

MINISTERUL EDUCAȚIEI



UNIVERSITATEA TEHNICĂ

DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

SISTEM DE GESTIONARE A FACTURILOR SI CHELTUIELILOR

PROIECT DE DIPLOMĂ

Autor: **Răzvan Pavel**

Conducător științific: **Ș.L.dr.ing. Mihai Hulea**

2023



UNIVERSITATEA TEHNICĂ

DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Vizat,

DECAN

Prof. dr. ing. Liviu MICLEA

DIRECTOR DEPARTAMENT AUTOMATICĂ

Prof. dr. ing. Honoriu VĂLEAN

Autor: **Răzvan Pavel**

SISTEM DE GESTIONARE A FACTURILOR SI CHELTUIELILOR

1. **Enunțul temei:** *Lucrarea de licență realizată prezintă un sistem de gestionare a facturilor si a cheltuielilor utilizatorului. Lucrarea este bazată pe colectarea de date introduse de utilizator si stocarea acestora într-o baza de date pentru a putea fi monitorizate.*
2. **Conținutul proiectului:** *Pagina de prezentare, Declarație privind autenticitatea proiectului, Sinteza proiectului, Cuprins, Introducere, Studiu bibliografic, Analiză și fundamentare teoretica, Proiectare si impementare, Testare si validare, Concluzii, Bibliografie.*
3. **Locul documentării:** *Universitatea Tehnică din Cluj-Napoca*
4. **Consultanți:** -
5. **Data emiterii temei:** 23.11.2022
6. **Data predării:** 01.07.2023

Semnătura autorului

Semnătura conducătorului științific _____



**Declarație pe proprie răspundere privind
autenticitatea proiectului de diplomă**

Subsemnatul(a) **Răzvan PAVEL**,
legitimat(ă) cu CI/BI seria HD nr. 869971 , CNP 1990908204485,
autorul lucrării:

SISTEM DE GESTIONARE A FACTURILOR SI CHELTUIELILOR

elaborată în vederea susținerii examenului de finalizare a studiilor de licență la **Facultatea de Automatică și Calculatoare**, specializarea **Automatică și Informatică Aplicată**, din cadrul Universității Tehnice din Cluj-Napoca, sesiunea Iulie 2023 a anului universitar 2022-2023, declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate, în textul lucrării, și în bibliografie.

Declar, că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.

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

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative, respectiv, *anularea examenului de licență*.

Data

01.07.2023

Răzvan PAVEL



SINTEZA

proiectului de diplomă cu titlul:

Sistem de gestionare a facturilor si cheltuielilor

Autor: **Răzvan PAVEL**

Conducător științific: **Ș.L.dr.ing. Mihai Hulea**

1. Cerințele temei: dezvoltarea unui sistem de gestionare a facturilor și cheltuielilor. Colectarea, procesarea și vizualizarea informațiilor financiare ale utilizatorului într-un mod eficient și accesibil.
2. Soluții alese: construirea unei aplicații web cu ajutorul framework-ului Spring Boot pentru backend și React pentru frontend, care să permită utilizatorului să introducă, să vizualizeze și să gestioneze informațiile despre facturi și cheltuieli. Crearea unei baze de date pentru stocarea acestor informații.
3. Rezultate obținute: un sistem flexibil și robust care permite utilizatorilor să își gestioneze facturile și cheltuielile într-un mod eficient și accesibil. Sistemul oferă o imagine clară asupra istoricului financiar al utilizatorului și îi ajută să gestioneze mai bine resursele financiare.
4. Testări și verificări: sistemul a fost testat manual pentru a acoperi fiecare funcționalitate prezentată și pentru a asigura funcționarea completă și corectă.
5. Contribuții personale: crearea și implementarea structurii generale a aplicației, inclusiv design-ul interfeței utilizator, dezvoltarea logicii de business din backend, configurarea bazei de date și implementarea funcționalităților principale ale aplicației.
6. Surse de documentare: articole, cărți, internet.

MINISTERUL EDUCAȚIEI



UNIVERSITATEA TEHNICĂ

DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Semnătura autorului

Semnătura conducătorului științific

Cuprins

1	INTRODUCERE.....	4
1.1	CONTEXT GENERAL	4
1.2	OBJECTIVE.....	4
1.3	SPECIFICAȚII	5
1.3.1	Autentificarea in aplicație.....	5
1.3.2	Crearea de noi utilizatori	6
1.3.3	Vizualizarea facturilor de către administrator.....	6
1.3.4	Vizualizarea cheltuielilor de către administrator	7
1.3.5	Crearea de cheltuieli și facturi de către utilizator	7
1.3.6	Vizualizarea istoricului cheltuielilor și facturilor de către utilizator	7
1.3.7	Marcarea cheltuielilor și facturilor ca fiind plătite.....	7
1.3.8	Vizualizarea evoluției facturilor plătite/neplătite	7
1.3.9	Vizualizarea unui tabel cu utilizatori de către administrator.....	7
1.3.10	Descărcarea atașamentelor din cheltuieli.....	8
2	STUDIU BIBLIOGRAFIC.....	8
2.1	ARHITECTURA REST.....	8
2.2	MODEL-VIEW-CONTROLLER (MVC)	9
2.3	COMBINAREA REST SI MVC	10
2.4	JAVA	11
2.4.1	Introducere	11
2.4.2	Designul limbajului Java	11
2.4.3	Mașina Virtuală Java (JVM)	11
2.4.3.1	Portabilitate și independență de platformă	11
2.4.3.2	Securitate	12
2.4.3.3	Performanță	12
2.4.3.4	Concluzie	12
2.4.4	Garbage Collector	12
2.4.4.1	Introducere	12
2.4.4.2	Funcționarea colectării gunoiului	12
2.4.4.3	Tipuri de colectoare de gunoi în JVM	13
2.4.4.4	Impactul colectării gunoiului asupra performanței	13
2.4.4.5	Concluzie	13
2.4.5	Java Development Kit și Java Runtime Environment	14
2.4.5.1	Introducere	14
2.4.5.2	Java Development Kit (JDK).....	14
2.4.5.3	Java Runtime Environment (JRE).....	14
2.4.5.4	Concluzie	14
2.5	HTML și CSS	15
2.5.1	Introducere	15
2.5.2	HTML: Definirea structurii unei pagini web	15
2.5.3	CSS: Controlul prezentării și aspectului paginilor web	15
2.5.4	HTML și CSS: Cum lucrează împreună.....	15
2.5.5	Concluzie.....	16
2.6	JAVASCRIPT	16
2.6.1	Introducere	16
2.6.2	Javascript in browser	16
2.6.3	Javascript pe server.....	16

2.6.4	Node Package Manager (npm)	17
2.6.5	Biblioteci și cadre de lucru JavaScript	17
2.6.6	Concluzie	17
2.7	SQL	17
2.7.1	Introducere	17
2.7.2	Structura limbajului SQL	17
2.7.3	SQL în practică: Sisteme de gestionare a bazelor de date	18
2.7.4	SQL și modelarea relațională a datelor	18
2.7.5	SQL în era Big Data	18
2.7.6	Concluzie	18
3	ANALIZĂ ȘI FUNDAMENTARE TEORETICĂ	19
3.1	SPRING	19
3.1.1	Arhitectura Spring	19
3.1.2	Spring Boot	20
3.2	MAVEN	21
3.2.1	Arhitectura Maven	21
3.2.2	Funcționalitatea Maven	21
3.2.3	Integrarea Maven cu alte instrumente	22
3.3	REACT	23
3.3.1	Introducere in React	23
3.3.2	Arhitectura React	23
3.3.3	Funcționalitatea React	24
3.3.4	Exemplu de utilizare a React	24
3.4	POSTGRESQL	25
3.4.1	Descrierea PostgreSQL	25
3.4.2	Arhitectura PostgreSQL	25
3.5	INTELLIJ IDEA	25
3.5.1	Descrierea IntelliJ IDEA	25
3.5.2	Funcționalitatea IntelliJ IDEA	25
3.6	WEBSTORM	26
3.6.1	Descrierea WebStorm	26
3.6.2	Funcționalitatea WebStorm	26
3.7	POSTMAN	26
3.7.1	Descrierea Postman	26
3.7.2	Funcționalitatea Postman	27
4	PROIECTARE SI IMPLEMENTARE	27
4.1	BACKEND	27
4.1.1	Sistemul de logare	27
4.1.1.1	AuthController și procesul de autentificare	27
4.1.1.2	JWTAuthenticationFilter și procesarea cererilor autentificate	29
4.1.1.3	JwtAuthEntryPoint și gestionarea erorilor de autentificare	29
4.1.1.4	Configurarea lanțului de filtre	30
4.1.1.5	UserDetailsService, UserEntity si roluri	32
4.1.2	Creearea de utilizatori	33
4.1.3	Crearea de facturi	34
4.1.4	Istoricul facturilor pe tip	35
4.1.5	Crearea de cheltuieli	36
4.1.6	Istoricul cheltuielilor	37
4.1.7	Pagina principala	38
4.1.7.1	Secțiunea de facturi	38
4.1.7.2	Secțiunea de cheltuieli	39

4.1.8	Descărcarea atașamentelor.....	39
4.2	FRONTEND	40
4.2.1	Sistemul de logare	40
4.2.2	Crearea de utilizatori	41
4.2.3	Crearea de facturi	43
4.2.4	Istoricul facturilor pe tip.....	45
4.2.5	Crearea de cheltuieli	46
4.2.6	Istoricul cheltuielilor.....	48
4.2.7	Pagina principala	49
4.2.7.1	Secțiunea de facturi	49
4.2.7.2	Secțiunea de cheltuieli	50
4.2.8	Descarcarea atasamentelor.....	51
4.3	BAZA DE DATE.....	52
5	CONCLUZII.....	54
5.1	REZULTATE OBȚINUTE.....	54
5.2	DIRECȚII DE DEZVOLTARE.....	54
6	BIBLIOGRAFIE.....	55

Figura 2.1	Arhitectura REST [2].....	9
Figura 2.2	Model-View-Controller architecture [4]	10
Figura 3.1	Arhitectura Spring [14].....	19
Figura 3.2	Arhitectura Spring Boot [16]	20
Figura 3.3	Arhitectura React [22].....	23
Figura 4.1	Metoda login()	27
Figura 4.2	Metoda generateToken()	28
Figura 4.3	Metoda doFilterInternal()	29
Figura 4.4	Componenta JwtAuthEntryPoint	30
Figura 4.5	Configurarea lanțului de filtre.....	31
Figura 4.6	Metoda loadUserByUsername()	32
Figura 4.7	Metoda maprolesToAuthorities().....	32
Figura 4.8	Metoda create() din BillService	34
Figura 4.9	Cererea de logare	40
Figura 4.10	Formularul de logare	41
Figura 4.11	Formularul de înregistrare.....	42
Figura 4.12	Exemplu eroare validare	43
Figura 4.13	Exemplu eroare server.....	43
Figura 4.14	Formularul pentru crearea unei noi facturi.....	44
Figura 4.15	Istoricul facturilor de apă TODO	46
Figura 4.16	Formularul pentru crearea unei cheltuieli noi	47
Figura 4.17	Exemplu afișare atașamente	47
Figura 4.18	Istoricul cheltuielilor TODO	48
Figura 4.19	Grafice facturi TODO	49
Figura 4.20	Popup atașamente TODO	51

1 Introducere

1.1 Context general

În lumea digitală rapidă de astăzi, gestionarea eficientă a facturilor și a cheltuielilor este crucială atât pentru organizații cât și pentru indivizi. Un sistem care facilitează monitorizarea și gestionarea acestor facturi și cheltuieli, furnizând în același timp o interfață prietenoasă și intuitivă, este de o importanță majoră. Această lucrare se concentrează pe dezvoltarea unei astfel de aplicații web, care combină un frontend creat cu React și un backend construit cu Java și Spring.

