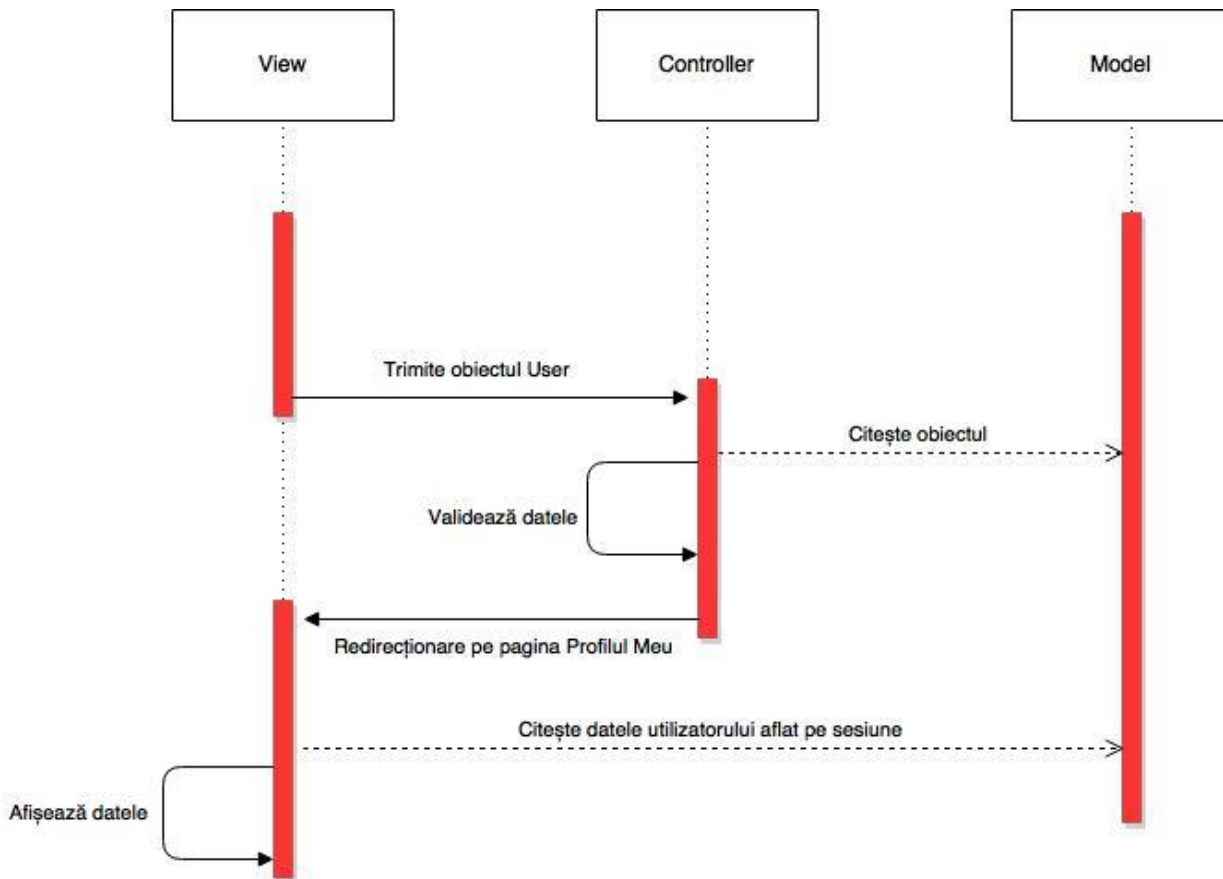


Figura 4.3. Diagrama de secvență pentru înregistrarea încheiată cu succes

Codul clasei de Java în Anexa 2.

4.2.3. Modulul de autentificare

Autentificarea funcționează pe modelul standard, utilizatorul trebuie să introducă username-ul și parola. După ce datele sunt introduse și butonul de logare este apăsat, acestea sunt trimise către un servlet predefinit de către framework-ul Spring și aici se va face verificarea lor în baza de date. Pentru verificarea parolei, controllerul criptează parola trimisă din formularul de autentificare și apoi o compară cu cea stocată în baza de date.

În cazul în care userul este valid, atunci acesta este dus în pagina de My Profile unde își poate vedea informațiile contului său.

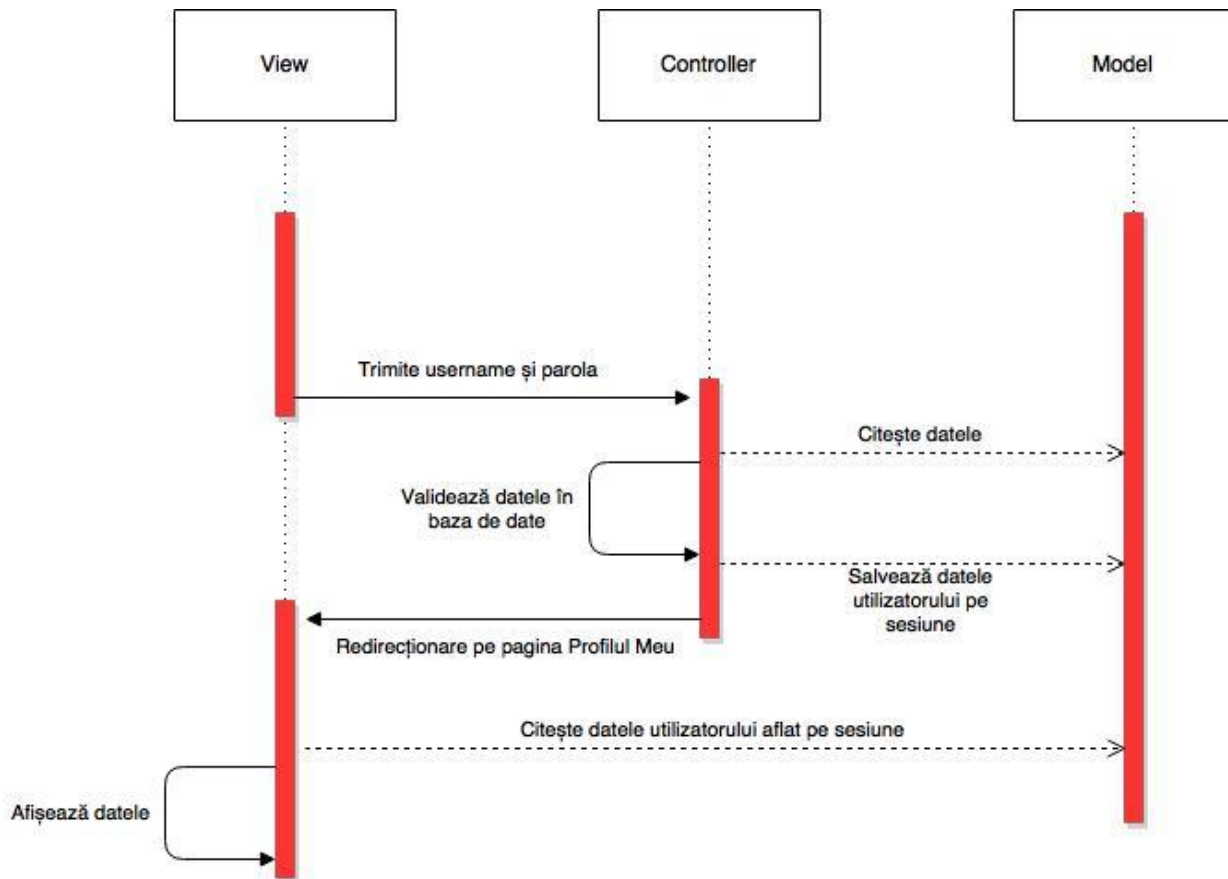


Figura 4.4. Diagrama de secvență pentru autentificarea încheiată cu succes

4.2.4. Modulul de profilul meu

După ce utilizatorul s-a autentificat sau înregistrat cu succes, acesta va fi dus pe pagina de My Profile. Aici el își poate vedea informațiile contului, având și posibilitatea de a le edita. Pe partea stânga a paginii sunt tab-urile de MyProfile, Assets, Complaints, Requests și Transactions prin care acesta poate naviga.

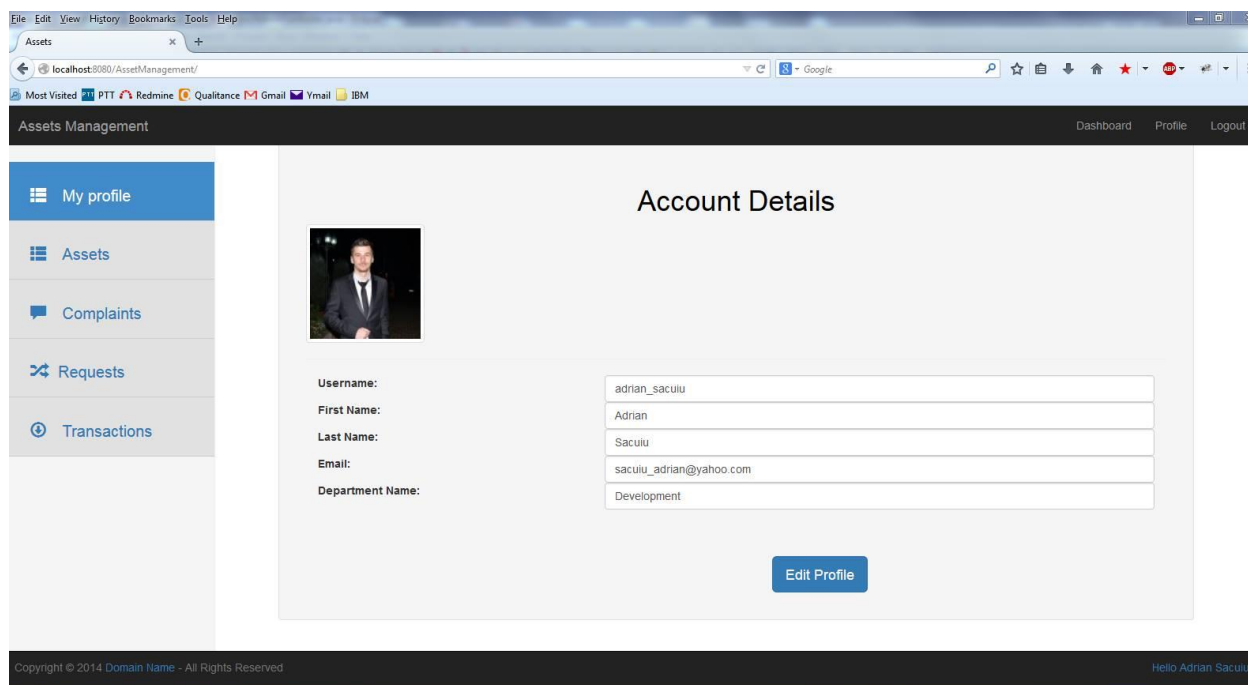


Figura 4.5 Pagina de My Profile

Contul de administrator mai are încă un tab în plus față de utilizatorul normal, numit Users.

În continuare vom detalia taburile atât pentru conturile de tip Angajat, cât și pentru Administrator.

4.2.4. Funcționalitățile angajatului

4.2.4.1. Modulul de Asset-uri

Angajații își pot vedea asset-urile atribuite accesând tabul Assets. Acestea vor fi afișate într-un tabel unde le sunt trecute numele și tipul. În pagină mai există și un buton, **Request Asset**, pentru a face o cerere pentru un asset nou. Nu se pot depune două cereri identice.

Utilizatorul poate da click pe unul din ele pentru a vedea mai multe detalii. Alături sunt și două butoane: Complaint (pentru a face o plângere din cauza asset-ului respectiv) și Remove Asset (pentru a elimina asset-ul din posesia angajatului).

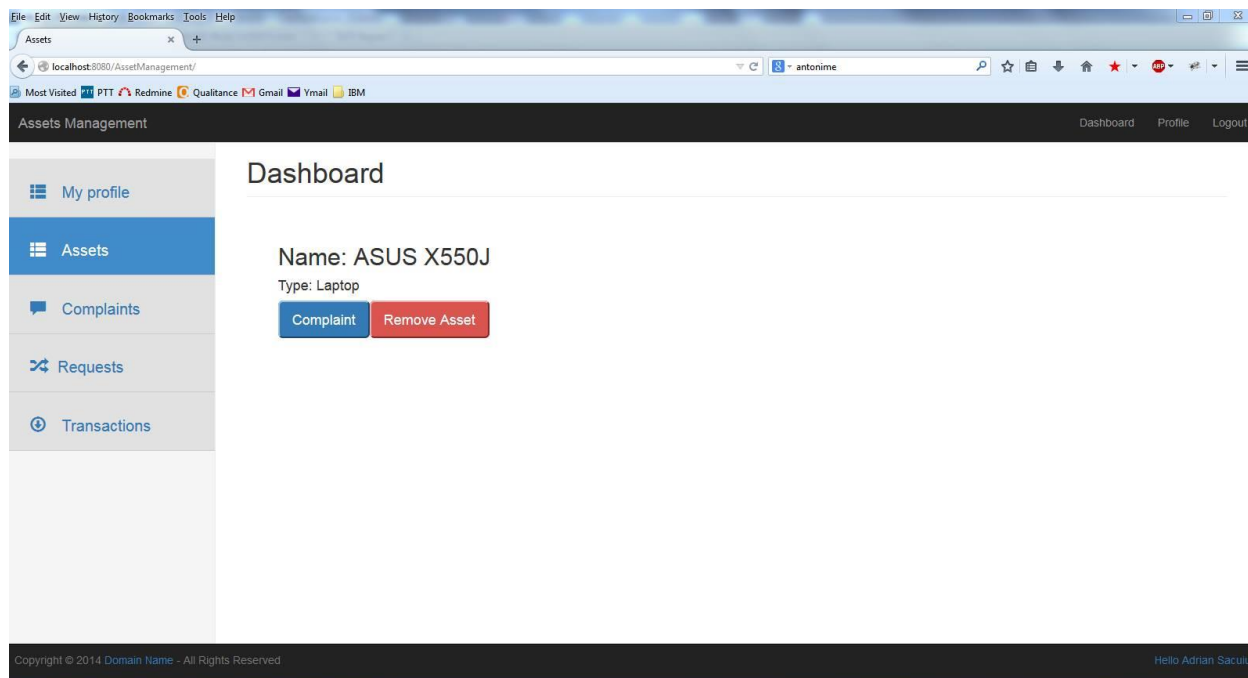


Figura 4.6 Pagina de detalii a asset-ului

Codul clasei de Java în Anexa 3.

4.2.4.2. Modulul de Plângeri

Pentru a vedea plângerile depuse ale angajatului, acesta poate accesa tab-ul **Complaints**. În această pagina sunt trecute plângerile într-un tabel și sunt afișate titlul, descrierea, prioritatea și statusul, dacă au fost rezolvate sau nu.