Aplicația dezvoltată este o soluție pentru utilizatorii care caută un mod mai eficient de a-și gestiona facturile și cheltuielile. Ea oferă funcționalități extinse, precum posibilitatea de a crea, vizualiza și marca facturile și cheltuielile ca plătite, dar și posibilitatea de a descărca anexele asociate. Cu două tipuri de utilizatori, administrator și utilizator normal, aplicația satisface atât nevoile de administrare ale unui grup, cât și pe cele individuale.

1.2 Obiective

Scopul principal al acestei aplicații este de a facilita o gestionare eficientă și intuitivă a facturilor și cheltuielilor, promovând astfel o mai bună organizare financiară pentru utilizatori. În plus, aplicația își propune să mențină un echilibru ideal între complexitatea funcționalităților oferite și ușurința de utilizare, asigurând o experiență fluidă și fără probleme pentru toți utilizatorii.

Principalele funcționalități implementate sunt următoarele:

- Autentificarea în aplicație prin intermediul Spring Security, asigurând astfel un nivel adecvat de securitate pentru datele utilizatorilor.
- Crearea de noi utilizatori de către administrator, facilitând gestionarea utilizatorilor în cadrul sistemului.
- Vizualizarea tuturor facturilor de către administrator, ceea ce oferă o imagine de ansamblu asupra activității de facturare în cadrul sistemului.
- Vizualizarea tuturor cheltuielilor de către administrator, permițând un control mai bun asupra finanțelor sistemului.
- Crearea de cheltuieli și facturi de către utilizator, permițând utilizatorilor să mențină o evidență a tuturor tranzacțiilor lor.
- Vizualizarea istoricului cheltuielilor și facturilor de către utilizator, asigurând o transparență și o capacitate de urmărire a tranzacțiilor lor.
- Marcarea cheltuielilor și facturilor ca fiind plătite, ajutând la menținerea unei evidențe clare a plăților efectuate și a celor restante.

- Prezentarea unei vizualizări grafice a evoluției facturilor plătite/neplătite, oferind utilizatorilor o modalitate vizuală intuitivă de a înțelege starea finanțelor lor.

Obiectivul final al acestui proiect este de a crea o aplicație care nu doar că îndeplinește aceste funcționalități, dar o face într-un mod care este ușor de utilizat și înțeles de către utilizatori, indiferent de nivelul lor de experiență tehnică.

1.3 Specificații

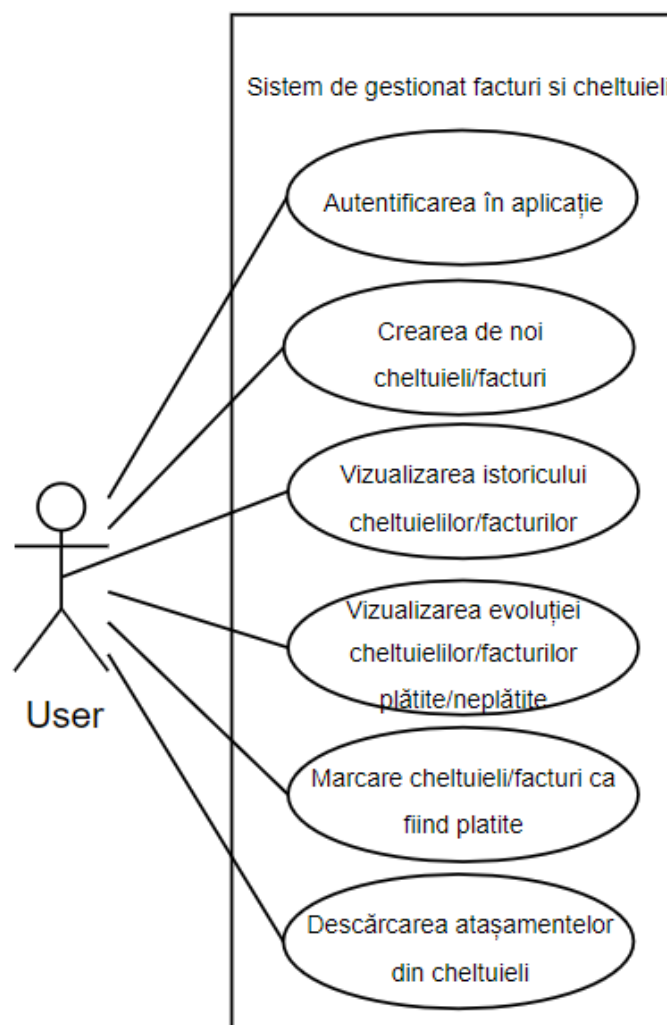


Figura 1.1 Diagrama use-case Utilizator

1.3.1 Autentificarea în aplicație

Logarea în aplicație este realizată prin intermediul tehnologiei Spring Security, oferind astfel un nivel de securitate adecvat pentru protejarea datelor utilizatorilor.

Aceasta se realizează prin trimiterea unei cereri de autentificare către server cu datele de logare. În caz de succes, serverul răspunde cu un token JWT, care se stochează în Session Storage pe partea de frontend și se atașează la fiecare cerere trimisă către server.

1.3.2 Crearea de noi utilizatori

Administratorul are posibilitatea de a crea noi utilizatori, facilitând astfel gestionarea conturilor de utilizatori în cadrul aplicației. După autentificare, administratorul este redirecționat către un panou de administrare unde poate introduce informațiile necesare pentru crearea unui nou utilizator: nume, prenume, nume de utilizator, parolă, confirmarea parolei, adresă și adresă de e-mail.

1.3.3 Vizualizarea facturilor de către administrator

Sistemul oferă posibilitatea administratorului de a vizualiza toate facturile, oferind astfel o perspectivă de ansamblu asupra activității de facturare din cadrul aplicației. Prin accesarea tabului "Bills" din panoul de administrare, administratorul poate vedea lista completă a facturilor, plătite și neplătite.

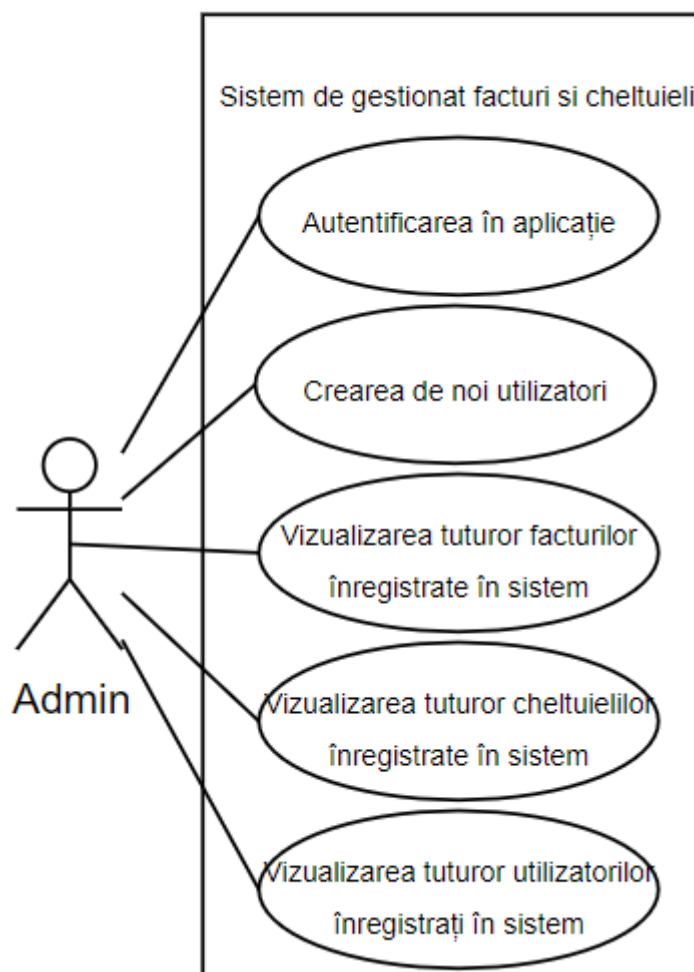


Figura 1.2 *Diagrama use-case Administrator*

1.3.4 Vizualizarea cheltuielilor de către administrator

De asemenea, administratorul are posibilitatea de a vizualiza toate cheltuielile înregistrate în sistem. Acest lucru permite un control mai eficient asupra finanțelor și un management optim al resurselor financiare ale aplicației.

1.3.5 Crearea de cheltuieli și facturi de către utilizator

Utilizatorul are posibilitatea de a înregistra atât cheltuieli, cât și facturi în cadrul aplicației, păstrând astfel o evidență clară a tuturor tranzacțiilor sale financiare. Acest lucru se face prin intermediul taburilor "Expenses" și "Bills", unde pot fi adăugate detalii precum tipul cheltuielii, suma, data și eventualele note atașate.

1.3.6 Vizualizarea istoricului cheltuielilor și facturilor de către utilizator

Utilizatorii au acces la istoricul tranzacțiilor lor prin intermediul aceleiași secțiuni "Expenses" și "Bills". Acest lucru permite o vizualizare transparentă și facilă a tuturor tranzacțiilor financiare, îmbunătățind astfel capacitatea de urmărire și gestionare a finanțelor personale.

1.3.7 Marcarea cheltuielilor și facturilor ca fiind plătite

Atât facturile, cât și cheltuielile pot fi marcate ca fiind plătite de către utilizator, contribuind la menținerea unei evidențe clare a tranzacțiilor efectuate și a celor care sunt încă restante.

1.3.8 Vizualizarea evoluției facturilor plătite/neplătite

Pentru a facilita înțelegerea stării finanțelor, utilizatorii au la dispoziție o reprezentare grafică a evoluției facturilor plătite și neplătite. Acest feature permite vizualizarea rapidă a tendințelor și a comportamentului de plată, ajutând la planificarea eficientă a cheltuielilor viitoare.

1.3.9 Vizualizarea unui tabel cu utilizatori de către administrator

Aplicația include o funcționalitate prin care administratorul poate vizualiza un tabel cu toți utilizatorii. Acesta include detalii precum numele, prenumele, numele de utilizator, e-mailul și data creării contului, oferind astfel o privire de ansamblu asupra comunității de utilizatori și facilitând gestionarea eficientă a acestora.

1.3.10 Descărcarea atașamentelor din cheltuieli

Funcționalitatea de descărcare a atașamentelor din cheltuieli este un alt aspect important al aplicației. Utilizatorii pot atașa documente la fiecare cheltuială înregistrată (cum ar fi facturi, chitanțe etc.) și pot descărca aceste documente ulterior. Acest lucru ajută la păstrarea unui istoric al documentelor legate de cheltuieli și la menținerea unui control strict asupra finanțelor personale.

2 Studiu bibliografic

2.1 Arhitectura REST

Representational State Transfer (REST) este un stil de arhitectură software pentru a proiecta sisteme scalabile, care a fost introdus în teza doctorală a lui Roy Fielding în anul 2000 [1]. De atunci, acesta a devenit standardul de facto pentru proiectarea API-urilor web, datorită simplității și eficienței sale.

Unul dintre principiile de bază ale REST este separarea între client și server. Aceasta înseamnă că interfața client este separată de stocarea datelor (backend), astfel încât fiecare dintre acestea poate fi dezvoltată independent una de cealaltă [1]. Acest aspect este esențial pentru scalabilitate, deoarece permite serverului să se dezvolte și să se adapteze în funcție de cerințele în continuă schimbare ale clienților, fără a afecta funcționarea acestora [1].

REST se bazează și pe un concept numit stateless, ceea ce înseamnă că fiecare cerere de la client la server trebuie să conțină toate informațiile necesare pentru a procesa cererea respectivă [1]. Asta înseamnă că serverul nu trebuie să mențină o evidență a stării fiecărui client între cereri, ceea ce ar putea fi o sarcină care consumă multe resurse pe măsură ce numărul de clienți crește. Astfel, stateless contribuie la eficiența și scalabilitatea sistemului [1].