4.2.4.3. Modulul de Cereri

Utilizatorul își poate vedea cererile pentru noi asset-uri în tab-ul **Requests**. Acestea sunt trecute într-un tabel unde le sunt afișate id-ul, data la care s-a făcut cererea și statusul cererii.

4.2.4.4. Modulul de Tranzacții

Toate tranzacțiile utilizatorului pot fi văzute prin accesarea tabului **Transactions**. Tranzacțiile se creează automat, odată cu cererile, și se termină atunci când cererea a fost acceptată sau respinsă.

În tabel sunt trecute id-ul tranzacției, data de start, data de încheiere și statusul tranzacției.

4.2.5. Funcționalitățile administratorului

Scopul administratorului este de a gestiona acțiunile angajaților, cum ar fi cererile sau plângerile acestora.

4.2.5.1. Modulul de Users

În tab-ul Users , administratorul vede toți angajații din firmă. Acesta nu are posibilitatea de a-i edita.

4.2.5.2. Modulul de Asset-uri

Accesând tab-ul Assets, administratorul poate vedea toate asset-urile din firmă, pe cele alocate angajaților, dar și pe cele disponibile.

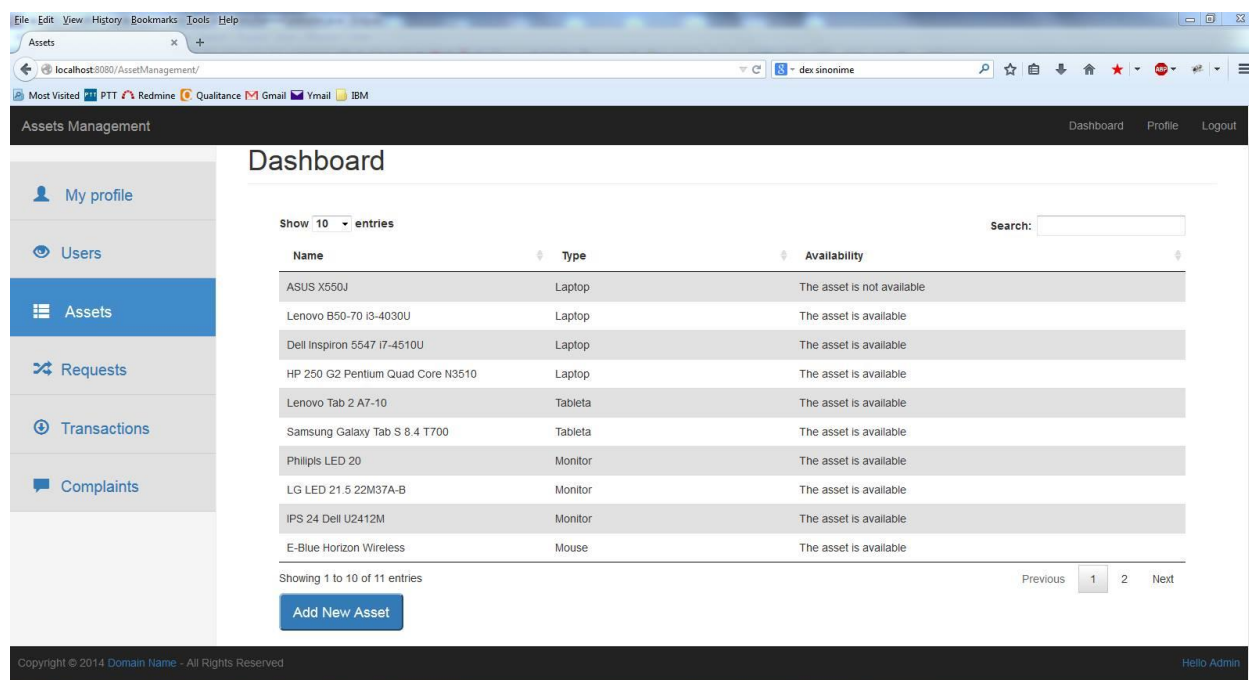


Figura 4.7 Pagina de asset-uri a administratorului

Tot el poate adăuga noi asset-uri în contul companiei. În această pagina se afla și butonul pentru adăugare. La acționarea acestuia v-a apărea formularul următor, în care vor fi trecute numele și tipul noului asset.

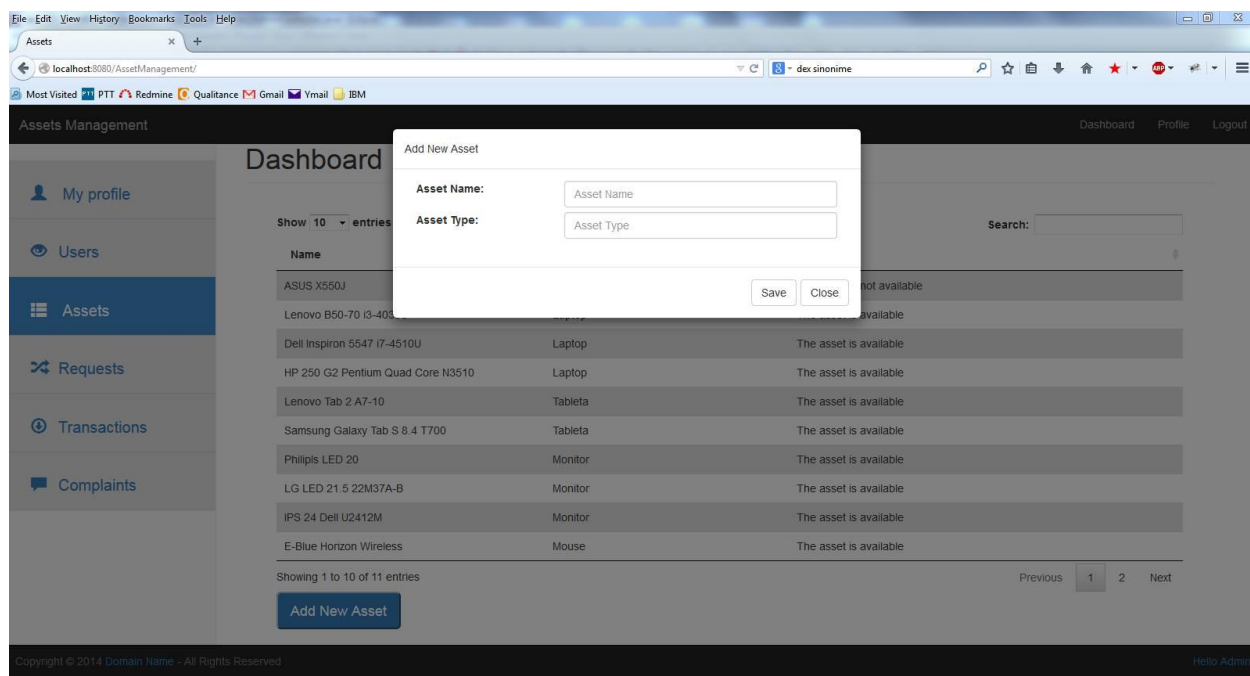


Figura 4.8 Formularul de adăugare a noului asset

Codul clasei de Java în Anexa 4.

4.2.5.3. Modulul de Plângeri

Administratorul va vedea toate plângerile angajaților în tab-ul Complaints, acestea fiind sortate în așa fel încât să fie afișate mai întâi cele nerezolvate. Selectând una dintre ele, ne vor fi afișate toate detaliile acesteia, având și opțiunea de a marca plângerea ca fiind rezolvată, apăsând pe butonul Solve Complaint.