În plus, REST impune o interfață uniformă pentru a asigura interoperabilitatea între aplicații. Aceasta înseamnă că toate cererile și răspunsurile urmează același format, indiferent de tipul de resurse cu care lucrează [1]. În practică, în cazul HTTP, aceasta implică utilizarea metodelor standard HTTP (GET, POST, PUT, DELETE etc.) pentru a realiza operațiile dorite. Astfel, este asigurată o coerență și predictibilitate în design, simplificând dezvoltarea și utilizarea sistemului [1].

Un alt aspect cheie al arhitecturii REST este că este orientată spre resurse. Acest lucru înseamnă că fiecare piesă de informație în sistem este considerată o "resursă", care poate fi accesată și manipulată prin intermediul API-ului [1]. Aceste resurse sunt identificate în mod unic prin URL-uri, ceea ce facilitează localizarea și manipularea lor în cadrul sistemului [1]. Acesta este un aspect central al designului RESTful și îmbunătățește modularitatea și reutilizarea codului.

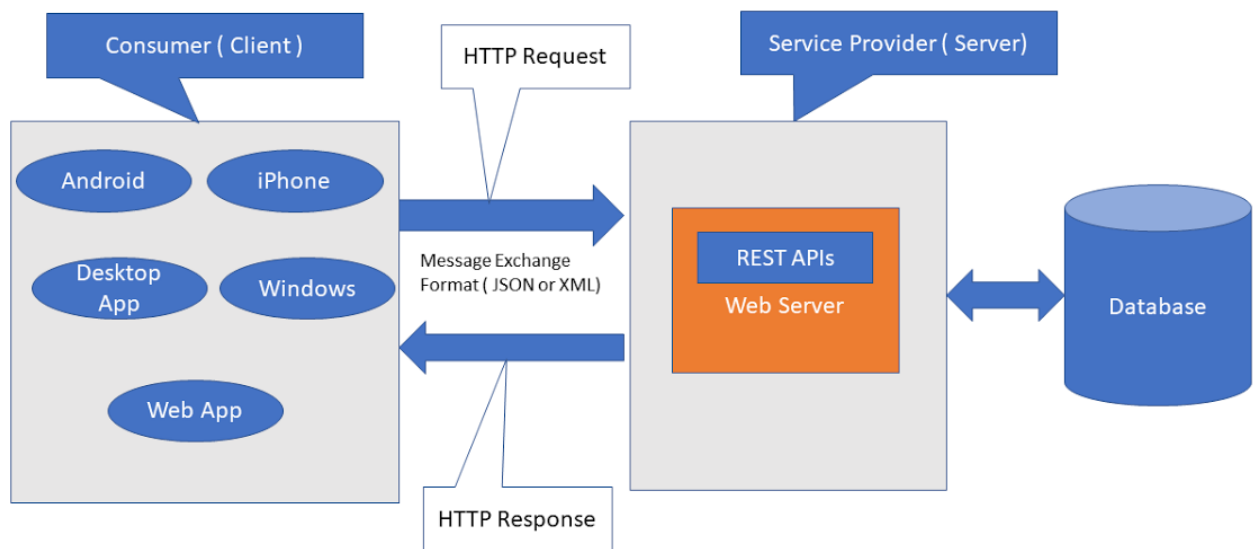


Figura 2.1 Arhitectura REST [2]

2.2 Model-View-Controller (MVC)

Model-View-Controller (MVC) este un pattern de design software pentru dezvoltarea de aplicații care separă logica de afaceri de interfața utilizatorului [3]. Acesta împarte codul în trei componente principale: modelul, vizualizarea și controlerul, fiecare cu roluri specifice.

- Modelul reprezintă structura de date a aplicației și regulile de afaceri. Acesta este "ceea ce este" aplicația - adică, datele și regulile de afaceri ale aplicației [3]. Modelul nu știe nimic despre interfața utilizatorului și este complet separat de vizualizare și controler [3]. Acest lucru are ca rezultat un cod mai curat și mai ușor de menținut, deoarece logica de afaceri este izolată de restul aplicației [3].
- Vizualizarea este reprezentarea vizuală a datelor - "cum arată" aplicația. Acesta poate include orice fel de interfață cu utilizatorul: de la interfețele grafice la răspunsurile de tip text la cererile API [3]. Vizualizarea primește datele de la model și decide cum să le prezinte utilizatorului [3]. Acest lucru permite o flexibilitate mare în modul în care datele pot fi prezentate, deoarece nu există nicio legătură directă între datele în sine și modul în care sunt afișate [3].
- Controlerul este puntea între model și vizualizare. Acesta gestionează interacțiunile cu utilizatorul, procesează inputurile acestora, interacționează cu modelul pentru a reflecta modificările în date și actualizează vizualizarea conform necesarului [3]. Controlerul asigură un flux coordonat între model și vizualizare, prelucrând datele într-o formă care poate fi utilizată eficient de către fiecare componentă [3].

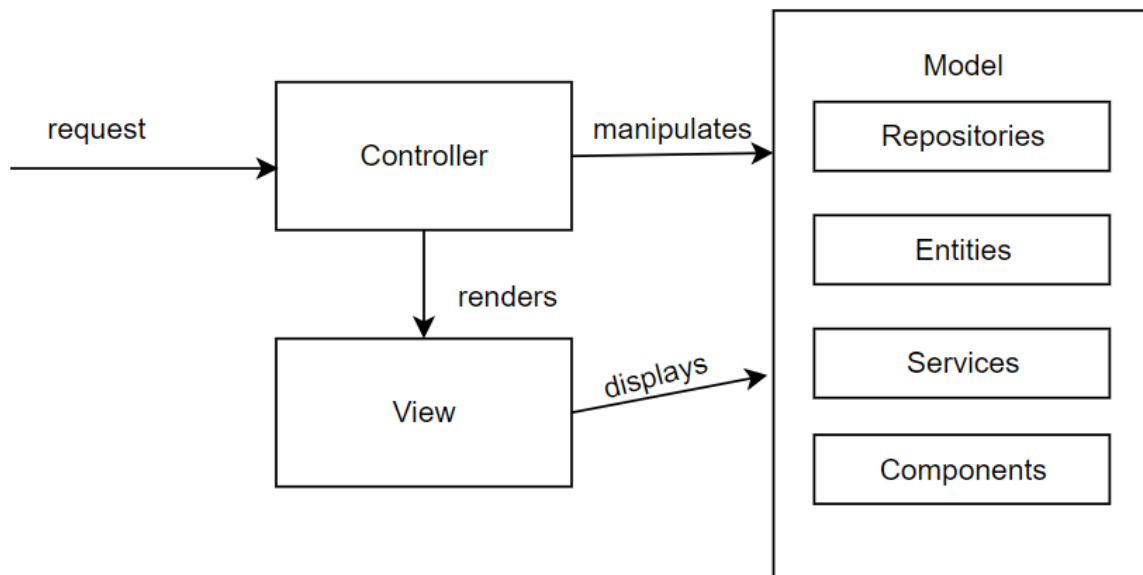


Figura 2.2 Model-View-Controller architecture [4]

2.3 Combinarea REST si MVC

MVC și REST pot fi utilizate împreună pentru a crea aplicații web puternice și eficiente. MVC poate fi folosit pentru a organiza logica aplicației pe partea serverului, în timp ce REST poate fi folosit pentru a gestiona comunicarea între client și server. Această combinație permite dezvoltatorilor să beneficieze de avantajele ambelor arhitecturi [5].

Pentru exemplu, considerăm o aplicație web de gestionare a unei biblioteci. API-ul RESTful al aplicației poate permite clienților să creeze (POST), să citească (GET), să actualizeze (PUT) și să șteargă (DELETE) cărți din bibliotecă, fiecare operație fiind efectuată printr-o cerere HTTP către un URL specific al resursei cărții. Între timp, pe server, logica aplicației ar putea fi organizată conform modelului MVC. Modelul ar putea defini structura de date a unei cărți și regulile pentru manipularea acesteia, vizualizarea ar putea genera răspunsul HTTP care este trimis înapoi la client, iar controlerul ar putea coordona procesul de recepție a cererii de la client, interacțiunea cu modelul și generarea răspunsului prin vizualizare [5].

Deci, combinarea REST cu MVC poate oferi un cadru puternic și flexibil pentru construirea de aplicații web moderne. Aceasta permite dezvoltatorilor să creeze sisteme scalabile, modulare și ușor de întreținut, în timp ce se bucură de avantajele oferite de ambele paradigme [5].

Concluzia este că atât arhitectura REST, cât și Model-View-Controller sunt abordări puternice și flexibile pentru proiectarea de sisteme software. Fiecare are punctele sale forte și poate fi folosit în diferite contexte pentru a satisface diferite cerințe. Prin combinarea lor, dezvoltatorii pot beneficia de avantajele ambelor și pot construi aplicații web scalabile, eficiente și ușor de întreținut.

2.4 Java

2.4.1 Introducere

Java este un limbaj de programare orientat-obiect, puternic tipizat, care a fost lansat de Sun Microsystems în 1995. Încă de la început, Java a fost conceput cu accent pe portabilitate și securitate, fiind adoptat pe scară largă în industrie datorită robusteții și flexibilității sale. În continuare, se va realiza o analiză detaliată a limbajului de programare Java, având în vedere trei aspecte principale: designul limbajului, mașina virtuală Java (JVM) și instrumentele de dezvoltare disponibile [6].

2.4.2 Designul limbajului Java

Java este un limbaj de programare static tipizat, cu un design orientat-obiect (OO). Programarea orientată-obiect implică organizarea codului în "obiecte", care sunt instanțe ale "claselor". Clasele definesc structura și comportamentul obiectelor și permit abstractizarea și încapsularea datelor, ceea ce face codul mai ușor de înțeles și de întreținut [6].

De asemenea, Java oferă suport pentru moștenire, permițând unei clase să "moștenească" proprietăți și comportamente de la o altă clasă, ceea ce permite reutilizarea codului. Java permite moștenirea unică, ceea ce înseamnă că o clasă poate moșteni de la o singură clasă părinte la un moment dat [6].

Java este, de asemenea, cunoscut pentru fiabilitatea sa, datorită verificărilor stricte de tip și a gestionării automate a memoriei. Sistemul de colectare a gunoiului (GC) al Java eliberează automat memoria ocupată de obiecte care nu mai sunt utilizate, eliminând necesitatea gestionării manuale a memoriei și reducând riscul de erori de memorie, cum ar fi scurgerile de memorie [6].

2.4.3 Mașina Virtuală Java (JVM)

Java Virtual Machine (JVM) este inima infrastructurii tehnologice Java. Ea servește ca mașinărie esențială care interpretează și rulează codul de octeți Java, ceea ce permite rularea programelor Java pe diferite tipuri de hardware și sisteme de operare. Acest lucru este posibil datorită abstracției pe care o oferă JVM, îndepărtând complexitatea interacțiunii directe cu sistemul de operare și hardware-ul [7].

2.4.3.1 Portabilitate și independență de platformă

Unul dintre principalele avantaje ale JVM este portabilitatea, care provine din mantra "scrie o dată, rulează oriunde" a Java. După compilarea codului sursă Java, acesta este transformat într-un cod de octeți, care este independent de platformă și poate fi rulat pe orice dispozitiv care are o JVM instalată. Acesta nu este legat de niciun sistem de operare sau arhitectură hardware specifică, ceea ce îi permite să ruleze pe o gamă largă de dispozitive, de la servere puternice la dispozitive mobile sau sisteme înglobate [7].

2.4.3.2 Securitate

JVM joacă, de asemenea, un rol esențial în oferirea unui mediu de rulare sigur pentru codul Java. Java Virtual Machine rulează codul Java într-un mediu de nisip (sandbox), izolându-l de restul sistemului. Aceasta previne executarea unor operațiuni care ar putea prejudicia sistemul sau utilizatorii acestuia, cum ar fi accesul neautorizat la resursele sistemului sau modificarea fișierelor sistemului. JVM dispune și de un verificador de cod de octeți care analizează codul de octeți înainte de a-l executa, asigurându-se că respectă toate regulile limbajului Java și că nu există comportamente nesigure [7].