4.2.5.4. Modulul de Cereri

În tab-ul Requests, administratorul vede toate cererile utilizatorilor, fiind afișate în tabel mai întâi cele mai noi cereri. Apăsând pe una dintre ele, administratorul poate vedea mai multe detalii despre cerere: ce angajat a făcut cererea, pentru ce asset este cererea și data creării acesteia.

De asemenea, vor fi disponibile și opțiuni pentru a aproba sau respinge cererea. În cazul în care sunt mai multe cereri pentru același asset și una dintre ele se aprobă, celelalte vor fi automat respinse.

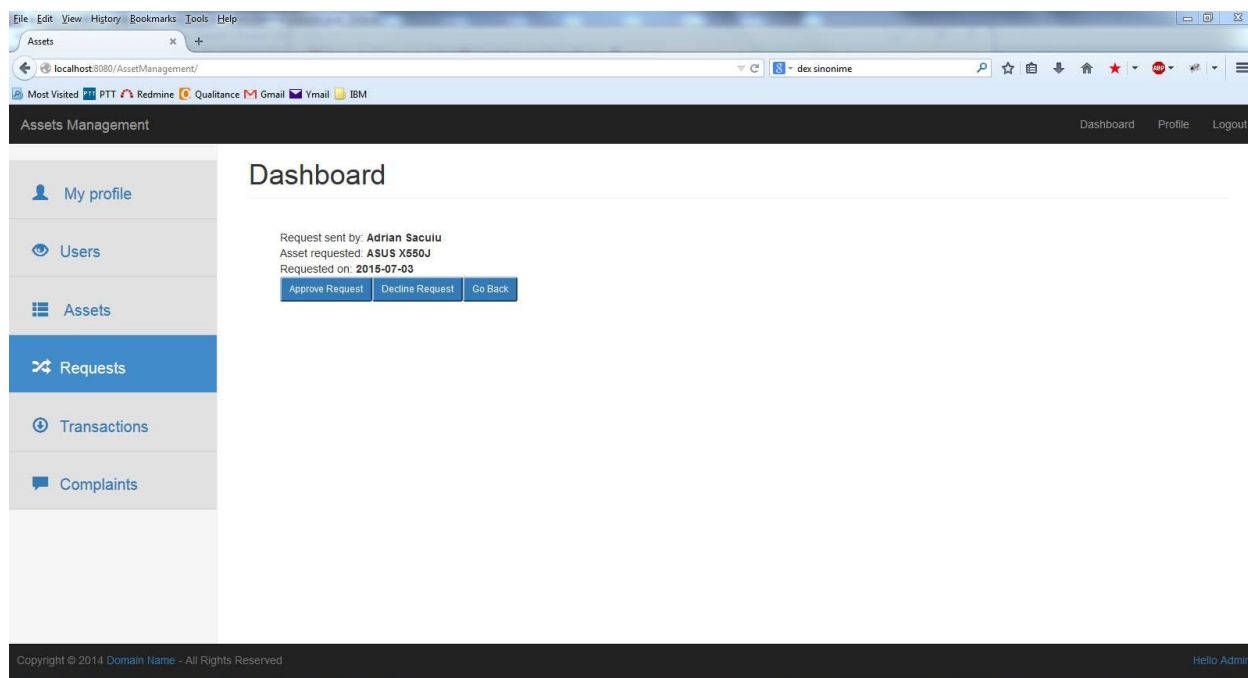


Figura 4.9 Pagina administratorului de vizionare a cererii

Codul clasei de Java în Anexa 4.

4.2.5.5. Modulul de Tranzacții

În tab-ul Transactions, administratorul vede toate tranzacțiile efectuate pentru primirea asset-urilor, sortate în ordinea începerii lor.

4.2.6. Securitatea aplicației

Securitatea în această aplicație este implementată cu ajutorul framework-ului Spring Security.

Spring Security este un framework de securitate care asigură securitate declarativă pentru aplicațiile bazate pe Spring. Acesta asigură soluții complete de securitate, manipularea autentificării și autorizării atât la nivelul de cereri web, cât și la nivelul de invocare a metodelor. Bazat pe framework-ul Spring, Spring Security profită la maxim de tehnicile de injectare a dependențelor și orientare pe aspecte.

Accesul la paginile și funcționalitățile aplicației noastre este gestionat de Spring Security în funcție de tipul de utilizator logat. În cazul nostru sunt 3 tipuri de utilizatori: administrator, angajat și anonim(nu este autentificat).

Pentru activarea a activa Spring Security în aplicație, trebuie adăugat în resursele proiectului fișierul AssetManagement-security.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/security"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security.xsd">

    <http auto-config="false" use-expressions="true">
        <access-denied-handler error-page="/views/AccessDenied.jsp"/>
        <form-login login-page="/#auth_request" authentication-failure url="/#Login_error"
            default-target-url="/saveUserOnSession" always-use-default-target="true" />
        <logout invalidate-session="true" logout-success-url="/" delete-cookies="JSESSIONID" />
        <intercept-url pattern="/index.jsp" access="permitAll" />
        <intercept-url pattern="/resources/**" access="permitAll" />
        <intercept-url pattern="/views/Login.jsp" access="isAnonymous()" />
        <intercept-url pattern="/views/register.jsp" access="isAnonymous()" />
        <intercept-url pattern="/views/myProfile.jsp" access="hasRole('ROLE_USER')" />
    </http>

    <jdbc-user-service id="userService" data-source-ref="dataSource"
        authorities-by-username-query="select username, concat('ROLE_',upper(replace(ROLE, ' ',
        '_')) as 'authority' from users where username=?"
        users-by-username-query="select username, password, 1 from USERS where username=?" />

    <authentication-manager>
        <authentication-provider user-service-ref="userService">
            <password-encoder hash="bcrypt" />
        </authentication-provider>
    </authentication-manager>
</beans:beans>
```

Spre exemplu, securitatea la nivelul elementelor de HTML din pagină. Pagina de start, cea în care sunt afișate opțiunile de Login și Register și pagina de My Profile sunt una și aceeași pagina, dar în funcție de tipul utilizatorului care o accesează aceasta are conținut diferit.

Acest lucru este posibil prin adăugarea tag-urilor JSP de Spring Security:

```
<security:authorize access="isAnonymous()">
  <div class="row" id="info">
    <div class="col-sm-6">
      <div class="panel panel-info">
        <div class="panel-heading">
          <h3 class="panel-title glyphicon glyphicon-lock"></h3>
        </div>
        <div class="panel-body">
          <p class="glyphicon glyphicon-user pull-left"></p><p> You can login to aces your account.</p><p class="pull-right"><button type="button" id="button1" class="btn btn-info btn-lg">
        </div>
      </div>
    </div>
    <div class="col-sm-6">
      <div class="panel panel-info">
        <div class="panel-heading">
          <h3 class="panel-title glyphicon glyphicon-plus"></h3>
        </div>
        <div class="panel-body">
          <p class="glyphicon glyphicon-user pull-left"></p><p> Don't have an account? You can register here.</p><p class="pull-right"><button type="submit" id="button2" class="btn btn-in
        </div>
      </div>
    </div>
  </div>
</security:authorize>
<security:authorize access="isAuthenticated()">
  <jsp:include page="views/myProfile.jsp" flush="true" />
</security:authorize>
```

Figura 4.10 Bucată din codul paginii de start

Codul paginii de start în Anexa 5.

Codul este pentru conținutul paginii este înconjurat de marcajul `<security:authorize>` care conține atributul `access`. Vedem că acesta are valori diferite în cele două cazuri: `isAnonymous()` care semnifică că vor avea acces la conținut doar persoanele care nu sunt autentificate, iar `isAuthenticated()` doar persoanele autentificate, indiferent că acesta este administrator sau angajat.

Mai avem și securitatea la nivelul autentificării. În fișierul XML afișat mai sus, este definită pagina de logare, pagina care va fi afișată în cazul în care utilizatorul nu s-a putut autentifica, paginile care vor fi interceptate de securitate(la acesarea acestor pagini se va verifica tipul utilizatorului și pe baza acestuia se va determina dacă va fi afișată pagina sau nu). După cum s-a precizat și în prezentarea modulului de autentificare, parola din baza de date este criptată. Pentru a nu mai fi nevoie de decriptarea acesteia pentru a verifica dacă este aceeași cu cea introdusă în formularul de logare, Spring Security o criptează pe cea din formular și apoi compară cele două parole criptate.

În cazul în care utilizatorul încearcă să acceseze o resursă sau pagină web la care nu are acces, el va fi redirectionat pe pagina următoare:

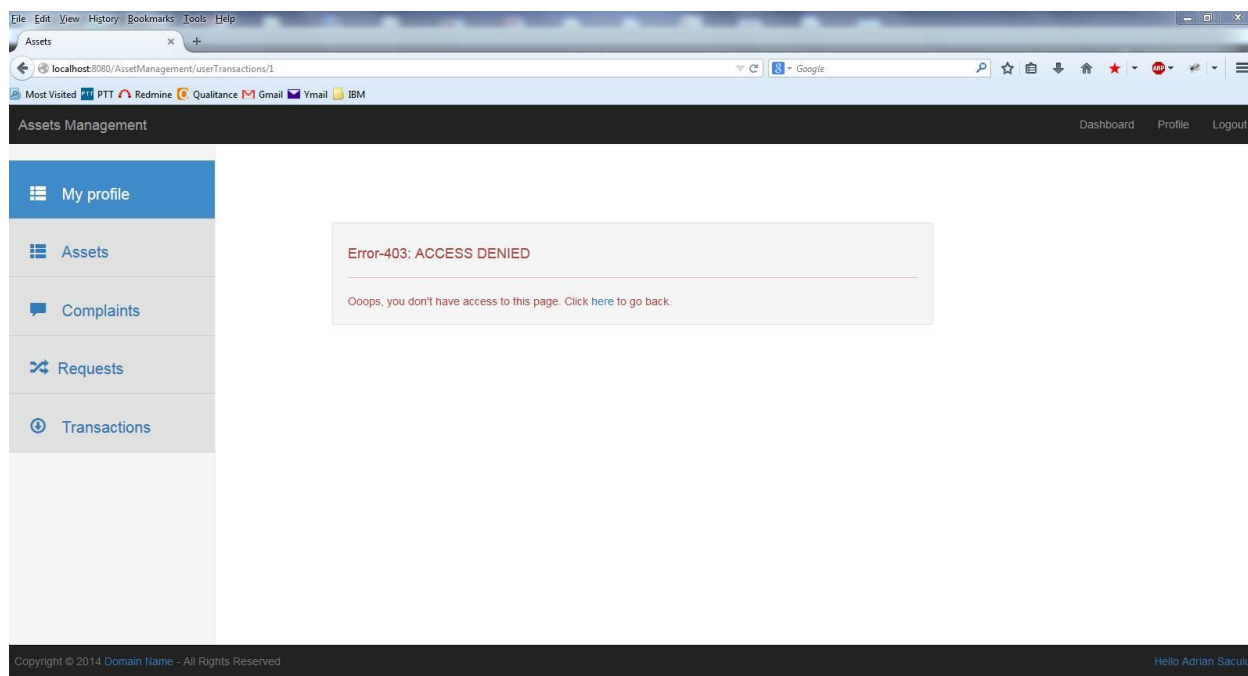


Figura 4.11 Pagina de eroare pentru acces interzis

Securitatea mai este implementată la nivelul metodelor claselor de Java:

```
@PreAuthorize("hasRole('ROLE_ADMIN')")
@RequestMapping(value = "createAsset", method = RequestMethod.POST, produces =
    "application/json")
@ResponseBody
public Map<String, Object> createAsset(HttpServletRequest request) {

    // conținutul metodei

}
```

Anotarea `@PreAuthorize` verifica condiția din paranteză înainte invocării metodei pe care este aplicată. În cazul de față, utilizatorul trebuie să aibă rolul de administrator pentru a o putea apela, dacă nu, este redirecționat pe pagina afișată mai sus.

CONCLUZII

În urma realizării acestui proiect, am obținut o aplicație web de gestiune și asignare a asset-urilor într-o companie IT perfect funcțională și ușor de utilizat.

Deoarece aplicația folosește un cluster de baze de date MariaDB, acesta previne scoaterea din funcțiune a site-ului în cazul defectării unuia dintre serverele de baze de date. Acest lucru a fost testat și s-a dovedit a funcționa optim.

Datorită folosirii unora dintre cele mai avansate tehnologii de programare web, timpul de răspuns al operațiilor efectuate în aplicație, precum și al interogărilor în baza de date, este unul foarte scurt, navigarea prin paginile aplicației fiind accesibilă într-un timp rapid.

Bibliografie

- [1] http://www.w3.org/community/webed/wiki/The_basics_of_HTML, accesat la data: 22.03.2015
- [2] <https://en.wikipedia.org/wiki/HTML>, accesat la data: 28.03.2015
- [3] https://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition, accesat la data: 29.03.2015
- [4] <https://en.wikipedia.org/wiki/JavaScript>, accesat la data: 05.04.2015
- [5] https://en.wikipedia.org/wiki/Cascading_Style_Sheets, accesat la data: 12.04.2015
- [6] Craig Walls, *Spring in action Third Edition*, Editura Manning Publications Co, SUA, 2011
- [7] Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems SIXTH EDITION*, Editura Pearson, SUA, 2010
- [8] http://databasemanagement.wikia.com/wiki/DBMS_Functions, accesat la data: 17.04.2015
- [9] <http://www.tutorialspoint.com/dbms>, accesat la data: 22.04.2015
- [10] <http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf>, accesat la data: 02.05.2015
- [11] <https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html>, accesat la data: 12.05.2015
- [12] <https://www.mongodb.com/nosql-explained>, accesat la data: 19.05.2015
- [13] <http://www.scaledb.com/pdfs/ArchitecturePrimer.pdf>, accesat la data: 22.05.2015
- [14] <http://galeracluster.com/documentation-webpages/>, accesat la data: 30.05.2015
- [15] <https://mariadb.com/kb/en/mariadb/mariadb-galera-cluster-known-limitations/>, accesat la data: 16.06.2015
- [16] - https://en.wikipedia.org/wiki/Hibernate_%28Java%29, accesat la data: 05.04.2015
- [17] - <https://en.wikipedia.org/wiki/JQuery>, accesat la data: 06.04.2015
- [18] - https://en.wikipedia.org/wiki/Apache_Maven, accesat la data: 06.04.2015
- [19] - https://en.wikipedia.org/wiki/Eclipse_%28software%29, accesat la data: 07.04.2015
- [20] - <https://ro.wikipedia.org/wiki/Model-view-controller>, accesat la data: 18.06.2015

ANEXE

Anexa 1

Database.sql

```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
```

```
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,  
FOREIGN_KEY_CHECKS=0;
```

```
SET @OLD_SQL_MODE=@@SQL_MODE,  
SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';
```

```
-- -----  
-- Schema AssetsManagement
```

```
-- -----  
-- Schema AssetsManagement
```

```
CREATE SCHEMA IF NOT EXISTS `AssetsManagement` ;
```

```
USE `AssetsManagement` ;
```

```
-- -----  
-- Table `AssetsManagement`.`COUNTRY`
```

```
CREATE TABLE IF NOT EXISTS `AssetsManagement`.`COUNTRY` (  
  `ID_COUNTRY` BIGINT NOT NULL,  
  `NAME` VARCHAR(225) NOT NULL,
```

```

`COUNTRY_CODE` VARCHAR(225) NOT NULL,
PRIMARY KEY (`ID_COUNTRY`),
UNIQUE INDEX `NAME_UNIQUE` (`NAME` ASC),
UNIQUE INDEX `COUNTRY_CODE_UNIQUE` (`COUNTRY_CODE` ASC))
ENGINE = InnoDB;

```

```

-- -----
-- Table `AssetsManagement`.`DEPARTMENTS`
-- -----

```

```

CREATE TABLE IF NOT EXISTS `AssetsManagement`.`DEPARTMENTS` (
  `ID_DEPARTMENT` BIGINT NOT NULL AUTO_INCREMENT,
  `ID_COUNTRY` BIGINT NULL,
  `NAME` VARCHAR(225) NOT NULL,
  `LOCATION` VARCHAR(225) NULL,
  `ADDRESS` VARCHAR(225) NULL,
  PRIMARY KEY (`ID_DEPARTMENT`),
  INDEX `fk_Departments_Country1_idx` (`ID_COUNTRY` ASC),
  UNIQUE INDEX `NAME_UNIQUE` (`NAME` ASC),
  CONSTRAINT `fk_Departments_Country1`
    FOREIGN KEY (`ID_COUNTRY`)
      REFERENCES `AssetsManagement`.`COUNTRY` (`ID_COUNTRY`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

```

```

-----
-- Table `AssetsManagement`.`USERS`
-----

CREATE TABLE IF NOT EXISTS `AssetsManagement`.`USERS` (
  `ID_USER` BIGINT NOT NULL AUTO_INCREMENT,
  `ID_DEPARTMENT` BIGINT NULL,
  `USERNAME` VARCHAR(225) NOT NULL,
  `FIRST_NAME` VARCHAR(225) NULL,
  `LAST_NAME` VARCHAR(225) NULL,
  `PASSWORD` VARCHAR(225) NULL,
  `EMAIL` VARCHAR(225) NOT NULL,
  `ROLE` VARCHAR(225) NULL,
  PRIMARY KEY (`ID_USER`),
  INDEX `fk_Users_Departments1_idx` (`ID_DEPARTMENT` ASC),
  UNIQUE INDEX `EMAIL_UNIQUE` (`EMAIL` ASC),
  UNIQUE INDEX `USERNAME_UNIQUE` (`USERNAME` ASC),
  CONSTRAINT `fk_Users_Departments1`
    FOREIGN KEY (`ID_DEPARTMENT`)
      REFERENCES `AssetsManagement`.`DEPARTMENTS` (`ID_DEPARTMENT`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

```

```
-- -----  
-- Table `AssetsManagement`.`ASSETS`  
-- -----
```

```
CREATE TABLE IF NOT EXISTS `AssetsManagement`.`ASSETS` (  
  `ID_ASSET` BIGINT NOT NULL AUTO_INCREMENT,  
  `ID_USER` BIGINT NULL,  
  `NAME` VARCHAR(225) NULL,  
  `TYPE` VARCHAR(225) NULL,  
  `IS_AVAILABLE` TINYINT(1) NULL,  
  PRIMARY KEY (`ID_ASSET`),  
  INDEX `fk_Assets_Users1_idx` (`ID_USER` ASC),  
  CONSTRAINT `fk_Assets_Users1`  
    FOREIGN KEY (`ID_USER`)  
    REFERENCES `AssetsManagement`.`USERS` (`ID_USER`)  
    ON DELETE NO ACTION  
    ON UPDATE NO ACTION)  
ENGINE = InnoDB;
```

```
-- -----  
-- Table `AssetsManagement`.`TRANSACTIONS`  
-- -----
```

```
CREATE TABLE IF NOT EXISTS `AssetsManagement`.`TRANSACTIONS` (  
  `ID_TRANSACTION` BIGINT NOT NULL AUTO_INCREMENT,  
  `ID_USER` BIGINT NULL,  
  `ID_ASSET` BIGINT NULL,
```



```

`START_DATE` DATE NULL,

`END_DATE` DATE NULL,

`STATUS` VARCHAR(225) NULL,

PRIMARY KEY (`ID_TRANSACTION`),

INDEX `fk_Transactions_Assets1_idx` (`ID_ASSET` ASC),

INDEX `fk_Transactions_Users1_idx` (`ID_USER` ASC),

CONSTRAINT `fk_Transactions_Assets1`

    FOREIGN KEY (`ID_ASSET`)

    REFERENCES `AssetsManagement`.`ASSETS` (`ID_ASSET`)

    ON DELETE CASCADE

    ON UPDATE NO ACTION,

CONSTRAINT `fk_Transactions_Users1`

    FOREIGN KEY (`ID_USER`)

    REFERENCES `AssetsManagement`.`USERS` (`ID_USER`)

    ON DELETE CASCADE

    ON UPDATE NO ACTION)

ENGINE = InnoDB;

```

```

-----
-- Table `AssetsManagement`.`REQUESTS`
-----

```

```

CREATE TABLE IF NOT EXISTS `AssetsManagement`.`REQUESTS` (

    `ID_REQUEST` BIGINT NOT NULL AUTO_INCREMENT,

    `ID_USER` BIGINT NULL,

    `ID_ASSET` BIGINT NULL,

```

```

`DATE` DATE NULL,

`STATUS` VARCHAR(45) NULL,

PRIMARY KEY (`ID_REQUEST`),

INDEX `fk_Requests_Assets1_idx` (`ID_ASSET` ASC),

INDEX `fk_Requests_Users1_idx` (`ID_USER` ASC),

CONSTRAINT `fk_Requests_Assets1`

FOREIGN KEY (`ID_ASSET`)

REFERENCES `AssetsManagement`.`ASSETS` (`ID_ASSET`)

ON DELETE CASCADE

ON UPDATE NO ACTION,

CONSTRAINT `fk_Requests_Users1`

FOREIGN KEY (`ID_USER`)

REFERENCES `AssetsManagement`.`USERS` (`ID_USER`)

ON DELETE CASCADE

ON UPDATE NO ACTION)

ENGINE = InnoDB;

```

```

-----

-- Table `AssetsManagement`.`COMPLAINTS`

-----

CREATE TABLE IF NOT EXISTS `AssetsManagement`.`COMPLAINTS` (

`ID_COMPLAINT` BIGINT NOT NULL AUTO_INCREMENT,

`ID_USER` BIGINT NULL,

`ID_ASSET` BIGINT NULL,

`TITLE` VARCHAR(225) NULL,

```

```

`DESCRIPTION` VARCHAR(225) NULL,

`PRIORITY` VARCHAR(45) NULL,

`STATUS` VARCHAR(45) NULL,

PRIMARY KEY (`ID_COMPLAINT`),

INDEX `fk_Complaints_Users1_idx` (`ID_USER` ASC),

INDEX `fk_Complaints_Assets1_idx` (`ID_ASSET` ASC),

CONSTRAINT `fk_Complaints_Users1`

FOREIGN KEY (`ID_USER`)