2.4.3.3 Performanță

În ciuda abstracției pe care o oferă, JVM este proiectată pentru a asigura o performanță excelentă a programelor Java. JVM implementează tehnici avansate, precum compilarea Just-In-Time (JIT), care traduce codul de octeți în cod nativ pentru sistemul de operare la rulare, asigurând o performanță ridicată. De asemenea, JVM se ocupă de gestionarea memoriei pentru aplicații, folosind un algoritm de colectare a gunoiului (garbage collection) pentru a elibera automat memoria care nu mai este necesară [7].

2.4.3.4 Concluzie

În concluzie, Java Virtual Machine este o componentă fundamentală a ecosistemului Java, oferind portabilitate, securitate și performanță aplicațiilor Java, indiferent de platforma pe care sunt rulate. Prin abstractizarea complexităților sistemului de operare și ale hardware-ului, JVM permite dezvoltatorilor să se concentreze pe construirea propriilor aplicații, fără a-și face griji despre aspectele specifice ale platformei pe care va fi rulat codul [7].

2.4.4 Garbage Collector

2.4.4.1 Introducere

Colectarea gunoiului, cunoscută și sub numele de Garbage Collection (GC), este un aspect crucial al mediului de execuție al Java, Java Virtual Machine (JVM). Conceptul a fost implementat încă de la început în Java, cu scopul de a ușura dezvoltarea prin gestionarea automată a memoriei. Astfel, colectarea gunoiului reprezintă o componentă esențială care diferă Java de alte limbaje de programare, precum C și C++, care necesită o gestionare explicită a memoriei de către programator [8].

2.4.4.2 Funcționarea colectării gunoiului

În limbajul de programare Java, atunci când se creează noi obiecte, JVM alocă memorie pentru acestea dintr-o zonă de memorie numită "heap". Acest proces de alocare a memoriei continuă pe măsură ce aplicația rulează și creează obiecte. Însă, după un timp, unii dintre acești obiecti devin inutili. De exemplu, atunci când nu există

referințe către un obiect în program, acesta nu mai poate fi accesat sau utilizat. În acest moment, obiectul devine eligibil pentru colectarea gunoiului, ceea ce înseamnă că memoria ocupată de obiect poate fi acum eliberată și reutilizată pentru alte obiecte [8].

Această eliberare a memoriei este realizată de procesul de colectare a gunoiului, care este o componentă a JVM. Acest proces se execută în fundal, fără intervenția explicită a programatorului. Este responsabilitatea collectorului de gunoi să identifice obiectele care au devenit inutile și să elibereze memoria pe care acestea o ocupă. Acest proces se desfășoară în mai multe etape, inclusiv marcarea obiectelor care sunt încă accesibile și cele care nu sunt, precum și eliminarea celor din urmă. În acest fel, colectarea gunoiului ajută la menținerea unei funcționări eficiente a programelor Java prin eliberarea memoriei care nu mai este necesară [9].

2.4.4.3 Tipuri de colectoare de gunoi în JVM

JVM dispune de mai multe tipuri de colectoare de gunoi, fiecare cu caracteristicile sale specifice, care sunt concepute pentru a se potrivi diferitelor tipuri de aplicații. De exemplu, Serial Collector este un collector de gunoi care funcționează cel mai bine pentru aplicațiile cu cerințe reduse de memorie și procesor, cum ar fi cele care rulează pe computere cu un singur procesor. Parallel Collector, pe de altă parte, este destinat aplicațiilor multithread care rulează pe sisteme multiprocesor.

Avem de asemenea și Concurrent Mark Sweep (CMS) Collector, care este conceput pentru aplicațiile care necesită timpi de răspuns scurți, și G1 Garbage Collector, care este destinat aplicațiilor de dimensiuni mari care necesită o gestionare eficientă a memoriei [8].

2.4.4.4 Impactul colectării gunoiului asupra performanței

Performanța unei aplicații Java poate fi influențată în mod semnificativ de modul în care este gestionată colectarea gunoiului. Deși aceasta contribuie la prevenirea scurgerilor de memorie și simplifică dezvoltarea, poate provoca și pauze în execuția aplicației atunci când collectorul de gunoi își face treaba. Aceste pauze pot afecta negativ performanța aplicației, în special în cazul aplicațiilor cu cerințe stricte de timp real.

Pentru a optimiza performanța, dezvoltatorii pot alege tipul de collector de gunoi care este cel mai potrivit pentru cerințele lor specifice, și pot ajusta parametrii de configurare ai collectorului de gunoi. Cu toate acestea, acestea sunt activități complexe care necesită o înțelegere aprofundată a modului în care funcționează colectarea gunoiului în Java [9].

2.4.4.5 Concluzie

Prin urmare, colectarea gunoiului în Java reprezintă un mecanism sofisticat care joacă un rol esențial în gestionarea eficientă a memoriei în aplicațiile Java. Înțelegerea acestui proces și a modului în care poate fi optimizat este o componentă esențială a dezvoltării aplicațiilor Java de înaltă performanță.

2.4.5 Java Development Kit și Java Runtime Environment

2.4.5.1 Introducere

Java este un limbaj de programare extrem de popular, cunoscut pentru portabilitatea sa, ceea ce înseamnă că o aplicație scrisă în Java poate rula pe o varietate de platforme hardware și software. Aceasta este o caracteristică importantă, deoarece este esențial pentru dezvoltatori să poată crea aplicații care să funcționeze pe cât mai multe dispozitive și sisteme de operare posibile. Această portabilitate se datorează, în mare parte, a două componente cheie ale platformei Java: Java Development Kit (JDK) și Java Runtime Environment (JRE) [10].

2.4.5.2 Java Development Kit (JDK)

JDK este setul complet de instrumente software necesare pentru a dezvolta aplicații în Java. Acesta include JRE, dar și alte instrumente precum compilatorul Java (javac), arhivarul Java (jar), și documentația Java (javadoc), printre altele. Compilatorul Java este cel care transformă codul sursă scris în Java în cod de mașină (bytecode) care poate fi interpretat și executat de JVM. Arhivarul Java este folosit pentru a împacheta fișierele de clasă și resursele asociate într-un singur fișier JAR (Java Archive), pentru a facilita distribuția aplicației. Instrumentul javadoc generează documentația aplicației direct din comentariile din codul sursă [10].

JDK este esențial pentru dezvoltatori, deoarece oferă instrumentele necesare pentru a scrie, testa și depana codul Java. Fără JDK, dezvoltarea de aplicații Java nu ar fi posibilă.

2.4.5.3 Java Runtime Environment (JRE)

Pe de altă parte, JRE este componenta care permite rularea aplicațiilor Java. Aceasta include JVM și bibliotecile de clasă Java standard, care sunt necesare pentru a executa aplicații Java. JVM este motorul care interpretează și execută bytecode-ul generat de compilatorul Java, în timp ce bibliotecile de clasă Java oferă un set extins de funcționalități predefinite pe care dezvoltatorii le pot folosi în codul lor [11].

JRE este necesar pe orice dispozitiv pe care se dorește rularea unei aplicații Java. Fie că este vorba de un server web care găzduiește o aplicație web Java, un smartphone care rulează o aplicație Android, sau un calculator personal care rulează o aplicație desktop Java, toate acestea necesită JRE.

2.4.5.4 Concluzie

În concluzie, JDK și JRE sunt două componente esențiale ale platformei Java. Fără JDK, dezvoltarea de aplicații Java nu ar fi posibilă, iar fără JRE, aceste aplicații nu ar putea fi rulate. Împreună, ele fac posibilă dezvoltarea și rularea de aplicații Java pe o varietate de platforme, ceea ce contribuie la popularitatea continuă a Java ca limbaj de programare.

2.5 HTML și CSS

2.5.1 Introducere

HyperText Markup Language (HTML) și Cascading Style Sheets (CSS) sunt tehnologii esențiale în construirea paginilor web. HTML oferă structura paginilor web, în timp ce CSS se ocupă de prezentarea și aspectul acestora. Ambele tehnologii sunt necesare pentru a crea pagini web funcționale și vizual atractive. Ele funcționează împreună pentru a crea și a stiliza conținutul pe care îl vedem pe web [12].

2.5.2 HTML: Definirea structurii unei pagini web

HTML este un limbaj de marcare ce stă la baza oricărei pagini web. Fiecare element al unei pagini web este definit de o anumită etichetă HTML. Aceste etichete oferă instrucțiuni browserului despre cum să formateze și să afișeze conținutul. De exemplu, etichetele de titlu, cum ar fi <h1>, <h2>, <h3>, etc., sunt folosite pentru a defini titlurile și subtitlurile, iar paragrafele sunt definite de eticheta <p> [12].

Pe lângă acestea, există și etichete HTML pentru a defini linkurile (<a>), imaginile (), listele (, ,), tabelele (<table>, <tr>, <td>), etc. Fiecare etichetă HTML are o anumită funcție și utilizare, care contribuie la structura generală a paginii web [12].

2.5.3 CSS: Controlul prezentării și aspectului paginilor web

În timp ce HTML se ocupă de structura paginii web, CSS este folosit pentru a controla modul în care aceasta arată. CSS controlează aspectul vizual al paginii, cum ar fi designul layout-ului, culorile, fonturile, spațierea, marginile și alinierea [13].

Un fișier CSS constă într-o serie de reguli. Fiecare regulă are un selector și o declarație. Selectorul indică elementul HTML la care se aplică regula, în timp ce declarația definește ce stil va fi aplicat. Declarația este formată dintr-o proprietate și o valoare. De exemplu, în regula `h1 {color: red;}`, selectorul este `h1`, proprietatea este `color`, iar valoarea este `red` [13].

2.5.4 HTML și CSS: Cum lucrează împreună

HTML și CSS lucrează împreună pentru a crea o pagină web completă. HTML este folosit pentru a defini și a marca structura paginii, în timp ce CSS este folosit pentru a controla aspectul și prezentarea acesteia.

Există trei metode prin care CSS poate fi aplicat elementelor HTML: inline, intern și extern. Stilizarea inline se face direct în eticheta HTML folosind atributul `style`. Metoda internă implică includerea regulilor CSS în secțiunea <head> a documentului HTML folosind eticheta <style>. Metoda externă, cea mai comună și eficientă din punct de vedere al gestionării codului, implică legarea unui fișier CSS extern la documentul HTML prin intermediul etichetei <link> [13].

2.5.5 Concluzie

HTML și CSS sunt tehnologii esențiale pentru dezvoltarea web. HTML oferă structura și semnificația conținutului unei pagini web, în timp ce CSS controlează prezentarea vizuală a acestui conținut. Împreună, ele permit dezvoltatorilor să creeze pagini web care sunt atât funcționale, cât și vizual atractive.

2.6 Javascript

2.6.1 Introducere

JavaScript este un limbaj de programare puternic, interpretat, care a luat naștere din nevoia de a adăuga dinamică paginilor web într-o perioadă în care internetul începea să devină din ce în ce mai omniprezent. Limbajul a fost inițial dezvoltat de Brendan Eich în timp ce lucra pentru Netscape Communications Corporation în 1995, și deși a trecut prin multe iterații și a suferit numeroase modificări, rămâne o componentă centrală a experienței web de astăzi [14].

2.6.2 Javascript in browser

Unul dintre locurile unde JavaScript își arată cel mai bine abilitățile este în cadrul browserului web. Aici, JavaScript controlează comportamentul elementelor HTML și CSS ale unei pagini web, oferind interactivitate și funcționalitate dinamică. Fie că este vorba de crearea unui buton care, atunci când este apăsat, schimbă culoarea fundalului paginii sau de afișarea unui mesaj în urma unei acțiuni a utilizatorului, toate acestea pot fi realizate cu JavaScript. Mai mult, JavaScript este responsabil pentru o mare parte din interacțiunile asincrone care au loc între browser și server, ceea ce face posibilă încărcarea de noi conținuturi fără a reîncărca întreaga pagină - o componentă cheie a ceea ce numim astăzi "web 2.0" [14].

2.6.3 Javascript pe server

Odată cu lansarea platformei Node.js în 2009, a devenit posibilă folosirea JavaScript și pe partea de server. Node.js este un mediu de rulare JavaScript bazat pe motorul V8 al Google Chrome, care permite rularea codului JavaScript pe server. Acest lucru a însemnat că JavaScript nu mai era limitat la a controla doar comportamentul unei pagini web, ci putea fi folosit pentru a construi servere web, a accesa baze de date și a realiza alte operații care erau anterior rezervate limbajelor de programare server-side. Acest pas a transformat JavaScript dintr-un limbaj pur de front-end într-un limbaj de full-stack, sporind semnificativ utilitatea și popularitatea acestuia [15].

2.6.4 Node Package Manager (npm)

O altă componentă importantă a ecosistemului JavaScript este npm (Node Package Manager). Lansat în 2010, npm este un manager de pachete pentru JavaScript, care permite dezvoltatorilor să împărtășească și să utilizeze codul scris de alții. npm facilitează instalarea, actualizarea și gestionarea dependențelor pentru proiecte JavaScript. În prezent, npm găzduiește sute de mii de pachete, acoperind o gamă largă de funcționalități, de la utilitare simple până la biblioteci complexe și cadre de lucru [16].

2.6.5 Biblioteci și cadre de lucru JavaScript

JavaScript beneficiază de o mulțime de biblioteci și cadre de lucru (frameworks) care simplifică dezvoltarea de aplicații complexe. De la jQuery, care a simplificat manipularea elementelor HTML și CSS, la cadre de lucru complexe precum Angular.js, React.js și Vue.js, care permit crearea de interfețe de utilizator declarative și bazate pe componente, aceste instrumente au crescut în complexitate și capacitate împreună cu limbajul însuși. Fiecare dintre acestea aduce abordări unice în rezolvarea problemelor comune, ceea ce duce la o diversitate de stiluri și abordări în dezvoltarea cu JavaScript [14].

2.6.6 Concluzie

JavaScript este un limbaj de programare extrem de versatil și puternic, care joacă un rol fundamental în dezvoltarea web modernă. Prin capacitatea sa de a controla comportamentul dinamic al unei pagini web și de a permite dezvoltarea server-side, JavaScript a transformat modul în care interacționăm cu web-ul. Cu ajutorul bibliotecilor și cadrelor de lucru, dezvoltatorii au posibilitatea de a crea experiențe bogate și interactive pe web, iar cu npm, pot beneficia de munca altora, sporind productivitatea și promovând colaborarea.

2.7 SQL

2.7.1 Introducere

SQL (Structured Query Language) este un limbaj de programare conceput pentru gestionarea și manipularea bazelor de date relaționale. A fost dezvoltat în anii 1970 de către IBM, având la bază teoria relațională a bazelor de date a lui E.F. Codd. De la lansarea sa, SQL a devenit limbajul standard pentru lucrul cu baze de date, fiind utilizat pe scară largă în industrie și în cercetare [17].

2.7.2 Structura limbajului SQL

Limbajul SQL este compus din mai multe sub-limbaje, fiecare având un scop specific. Acestea includ DDL (Data Definition Language) pentru definirea și modificarea schemelor bazei de date, DML (Data Manipulation Language) pentru inserarea,

modificarea și ștergerea datelor, și DQL (Data Query Language) pentru interogarea și extragerea datelor. Fiecare dintre aceste sub-limbaje conține un set de comenzi care permit manipularea bazelor de date la diferite niveluri de abstracție [17].

2.7.3 SQL în practică: Sisteme de gestionare a bazelor de date

SQL este utilizat în cadrul sistemelor de gestionare a bazelor de date (DBMS) relaționale, cum ar fi MySQL, Oracle, SQL Server, PostgreSQL și multe altele. Acestea oferă interfețe pentru limbajul SQL, permițând dezvoltatorilor să creeze, interogheze și manipuleze baze de date. Fiecare dintre aceste sisteme poate avea propriile extensii sau variații ale standardului SQL, însă toate se bazează pe același set core de funcționalități, asigurându-se că competențele dobândite în SQL sunt transferabile între diferite sisteme [18].

2.7.4 SQL și modelarea relațională a datelor

Conceptul central al SQL și al bazelor de date relaționale este ideea de modelare a datelor în forme de tabele sau "relații". Fiecare tabel este format dintr-un set de rânduri ("înregistrări") și coloane ("câmpuri"), fiecare coloană reprezentând un tip de date. Relațiile între tabele se realizează prin "chei primare" și "chei străine", care leagă în mod unic înregistrările dintr-o tabelă cu cele din alta. Aceasta oferă o modalitate eficientă și flexibilă de a reprezenta și a interoga date complexe [17].

2.7.5 SQL în era Big Data

În era Big Data, SQL continuă să joace un rol esențial. Sisteme precum Apache Hive, Google BigQuery și Amazon Redshift extind conceptul de SQL pentru a se potrivi cu paradigma datelor de mare volum, oferind interfețe SQL pentru manipularea și interogarea seturilor masive de date. Aceasta permite dezvoltatorilor să folosească familiaritatea cu SQL în contextul noilor provocări ale Big Data [18].

2.7.6 Concluzie

SQL a avut un impact profund asupra modului în care datele sunt stocate, interogate și manipulate. Prin concentrarea sa pe modelarea relațională a datelor, SQL permite gestionarea eficientă a seturilor de date complexe și interogarea acestora într-un mod intuitiv și puternic. În era digitală, competența în SQL este o abilitate esențială pentru dezvoltatorii de software, analiștii de date și o varietate de alte roluri.

3 Analiză și fundamentare teoretică

3.1 Spring

3.1.1 Arhitectura Spring

Spring Framework este un cadru de lucru open-source pentru dezvoltarea de aplicații Java enterprise. A fost creat pentru a aborda complexitatea dezvoltării aplicațiilor enterprise și este cunoscut pentru abordarea sa "lightweight" și pentru infrastructura inversă a controlului (IoC) [13]. Framework-ul Spring a fost conceput pentru a facilita dezvoltarea și testarea de aplicații Java, oferind servicii de infrastructură de bază și module pentru dezvoltarea de aplicații pe platforma Java EE.

Arhitectura Spring este modulară, ceea ce înseamnă că dezvoltatorii pot alege să utilizeze numai părțile de care au nevoie. Framework-ul este compus dintr-o serie de module care oferă o gamă largă de servicii. Printre acestea se numără injectarea dependențelor (DI), aspect-oriented programming (AOP), managementul tranzacțiilor, accesul la date și multe altele [13]. Spring oferă, de asemenea, integrare cu alte tehnologii, cum ar fi Hibernate, JPA, JMS și JNDI, pentru a numi doar câteva.

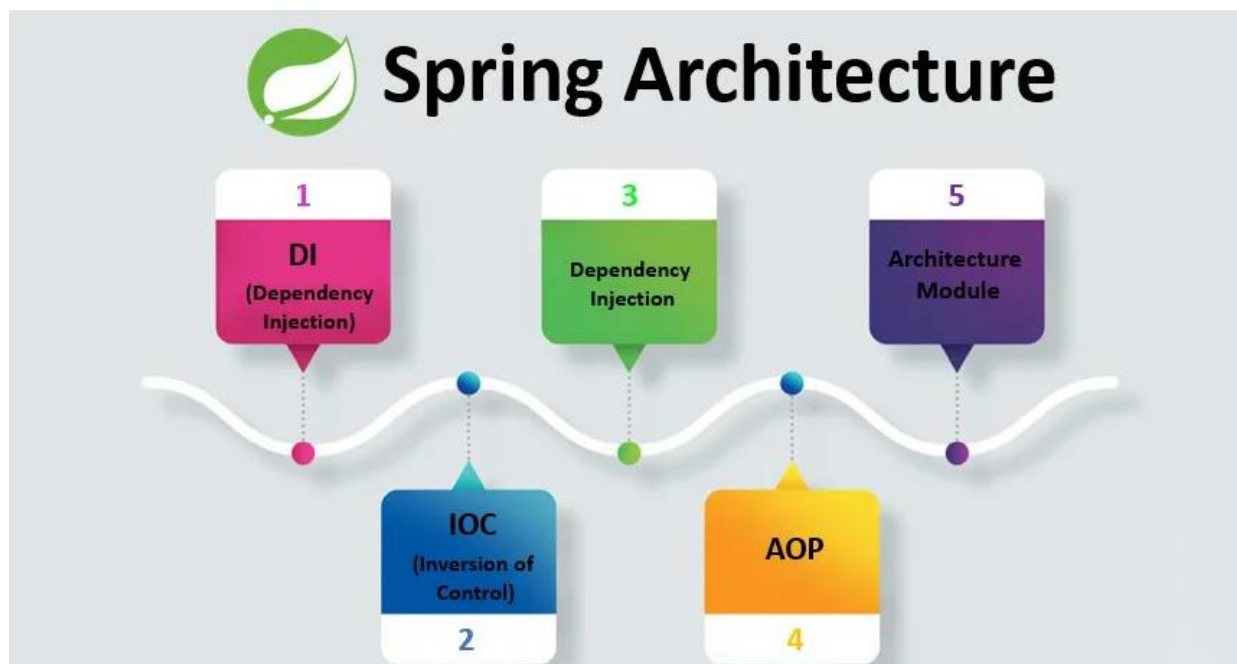


Figura 3.1 Arhitectura Spring [14]

Arhitectura Spring este centrată în jurul principiului Inversării Controlului (IoC), care este fundamental pentru acest framework. Principiul IoC se referă la procesul prin care obiectele sunt creat, asamblat și gestionat de containerul Spring, în loc să fie gestionat direct de codul aplicației. Acest lucru simplifică dezvoltarea și testarea,

deoarece obiectele sunt gestionate în mod central și pot fi injectate acolo unde sunt necesare, ceea ce îmbunătățește modularitatea și decuplarea codului.

Un alt concept cheie în Spring este Programarea Orientată pe Aspecte (AOP), care separă logica de afaceri a aplicației de problemele de infrastructură, cum ar fi securitatea și gestionarea tranzacțiilor. AOP permite dezvoltatorilor să încapsuleze problemele transversale (cum ar fi jurnalizarea și securitatea) în "aspecte" separate, ceea ce face codul mai ușor de înțeles și de întreținut.

3.1.2 Spring Boot

Spring Boot este un proiect de la Pivotal menit să simplifice crearea de aplicații Spring. Scopul său principal este de a reduce complexitatea de configurare a unei aplicații Spring, oferind un set de "opinii" despre modul în care ar trebui să fie configurată o aplicație Spring [15]. Acest lucru înseamnă că, în loc să trebuiască să configurăm manual o mulțime de componente și servicii, putem lăsa Spring Boot să facă majoritatea lucrului pentru noi.

Unul dintre principalele avantaje ale Spring Boot este "auto-configurarea". Acest lucru înseamnă că, pe baza dependențelor pe care le avem în fișierul nostru de proiect, Spring Boot va face niște presupuneri rezonabile despre ceea ce dorim să facem și va configura aplicația în consecință [15]. De exemplu, dacă avem o dependență de Spring MVC și o dependență de Tomcat în proiectul nostru, Spring Boot va presupune că dorim să creăm o aplicație web și va configura Tomcat ca server embedded.

Spring Boot vine, de asemenea, cu o serie de starter POMs (Project Object Models), care sunt un fel de șabloane de proiect. Acestea includ un set de dependențe care sunt, de obicei, utile pentru un anumit tip de proiect [15]. De exemplu, dacă dorim să creăm o aplicație web cu Spring MVC și Thymeleaf, putem folosi spring-boot-starter-web pentru a include toate dependențele necesare.