REFERENCES `AssetsManagement`.`USERS` (`ID_USER`)

ON DELETE CASCADE

ON UPDATE NO ACTION,

CONSTRAINT `fk_Complaints_Assets1`

FOREIGN KEY (`ID_ASSET`)

REFERENCES `AssetsManagement`.`ASSETS` (`ID_ASSET`)

ON DELETE CASCADE

ON UPDATE NO ACTION)

ENGINE = InnoDB;

SET SQL_MODE=@OLD_SQL_MODE;

SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;

SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;

```

Anexa 2

Register.java

```
package controllers;

import java.util.List;

import javax.servlet.http.HttpServletRequest;

import entities.Department;
import entities.User;

import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.ModelAndView;

import services.DepartmentService;
import services.UsersService;
import util.ImageUploadException;
import static util.OperationsUtils.*;

@PreAuthorize("isAnonymous()")
@RequestMapping(value = "register")
@Controller
public class RegisterController {
    private static final Logger logger = Logger.getLogger(RegisterController.class);

    @Autowired
    private UsersService userService;

    @Autowired
    private DepartmentService departmentService;

    @PreAuthorize("isAnonymous()")
    @RequestMapping(method = RequestMethod.GET)
    public ModelAndView createUser() {
        logger.info("Inside createUser method");

        ModelAndView mv = new ModelAndView("views/register.jsp");
        try{
            List<Department> departments = departmentService.getAllDepartments();
            mv.addObject("user", new User());
            mv.addObject("departments", departments);
        } catch(Exception e){
```

```

        logger.error("in createUser method Exception: " + e.getMessage());
    }
    return mv;
}

@PreAuthorize("isAnonymous()")
@RequestMapping(method = RequestMethod.POST, produces = "application/json")
public ModelAndView registerUser(@ModelAttribute("user") User user,
HttpServletRequest request, @RequestParam(value = "image", required = false) MultipartFile
image) {
    logger.info("Inside registerUser method");
    ModelAndView modelAndView = new ModelAndView("");
    Long idDepartment = 0L;
    Department department = null;

    try {
        idDepartment = Long.parseLong(request.getParameter("idDepartment"));
        String username = user.getUsername();
        department = departmentService.getDepartmentById(idDepartment);
        user.setDepartment(department);

        if (image != null && !image.isEmpty()) {
            validateImage(image);
            saveImage(username, image);
        }

        String password = user.getPassword();
        userService.addUser(user);

        user.setPassword(password);
        modelAndView.addObject("registered_user", user);

    } catch (ImageUploadException iue) {
        logger.error("in registerUser method ImageUploadException: " +
iue.getMessage());
        iue.printStackTrace();
        modelAndView.setViewName("#register");
        modelAndView.addObject("error", iue.getMessage());
    }

    catch (Exception e) {
        logger.error("in registerUser method Exception: " + e.getMessage() +
"; Cause: " + e.getCause());
        e.printStackTrace();
        modelAndView.setViewName("#register");
        modelAndView.addObject("error", "Error registering new user!");
    }

    return modelAndView;
}
}

```

Anexa 3

UserController.java

```
package controllers;

import java.sql.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;

import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import services.AssetService;
import services.ComplaintService;
import services.RequestService;
import services.TransactionService;
import services.UsersService;
import entities.Asset;
import entities.Complaint;
import entities.Request;
import entities.Transaction;
import entities.User;

@Controller
public class UserController {

    private static final Logger logger = Logger.getLogger(UserController.class);

    @Autowired
    private UsersService usersService;

    @Autowired
    private AssetService assetService;

    @Autowired
    private ComplaintService complaintService;

    @Autowired
    private RequestService requestService;

    @Autowired
    private TransactionService transactionService;

    @PreAuthorize("(hasRole('ROLE_USER') and #username == principal.username)")
    @RequestMapping(value = "{username}/createComplaint", produces = "application/json")
    @ResponseBody
```

```

    public Map<String, Object> createComplaint(@PathVariable String username,
HttpServletRequest request) {
    logger.info("Inside createComplaint method");
    Map<String, Object> resultMap = new HashMap<String, Object>();
    Complaint complaint = new Complaint();
    Long idAsset = 0L;

    try {
        complaint.setTitle(request.getParameter("idAsset"));
        complaint.setDescription(request.getParameter("description"));
        complaint.setPriority(request.getParameter("priority"));
        idAsset = Long.parseLong(request.getParameter("idAsset"));

        User user = userService.getUserByUsername(username);
        Asset asset = assetService.getAssetById(idAsset);

        complaint.setUser(user);
        complaint.setAsset(asset);

        boolean result = complaintService.addComplaint(complaint);
        if (result) {
            resultMap.put("status", "true");
            resultMap.put("message", "Complaint created successfully!");
        } else {
            logger.error("Error creating new request!");
            resultMap.put("status", "false");
            resultMap.put("message", "Error creating new complaint!");
        }
    } catch (Exception e) {
        logger.error("in createComplaint method Exception: " + e.getMessage()
+ "; Cause: " + e.getCause());
        e.printStackTrace();
        resultMap.put("status", "false");
        resultMap.put("message", "Error creating new complaint!");
    }

    return resultMap;
}

@PreAuthorize("(hasRole('ROLE_USER') and #username == principal.username)")
@RequestMapping(value = "{username}/removeAsset", produces = "application/json")
@ResponseBody
    public Map<String, String> removeAsset(@PathVariable String username,
HttpServletRequest request) {
    logger.info("Inside removeAsset method");
    Map<String, String> resultMap = new HashMap<String, String>();
    Long idAsset = 0L;

    try {
        idAsset = Long.parseLong(request.getParameter("idAsset"));
        Asset asset = assetService.getAssetById(idAsset);
        if (asset.getUser().getUsername().equals(username)) {

            asset.setUser(null);
            asset.setIsAvailable(true);

```

```

        boolean result = assetService.updateAsset(asset);
        if (result) {
            resultMap.put("status", "true");
            resultMap.put("message", "Asset removed successfully!");
        } else {
            Logger.error("Error creating new request!");
            resultMap.put("status", "false");
            resultMap.put("message", "Error removing asset!");
        }

    } else {
        Logger.error("Error creating new request!");
        resultMap.put("status", "false");
        resultMap.put("message", "Error removing asset! This asset
doesn't belong to you!");
    }

    } catch (Exception e) {
        Logger.error("in removeAsset method Exception: " + e.getMessage() + ";
Cause: " + e.getCause());
        e.printStackTrace();
        resultMap.put("status", "false");
        resultMap.put("message", "Error removing asset!");
    }

    return resultMap;
}
}

```


Anexa 4

AdminController.java

```
package controllers;

import java.sql.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;

import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

import services.AssetService;
import services.ComplaintService;
import services.DepartmentService;
import services.RequestService;
import services.TransactionService;
import services.UsersService;
import entities.Asset;
import entities.Complaint;
import entities.Request;
import entities.Transaction;
import entities.User;

@PreAuthorize("hasRole('ROLE_ADMIN')")
@Controller
public class AdminController {
    private static final Logger logger = Logger.getLogger(AdminController.class);