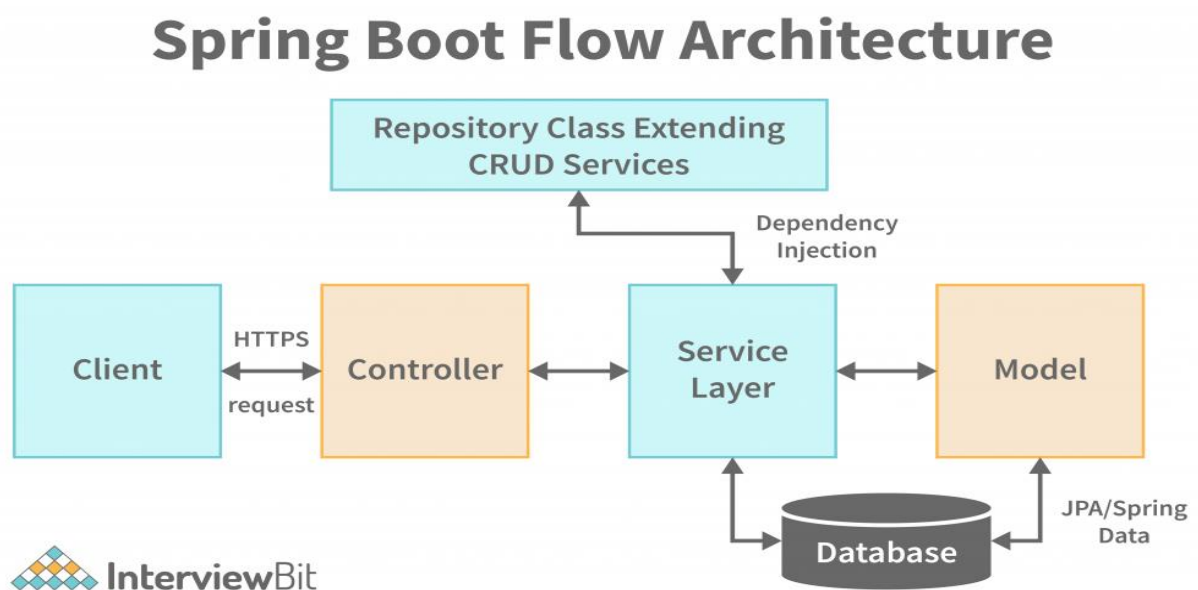


Figura 3.2 Arhitectura Spring Boot [16]

Spring Boot vine, de asemenea, cu un plugin Maven/Gradle care permite crearea de aplicații standalone. Aceste aplicații pot fi rulate ca aplicații Java obișnuite, dar includ un server web embedded, astfel încât nu este nevoie de un server web separat. Acest lucru este util pentru dezvoltare și testare, dar și pentru desfășurarea în producție.

În concluzie, Spring și Spring Boot sunt instrumente puternice pentru dezvoltarea de aplicații enterprise în Java. Ele oferă o mulțime de funcționalități și fac multe din lucrurile "grele", permițând dezvoltatorilor să se concentreze pe logica de afaceri a aplicației în loc de configurare și infrastructură.

3.2 Maven

3.2.1 Arhitectura Maven

Maven este un instrument de gestionare a proiectelor și a înțelegerii proiectelor, folosit primar în proiecte Java [17]. Scopul său principal este de a permite dezvoltatorilor să înțeleagă starea unui proiect în cel mai scurt timp posibil. Pentru a atinge acest scop, Maven se concentrează pe două aspecte: reutilizabilitatea și centralizarea.

Arhitectura Maven este structurată în jurul conceptului de Project Object Model (POM), care descrie software-ul de construit, dependențele sale de alte module și componente externe, și ordinea de construire a elementelor [17]. Acesta permite dezvoltatorilor să gestioneze aspectele legate de construcție, raportare și documentare dintr-o perspectivă centralizată.

Maven folosește un set de convenții pentru a simplifica procesul de construcție a proiectului. În cazul în care aceste convenții corespund cu cerințele proiectului, procesul de configurare a Maven poate fi considerabil simplificat. De exemplu, este standard pentru un proiect Java să aibă o structură de director specifică pentru sursele de cod, resurse și fișiere de testare. Dacă un proiect urmează această structură, Maven nu necesită niciun fel de configurație suplimentară pentru a construi proiectul.



3.2.2 Funcționalitatea Maven

Apache Maven este o unealtă puternică pentru gestionarea proiectelor Java, folosită pentru a simplifica și standardiza procesul de construcție a unui proiect [18]. Acesta este bazat pe conceptul de ciclu de viață al proiectului, care include etape precum curățare, compilare, testare și împachetare.

Una dintre cele mai importante funcționalități ale Maven este gestionarea dependențelor [18]. Cu Maven, dezvoltatorii pot să declare și să gestioneze bibliotecile

de care proiectul depinde într-un singur loc - fișierul pom.xml. Maven își face apoi treaba prin descărcarea acestor dependențe și asigurarea că sunt disponibile pentru proiect atunci când sunt necesare. Acesta oferă, de asemenea, un mod de a gestiona conflictele de dependențe și de a asigura că se utilizează versiunile corecte ale bibliotecilor.

Un alt aspect fundamental al funcționalității Maven este ciclul de viață al construcției [17]. Maven definește un set standard de etape (sau faze) prin care trece un proiect, inclusiv validarea codului sursă, compilarea codului sursă, testarea folosind suita de teste unitare, împachetarea codului compilat într-un format distribuibil (cum ar fi un JAR) și instalarea pachetului în depozitul local. Maven permite, de asemenea, dezvoltatorilor să adauge sau să suprascrie fazele pentru a se potrivi nevoilor specifice ale proiectului lor.

Maven poate fi, de asemenea, utilizat pentru a gestiona proiecte multi-modul [18]. Aceasta este o caracteristică puternică care permite dezvoltatorilor să construiască și să gestioneze proiecte care sunt alcătuite din mai multe module, fiecare având propriul său pom.xml. Maven se asigură că toate modulele sunt construite în ordinea corectă, în funcție de dependențele dintre ele.

Maven este, de asemenea, extensibil, oferind posibilitatea de a adăuga pluginuri pentru a extinde sau modifica funcționalitatea de bază [17]. De exemplu, există pluginuri pentru compilarea codului, testarea unităților, generarea documentației, managementul versiunilor și multe altele. Pluginurile sunt, de asemenea, gestionate ca dependențe în fișierul pom.xml.

În concluzie, funcționalitatea principală a Maven include gestionarea dependențelor, gestionarea ciclului de viață al construcției, suportul pentru proiecte multi-modul și extensibilitatea prin pluginuri. Aceste funcționalități contribuie la eficiența și la standardizarea procesului de construire a proiectelor Java.

3.2.3 Integrarea Maven cu alte instrumente

Maven se integrează foarte bine cu o serie de alte instrumente de dezvoltare, inclusiv sisteme de control al versiunilor, sisteme de integrare continuă, și IDE-uri. De exemplu, Eclipse și IntelliJ IDEA oferă suport pentru Maven prin intermediul unui set de plugin-uri. Acestea permit dezvoltatorilor să importe proiecte Maven, să execute golurile Maven și să lucreze cu fișiere POM direct în IDE [19].

De asemenea, Maven este adesea utilizat în combinație cu sisteme de integrare continuă, cum ar fi Jenkins. Aceasta permite automatizarea procesului de construcție și testare, îmbunătățind calitatea software-ului și eficiența procesului de dezvoltare [19].

În concluzie, Maven este un instrument puternic pentru gestionarea proiectelor Java, oferind o gamă largă de funcționalități care sporesc eficiența și calitatea procesului de dezvoltare.

3.3 React

3.3.1 Introducere in React

Potrivit [20], React este o bibliotecă JavaScript pentru construirea de interfețe de utilizator. A fost dezvoltat de Facebook și este utilizat în prezent pe scară largă în dezvoltarea aplicațiilor web pe partea de frontend. React facilitează crearea de componente interfeței utilizator care sunt reutilizabile și pot fi combinate în diverse moduri pentru a crea o interfață de utilizator complexă.

3.3.2 Arhitectura React

La baza React stă conceptul de componentă. O componentă React este o unitate independentă de cod care controlează o anumită parte a interfeței utilizator. Componentele pot fi combinate și reutilizate în diverse moduri pentru a crea o interfață de utilizator completă [21]. Fiecare componentă are propriul ei "stat" și "proprietăți". Statul reprezintă datele care pot fi modificate în timp, în timp ce proprietățile (sau "props") sunt datele care sunt transmise componentei de la componenta părinte.

Arhitectura React este bazată pe un concept numit "Virtual DOM" [21]. În loc să modifice direct Document Object Model (DOM), React creează o reprezentare în memorie a acestuia, cunoscută sub numele de Virtual DOM. Când statul unei componente se schimbă, React actualizează doar acele părți ale Virtual DOM care sunt afectate de acea schimbare. Apoi, React compară noul Virtual DOM cu versiunea anterioară și face doar modificările minime necesare în DOM real. Acest proces este cunoscut sub numele de "reconciliere" și este motivul pentru care React este atât de rapid.

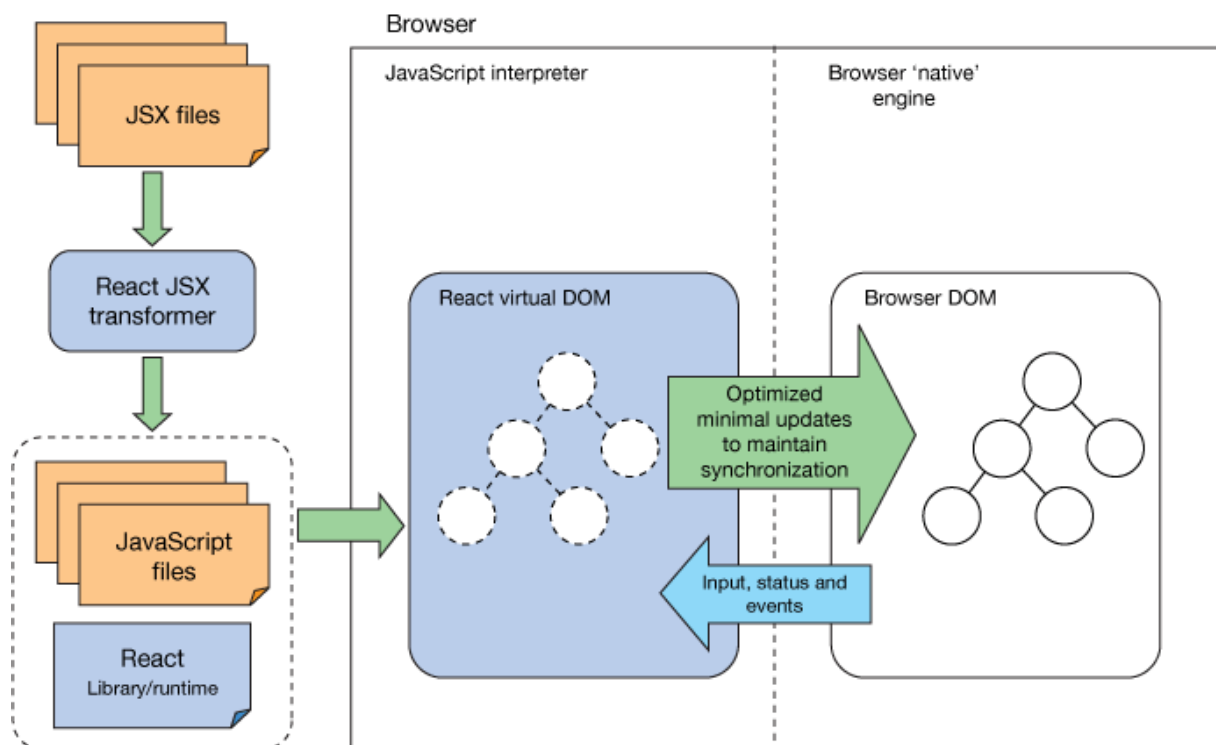


Figura 3.3 Arhitectura React [22]

3.3.3 Funcționalitatea React

React oferă o serie de caracteristici care facilitează dezvoltarea de interfețe de utilizator interactive.

Una dintre cele mai importante caracteristici este sistemul de componente. În React, interfața utilizator este împărțită în componente reutilizabile. Fiecare componentă controlează o anumită parte a interfeței utilizator și poate fi utilizată în mod independent sau în combinație cu alte componente. Componentele pot fi de asemenea imbricate, creând astfel o structură de arbore [23].

Un alt aspect cheie al funcționalității React este gestionarea statului și a datelor. React folosește un model unidirecțional de flux de date, în care datele sunt transmise de la componente părinte la componente copil prin intermediul "props". Acest lucru face ca datele să fie ușor de urmărit și de înțeles, facilitând gestionarea statului aplicației [23].

React oferă, de asemenea, un sistem puternic de gestionare a evenimentelor. Acesta permite dezvoltatorilor să ataseze manipulatori de evenimente la elemente ale interfeței de utilizator și să răspundă la diferite tipuri de evenimente, cum ar fi clicurile sau schimbările de input [23].

În cele din urmă, React oferă și un mecanism pentru lucrul cu forme și validarea datelor. Acesta include funcționalități pentru manipularea inputurilor de formă, validarea datelor și manipularea submiturilor de formă [23].

3.3.4 Exemplu de utilizare a React

Un exemplu simplu de componentă React ar putea fi un buton. Această componentă ar putea avea un stat care urmărește dacă butonul este sau nu apăsat, și un eveniment de clic care modifică acest stat. Componenta buton ar putea fi reutilizată în orice loc în care este necesar un buton, cu diferite funcționalități asociate evenimentului de clic.

Un alt exemplu ar putea fi o listă de elemente. Aceasta ar putea fi o componentă care primește un array de elemente ca "props" și le afișează într-o listă. Fiecare element din listă ar putea fi, de asemenea, o componentă, care afișează datele elementului și poate include un buton pentru ștergerea elementului din listă.

Într-o aplicație mai complexă, ar putea exista mai multe niveluri de componente imbricate, fiecare cu propriul stat și propriile evenimente, lucru care creează o interfață de utilizator dinamică și interactivă.

3.4 PostgreSQL

3.4.1 Descrierea PostgreSQL

PostgreSQL este un sistem de gestionare a bazelor de date relaționale orientate pe obiecte și open source care utilizează și extinde limbajul SQL combinat cu multe caracteristici care îi permit stocarea și scalarea în mod sigur a volumelor de date mari [24].

PostgreSQL a fost dezvoltat pentru prima dată la Universitatea din California, la Berkeley, în 1986, și de atunci a devenit una dintre cele mai populare baze de date open source din lume. Acesta oferă o gamă largă de funcționalități, inclusiv tranzacții ACID (Atomicity, Consistency, Isolation, Durability), suport pentru tipuri de date definite de utilizator, indexare funcțională, colectarea și raportarea statistică și multe altele [25].

3.4.2 Arhitectura PostgreSQL

În arhitectura PostgreSQL, o instanță a bazei de date este numită un "cluster". Un cluster conține baze de date, care conțin în continuare scheme. Schemelere cuprind tabele și alte obiecte de bază de date, cum ar fi tipuri de date, funcții și operatori [24].

PostgreSQL funcționează prin utilizarea proceselor separate de sistem de operare pentru a asigura izolarea între diferite sesiuni ale clientului. Un proces master, numit "postmaster", acceptă conexiuni de la clienți, le autentifică și creează un nou proces pentru fiecare sesiune [25].

3.5 IntelliJ IDEA

3.5.1 Descrierea IntelliJ IDEA

IntelliJ IDEA este un mediu de dezvoltare integrat (IDE) creat de JetBrains pentru dezvoltarea de software. Este un instrument folosit în mod obișnuit de dezvoltatorii de Java, dar suportă și multe alte limbaje de programare, cum ar fi JavaScript, Kotlin, Groovy, Scala și altele [26].

IntelliJ IDEA se remarcă prin caracteristicile sale avansate de analiză a codului, care pot ajuta dezvoltatorii să scrie cod mai curat și mai eficient. Acesta include refactorizarea codului avansată, un sistem inteligent de finalizare a codului, instrumente de depurare și testare, și integrare cu multe alte instrumente și tehnologii populare [26].

3.5.2 Funcționalitatea IntelliJ IDEA

Funcționalitățile oferite de IntelliJ IDEA variază în funcție de versiunea folosită. Versiunea sa comunitară este disponibilă gratuit și include funcționalități pentru dezvoltarea de aplicații Java, Scala, Groovy și Android, în timp ce versiunea sa Ultimate, care este disponibilă printr-un abonament plătit, include funcționalități suplimentare pentru dezvoltarea de aplicații web, dezvoltarea de baze de date, și suport pentru multe alte limbaje și tehnologii [26].

De asemenea, IntelliJ IDEA oferă un sistem de plug-inuri extensibil, care permite dezvoltatorilor să-și personalizeze mediul de dezvoltare prin adăugarea de funcționalități suplimentare [26].

3.6 WebStorm

3.6.1 Descrierea WebStorm

WebStorm este un mediu de dezvoltare integrat (IDE) creat de JetBrains specializat în dezvoltarea web. Acesta oferă suport pentru JavaScript și limbaje conexe, cum ar fi TypeScript, HTML, CSS, precum și pentru Node.js și React și alte biblioteci și cadre JavaScript [27].

Caracteristicile WebStorm includ analiză a codului, finalizare a codului bazată pe inteligența artificială, depurare, testare și integrare cu multe instrumente și tehnologii populare. WebStorm este apreciat pentru abilitatea sa de a oferi un flux de lucru productiv și eficient pentru dezvoltatorii web [27].

3.6.2 Funcționalitatea WebStorm

WebStorm include un set de instrumente avansate pentru dezvoltarea web, cum ar fi editorul său inteligent care oferă completări de cod și refactorizări, un instrument de depurare integrat pentru JavaScript și Node.js, și un tester integrat pentru a ajuta la scrierea și rularea testelor unitare. WebStorm include, de asemenea, un terminal integrat, asistare pentru lucrul cu Git și alte sisteme de control al versiunilor, și suport pentru dezvoltarea web mobilă cu Cordova și Ionic [27].

WebStorm este disponibil printr-un abonament plătit, dar JetBrains oferă și licențe gratuite pentru studenți și profesori, precum și pentru unele proiecte open source [27].

3.7 Postman

3.7.1 Descrierea Postman

Postman este o platformă populară pentru testarea API-urilor. Acesta permite dezvoltatorilor să creeze și să trimită cereri HTTP, să vizualizeze răspunsurile și să creeze teste automate pentru API-uri [28].

Postman a fost lansat pentru prima dată în 2012 ca o extensie pentru Google Chrome, dar acum este disponibil ca o aplicație de sine stătătoare pentru Windows, MacOS și Linux. Postman este folosit de milioane de dezvoltatori din întreaga lume și este adesea considerat un instrument esențial pentru orice dezvoltator de aplicații web [28].

3.7.2 Funcționalitatea Postman

Postman oferă un set de instrumente pentru lucrul cu API-uri, inclusiv un client pentru a crea și a trimite cereri HTTP, un set de instrumente pentru a vizualiza și a analiza răspunsurile, și un set de instrumente pentru a crea și a rula teste automate. Postman include, de asemenea, suport pentru diverse tipuri de autentificare, posibilitatea de a importa și exporta colecții de cereri, și suport pentru variabile de mediu pentru a simplifica lucrul cu diferite configurații de mediu [28].

Postman este disponibil în versiuni gratuite și plătite. Versiunea gratuită include funcționalități de bază, în timp ce versiunea plătită, numită Postman Pro, include funcționalități suplimentare, cum ar fi colaborarea în echipă, monitorizarea API-urilor și integrare cu sistemele de integrare continuă [28].

4 Proiectare si implementare

4.1 Backend

4.1.1 Sistemul de logare

4.1.1.1 AuthController și procesul de autentificare

AuthController este punctul de intrare pentru cererile de autentificare. Este un controler Spring, indicat prin adnotarea `@RestController`, care înseamnă că răspunde la cererile HTTP și își returnează răspunsurile ca JSON (sau alte formate, dacă sunt specificate).

Metoda `login()` din AuthController gestionează cererile de autentificare. Acesta este semnalată de `@PostMapping("/auth/login")`, care specifică faptul că metoda va răspunde la cererile POST către ruta `/auth/login`. În Spring, aceasta este o modalitate concisă de a adnota o metodă ca fiind un punct de terminare HTTP.

Metoda `login()` primește un obiect de tip `CredentialsDTO`, care conține datele de autentificare ale utilizatorului (nume de utilizator și parolă). Aceasta este încapsulată în corpul cererii HTTP, care este serializat automat în obiectul DTO de către Spring.

```
@PostMapping("/auth/login")
public ResponseEntity<AuthResponseDTO> login(@RequestBody CredentialsDTO credentialsDTO) {
    var authentication : Authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(credentialsDTO.getUsername(),
            credentialsDTO.getPassword()));

    SecurityContextHolder.getContext().setAuthentication(authentication);
    String token = jwtGenerator.generateToken(authentication);
    return new ResponseEntity<>(new AuthResponseDTO(token), HttpStatus.OK);
}
```

Figura 4.1 Metoda `login()`

Se utilizează apoi `AuthenticationManager` pentru a valida aceste credențiale. `AuthenticationManager` este o interfață centrală în sistemul de securitate Spring, care are rolul de a valida certificatele de autentificare. În contextul nostru, este folosit pentru a valida combinația de nume de utilizator și parolă furnizate de utilizator.

`AuthController` construiește un obiect `UsernamePasswordAuthenticationToken` cu numele de utilizator și parola din `CredentialsDTO`, și îl trece către metoda `authenticate()` a `AuthenticationManager`. `UsernamePasswordAuthenticationToken` este o implementare a interfeței `Authentication`, care deține detaliile de autentificare (în acest caz, numele de utilizator și parola). Acesta este doar un vehicul pentru transportul datelor de autentificare.

Dacă autentificarea este reușită, `AuthenticationManager` returnează un obiect `Authentication` completat, care include detalii despre utilizatorul autentificat. Dacă autentificarea eșuează, `AuthenticationManager` aruncă o excepție `AuthenticationException`.

După autentificarea cu succes, `AuthController` stochează obiectul `Authentication` în `SecurityContext`, folosind `SecurityContextHolder`. `SecurityContext` deține detaliile de securitate ale aplicației, inclusiv detalii despre autentificare. `SecurityContextHolder` este un ajutor care oferă acces la `SecurityContext`.

În continuare, `AuthController` generează un token JWT pentru utilizatorul autentificat. Aceasta este realizată prin metoda `generateToken()` a componentei `JWTGenerator`. Token-ul JWT este o reprezentare compactă, URL-safe, a unei colecții de afirmații (claims) care pot fi făcute despre un subiect (în acest caz, utilizatorul). Aceste afirmații sunt codificate și semnate digital, astfel încât să poată fi verificate mai târziu de către aplicație.

```
public String generateToken(Authentication authentication) {
    String username = authentication.getName();
    Date currentDate = new Date();
    Date expireDate = new Date(currentDate.getTime() + JWT_EXPIRATION);

    return Jwts.builder()
        .setSubject(username)
        .setIssuedAt(currentDate)
        .setExpiration(expireDate)
        .claim("authorities", authentication.getAuthorities().iterator().next().getAuthority())
        .claim("username", username)
        .signWith(SignatureAlgorithm.HS512, JWT_SECRET)
        .compact();
}
```

Figura 4.2 Metoda `generateToken()`

Token-ul JWT este apoi returnat către client în răspunsul HTTP, încapsulat într-un obiect `AuthResponseDTO`.

4.1.1.2 *JWTAuthenticationFilter și procesarea cererilor autentificate*

Dacă un client face o cerere la un endpoint securizat al aplicației, aceasta trebuie să includă token-ul JWT în header-ul Authorization al cererii HTTP. Aplicația trebuie să verifice acest token pentru a se asigura că cererea provine de la un client autentificat. Aceasta este responsabilitatea JWTAuthenticationFilter.