    @Autowired
    private UsersService usersService;

    @Autowired
    private AssetService assetService;

    @Autowired
    private ComplaintService complaintService;

    @Autowired
    private RequestService requestService;

    @Autowired
    private TransactionService transactionService;

    @Autowired
```

```

    private DepartmentService departmentService;

@RequestMapping(value = "createAsset", method = RequestMethod.POST, produces =
"application/json")
@ResponseBody
    public Map<String, Object> createAsset(HttpServletRequest request) {
        Logger.info("Inside createAsset method");
        Map<String, Object> resultMap = new HashMap<String, Object>();
        Asset asset = new Asset();

        String name = request.getParameter("name");
        String type = request.getParameter("type");

        try {
            asset.setName(name);
            asset.setType(type);
            boolean result = assetService.addAsset(asset);
            if (result) {
                resultMap.put("status", "true");
                resultMap.put("message", "Asset created successfully!");
            } else {
                Logger.error("Error creating new asset!");
                resultMap.put("status", "false");
                resultMap.put("message", "Error creating new asset!");
            }
        } catch (Exception e) {
            Logger.error("in createAsset method Exception: " + e.getMessage() + ";
Cause: " + e.getCause());
            e.printStackTrace();
            resultMap.put("status", "false");
            resultMap.put("message", "Error creating new asset!");
        }
        return resultMap;
    }

@RequestMapping(value = "assignAsset", produces = "application/json")
@ResponseBody
    public Map<String, Object> assignAsset(HttpServletRequest servletRequest) {
        Logger.info("Inside assignAsset method");
        Map<String, Object> resultMap = new HashMap<String, Object>();
        Long idUser = 0L;
        Long idAsset = 0L;

        try {
            idUser = Long.parseLong(servletRequest.getParameter("idUser"));
            idAsset = Long.parseLong(servletRequest.getParameter("idAsset"));

            User user = userService.getUserById(idUser);
            Asset asset = assetService.getAssetById(idAsset);
            Request request = requestService.getNewRequestByUserAndAsset(idUser,
idAsset);

            Transaction transaction =
transactionService.getPendingTransactionByUserAndAsset(idUser, idAsset);

            if (request != null && transaction != null) {

```

```

        try {
            asset.setUser(user);
            asset.setIsAvailable(false);
            assetService.updateAsset(asset);

            request.setStatus("Done");
            requestService.updateRequest(request);

            transaction.setStatus("Success");
            transaction.setEndDate(new
Date(System.currentTimeMillis()));
            transactionService.updateTransaction(transaction);

            requestService.rejectRequestsByIdAsset(idAsset);
            transactionService.declineTransactionsByIdAsset(idAsset);

            resultMap.put("status", "true");
            resultMap.put("message", "Asset assigned successfully!");
        } catch (Exception e) {
            if (asset.getIsAvailable() == false) {
                asset.setUser(null);
                asset.setIsAvailable(true);
                assetService.updateAsset(asset);

            } else if (request.getStatus().equals("Done")) {
                request.setStatus("New");
                requestService.updateRequest(request);
            }

            e.printStackTrace();
            Logger.error("in assignAsset method Exception1: " +
e.getMessage() + "; Cause: " + e.getCause());
            resultMap.put("status", "false");
            resultMap.put("message", "Error assigning asset!");
        }

        } else {
            Logger.error("Error assigning asset! No request or
transaction!");

            resultMap.put("status", "false");
            resultMap.put("message", "Error assigning asset!");
        }

    } catch (Exception e) {
        e.printStackTrace();
        Logger.error("in assignAsset method Exception2: " + e.getMessage() +
"; Cause: " + e.getCause());
        resultMap.put("status", "false");
        resultMap.put("message", "Error assigning asset!");
    }

    return resultMap;
}

@RequestMapping(value = "rejectRequest/{idRequest}", produces = "application/json")
@ResponseBody

```

```

public Map<String, Object> rejectRequest(@PathVariable Long idRequest) {
    Logger.info("Inside rejectRequest method");
    Map<String, Object> resultMap = new HashMap<String, Object>();
    Request request = null;
    Transaction transaction = null;
    Long idUser = 0L;
    Long idAsset = 0L;

    try {
        request = requestService.getRequestById(idRequest);
        idUser = request.getUser().getIdUser();
        idAsset = request.getAsset().getIdAsset();
        transaction
transactionService.getPendingTransactionByUserAndAsset(idUser, idAsset);

        transaction.setEndDate(new Date(System.currentTimeMillis()));
        transaction.setStatus("Declined");
        request.setStatus("Rejected");

        transactionService.updateTransaction(transaction);
        requestService.updateRequest(request);

        resultMap.put("status", "true");
        resultMap.put("message", "Request rejected successfully");

    } catch (Exception e) {
        Logger.error("in rejectRequest method Exception: " + e.getMessage() +
"; Cause: " + e.getCause());
        e.printStackTrace();
        resultMap.put("status", "false");
        resultMap.put("message", "Error rejecting request!");
    }

    return resultMap;
}
}

```

Anexa 5

index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java"%>
<%@ taglib prefix="security" uri="http://www.springframework.org/security/tags" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html>
<html lang="en">
<!-- START PAGE SOURCE -->
<jsp:include page="views/TopMenu.jsp" flush="true" />

<div class="container-fluid">
  <div class="row">
    <jsp:include page="views/LeftMenu.jsp" flush="true" />

    <div class="col-sm-9 col-sm-offset-3 col-md-10 col-md-offset-2 main">
      <security:authorize access="isAuthenticated()">
        <h1 class="page-header">Dashboard</h1>
      </security:authorize>
      <script>
        $(function() {
          var $info = $('#info'), startInfo = $info.html();
          var j_username = '<c:out value="{registered_user.username}"/>';
          var j_password = '<c:out value="{registered_user.password}"/>';

          if(j_username!=""){
            jQuery.ajax({
              type : 'POST',
              url : 'j_spring_security_check',
              data :
                {j_username:j_username,j_password:j_password},
              success : function(data) {
                location.href = "/AssetManagement/";
              }
            });
          }
          $info.on('click', '#back', function() {

            $info.html(startInfo);

          });
        })
      </script>
      <security:authorize access="isAnonymous()">
        <div class="row" id="info">
          <div class="col-sm-6">
            <div class="panel panel-info">
              <div class="panel-heading">
                <h3 class="panel-title glyphicon glyphicon lock"></h3>

```

```

        </div>
        <div class="panel-body">
        <p class="glyphicon glyphicon-user pull-left"></p><p> You can login to acces
        your account.</p><p class="pull-right"><button type="button"
        id="button1" class="btn btn-info btn-lg">Login</button></p>
        </div>
        </div>
    </div>
    <div class="col-sm-6">
        <div class="panel panel-info">
            <div class="panel-heading">
                <h3 class="panel-title glyphicon glyphicon-plus"></h3>
            </div>
            <div class="panel-body">
                <p class="glyphicon glyphicon-user pull-left"></p><p> Don't have an
                account? You can register here.</p><p class="pull-right"><button
                type="submit" id="button2" class="btn btn-info btn-
                lg">Register</button></p>
            </div>
        </div>
    </div>
</div>
</div>
</div>
</security:authorize>
    <security:authorize access="isAuthenticated()">
        <jsp:include page="views/myProfile.jsp" flush="true" />

    </security:authorize>
    <div class="row placeholders">
        <jsp:include page="views/Footer.jsp" flush="true" />
    </div>
</div>
</div>
</div>
</body>
</html>

```