JWTAuthenticationFilter este un tip de OncePerRequestFilter, care este un filtru special în Spring care se execută o singură dată pentru fiecare cerere HTTP. JWTAuthenticationFilter este configurat să se execute înainte de celelalte filtre și interceptori din lanțul de filtre (filter chain), astfel încât să poată autentifica cererea înainte de a se continua cu procesarea ei.

JWTAuthenticationFilter extrage token-ul JWT din header-ul Authorization al cererii. Acesta utilizează apoi JWTGenerator pentru a decoda și a verifica token-ul. Dacă token-ul este valid, JWTAuthenticationFilter extrage detaliile de autentificare ale utilizatorului din acesta, creează un obiect Authentication și îl stochează în SecurityContext, folosind SecurityContextHolder. Acest proces este adesea numit "autentificare cu starea fără stări" (stateless authentication), deoarece aplicația nu stochează nicio informație despre sesiunea utilizatorului între cereri.

```
String token = tokenGenerator.getJWTFromRequest(request);

if (StringUtils.hasText(token) && tokenGenerator.validateToken(token)) {
    var username :String = tokenGenerator.getUsernameFromJWT(token);
    var userDetails :UserDetails = customUserDetailsService.loadUserByUsername(username);
    var authenticationToken = new UsernamePasswordAuthenticationToken(userDetails, credentials: null,
        userDetails.getAuthorities());

    authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
    SecurityContextHolder.getContext().setAuthentication(authenticationToken);
}
filterChain.doFilter(request, response);
```

Figura 4.3 Metoda *doFilterInternal()*

4.1.1.3 *JwtAuthEntryPoint și gestionarea erorilor de autentificare*

JwtAuthEntryPoint este o componentă care gestionează erorile de autentificare în aplicație. Aceasta implementează interfața AuthenticationEntryPoint, care definește un punct de intrare pentru gestionarea excepțiilor de autentificare în Spring Security.

```

@Component
public class JwtAuthEntryPoint implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
        AuthenticationException authException) throws IOException {
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, authException.getMessage());
    }
}

```

Figura 4.4 Componenta *JwtAuthEntryPoint*

Dacă o cerere este trimisă la un endpoint securizat fără un token JWT valid sau fără niciun token deloc, *JWTAuthenticationFilter* va arunca o excepție *AuthenticationException*. *JwtAuthEntryPoint* este configurat pentru a intercepta aceste excepții și pentru a returna un răspuns HTTP cu un cod de eroare corespunzător (401 Unauthorized).

4.1.1.4 Configurarea lanțului de filtre

Lanțul de filtre (filter chain) este un mecanism central în infrastructura de securitate a Spring. Acesta este o serie de filtre care sunt aplicate pentru fiecare cerere HTTP către aplicația Spring. Fiecare filtru poate procesa cererea într-un mod specific, cum ar fi autentificarea, autorizarea, logarea, gestionarea erorilor etc.

Filtrul *JWTAuthenticationFilter* este adăugat în lanțul de filtre printr-o clasă de configurare care este adnotată cu *@EnableWebSecurity*. Aceasta clasă este, de obicei, locul unde se configurează toate aspectele legate de securitatea web în Spring, inclusiv autentificarea, autorizarea, gestionarea sesiunilor, protecția împotriva atacurilor CSRF etc.

În această clasă de configurare, metoda *filterChain(HttpSecurity http)* este suprascrisă pentru a configura lanțul de filtre. Metoda ***filterChain(HttpSecurity http)*** este o metodă **@Bean** care configurează lanțul de filtre.

1. ***http.cors().and()***: Activarea suportului pentru Cross-Origin Resource Sharing (CORS). CORS este o politică de securitate a browserului care controlează cum resursele pot fi accesate de pe un domeniu diferit de cel de pe care a fost trimisă cererea.
2. ***http.csrf().disable()***: Dezactivarea protecției împotriva atacurilor Cross-Site Request Forgery (CSRF). CSRF este un tip de atac care face ca utilizatorul victima să efectueze o acțiune nedorită pe un site web în care este autentificat.
3. ***http.exceptionHandling().authenticationEntryPoint(authEntryPoint)***: Configurarea unui entry point pentru gestionarea excepțiilor. ***authEntryPoint*** este un bean care se ocupă de manipularea excepțiilor de autentificare.

4. **http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS):** Configurarea politicii de gestionare a sesiunii. În acest caz, aplicația este configurată să nu creeze nicio sesiune, ceea ce face aplicația "stateless".
5. **http.authorizeRequests():** Începe definirea politicilor de autorizare.
6. **.antMatchers("/api/auth/**").permitAll():** Orice cerere care se potrivește cu acest model de URL ("/api/auth/**") este permisă pentru toți utilizatorii, indiferent dacă sunt autentificați sau nu.
7. **.antMatchers("/api/admin/**").hasAuthority("ADMIN"):** Orice cerere care se potrivește cu acest model de URL ("/api/admin/**") este permisă numai pentru utilizatorii cu autoritatea "ADMIN".
8. **.anyRequest().authenticated():** Orice alte cereri sunt permise numai pentru utilizatorii autentificați.
9. **http.httpBasic():** Activarea autentificării HTTP Basic. Aceasta este o schemă simplă de autentificare care trimite numele de utilizator și parola într-un header HTTP codificat în Base64.
10. **http.addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class):** Adăugarea filtrului JWT în lanțul de filtre, înainte de **UsernamePasswordAuthenticationFilter**. Filtrul JWT este responsabil pentru extragerea token-ului JWT din cerere și pentru autentificarea utilizatorului pe baza acestuia.
11. **.build():** Construirea lanțului de filtre.

Această configurație oferă aplicației o securitate robustă, permițând accesul liber la unele URL-uri, dar restricționând alte URL-uri doar pentru utilizatorii autentificați sau utilizatorii cu anumite autorități.

```
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
        .cors().and() HttpSecurity
        .csrf().disable()
        .exceptionHandling() ExceptionHandlingConfigurer<HttpSecurity>
        .authenticationEntryPoint(authEntryPoint)
        .and() HttpSecurity
        .sessionManagement() SessionManagementConfigurer<HttpSecurity>
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and() HttpSecurity
        .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
        .antMatchers( ...antPatterns: "/api/auth/**").permitAll()
        .antMatchers( ...antPatterns: "/api/admin/**").hasAuthority("ADMIN")
        .anyRequest().authenticated()
        .and() HttpSecurity
        .httpBasic() HttpBasicConfigurer<HttpSecurity>
        .and() HttpSecurity
        .addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class)
        .build();
}
```

Figura 4.5 Configurarea lanțului de filtre

4.1.1.5 UserDetailsService, UserEntity si roluri

În Spring Security, UserDetailsService este o interfață cheie pentru recuperarea detaliilor unui utilizator. Aceasta este folosită de DaoAuthenticationProvider pentru a încărca detaliile despre utilizator în timpul procesului de autentificare.

Interfața UserDetailsService are o singură metodă, **loadUserByUsername(String username)**, care găsește un utilizator în funcție de numele de utilizator și returnează un obiect de tip UserDetails. Dacă nu poate găsi utilizatorul, aruncă o excepție UsernameNotFoundException.

Am implementat UserDetailsService printr-o clasă CustomUserDetailsService, care folosește un UserRepository pentru a găsi utilizatorul în baza de date și un userConverter pentru a converti obiectul de entitate UserEntity într-un DTO.

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    UserEntity user = userRepository.findByUsername(username)
        .orElseThrow(() -> new UsernameNotFoundException("Username not found"));

    return new User(user.getUsername(), user.getPassword(), mapRolesToAuthorities(user.getRoles()));
}
```

Figura 4.6 Metoda loadUserByUsername()

În metoda **loadUserByUsername()**, se folosește metoda **findByUsername()** a UserRepository pentru a căuta utilizatorul în baza de date. Dacă utilizatorul nu există, se aruncă o excepție **UsernameNotFoundException**. Dacă utilizatorul există, se returnează un obiect UserDetails, creat printr-un nou obiect User, cu numele de utilizator, parola și autoritățile utilizatorului.

Autoritățile reprezintă drepturile pe care le are un utilizator. Acestea sunt obținute din rolurile utilizatorului, prin metoda mapRolesToAuthorities(). Rolurile sunt transformate în obiecte GrantedAuthority folosind clasa SimpleGrantedAuthority. Autoritățile sunt apoi folosite de Spring Security pentru a verifica dacă un utilizator are dreptul să acceseze anumite resurse ale aplicației.

```
private Collection<GrantedAuthority> mapRolesToAuthorities(List<RoleEntity> roles) {
    return roles.stream().map(roleEntity -> new SimpleGrantedAuthority(roleEntity.getName()))
        .collect(Collectors.toList());
}
```

Figura 4.7 Metoda maprolesToAuthorities()

Avem două roluri, ADMIN și USER. În mod general, ADMIN ar avea acces la toate resursele, în timp ce USER ar avea acces restricționat.

UserEntity este o entitate JPA care reprezintă un utilizator în baza de date. Aceasta conține toate informațiile despre un utilizator, inclusiv numele, parola și rolurile acestuia. Rolurile sunt mapate printr-o relație **ManyToMany** cu RoleEntity, o altă entitate JPA care reprezintă un rol.

RoleEntity conține doar un id și un nume, care este numele rolului. UserEntity și RoleEntity sunt conectate printr-un tabel intermediar, user_roles, care mapază rolurile pe care le are fiecare utilizator.

4.1.2 Creerea de utilizatori

Procesul de înregistrare a unui nou utilizator începe cu un request HTTP POST trimis la endpoint-ul "/admin/register". Acest request trebuie să fie trimis de un utilizator cu rolul de ADMIN, datorită configurației de securitate implementată prin filtrul de autentificare.

Requestul conține un corp care reprezintă un obiect **UserDTO**. Acest obiect conține informațiile necesare pentru înregistrarea unui nou utilizator, cum ar fi numele, prenumele, credențialele (username și password), adresa și emailul.

După ce requestul este primit de server, acesta este gestionat de metoda **register** din **AuthController**. Această metodă apelează **registerUser** din **UserService** și îi trimite **UserDTO**-ul primit.

În metoda **registerUser**, prima operațiune este validarea datelor utilizatorului. Aceasta se face prin **UserValidator**, care verifică dacă există deja un utilizator cu același username sau email în baza de date. Dacă există deja un astfel de utilizator, validatorul va arunca o excepție de tip **BadRequestException** cu un mesaj corespunzător.

Dacă validarea trece cu succes, atunci **UserDTO**-ul este convertit într-un obiect **UserEntity** prin **UserConverter**. Acest **UserEntity** este apoi salvat în baza de date prin metoda **save** a **UserRepository**.

În cazul în care apare vreo eroare în timpul procesului de înregistrare, aceasta va fi gestionată de **ControllerExceptionHandler**. Acesta este un **ControllerAdvice**, ceea ce înseamnă că el va fi aplicat tuturor controlerelor din aplicație.

În **ControllerExceptionHandler**, am definit câte un handler pentru excepțiile **AuthenticationException** și **BadRequestException**. În cazul în care o astfel de excepție este aruncată în timpul procesării unui request, handlerul corespunzător va fi apelat automat de Spring și va returna un răspuns HTTP cu un cod de stare corespunzător și cu mesajul excepției ca corp.

Rezumând, procesul de înregistrare a unui nou utilizator este următorul:

1. Se primește un request HTTP POST la endpoint-ul "/admin/register" de la un utilizator cu rolul de ADMIN.
2. **AuthController** preia requestul și îi extrage corpul, care este un **UserDTO**.
3. **AuthController** apelează **registerUser** din **UserService** și îi trimite **UserDTO**-ul.
4. **UserService** validează **UserDTO**-ul folosind **UserValidator**. Dacă validarea eșuează, se aruncă o excepție.

5. Dacă validarea reușește, **UserService** convertește **UserDTO**-ul într-un **UserEntity**.
6. **UserRepository** salvează **UserEntity**-ul în baza de date prin **JpaUserRepository**.
7. Dacă apare vreo eroare în timpul procesului, aceasta este gestionată de **ControllerExceptionHandler**, care returnează un răspuns HTTP corespunzător.

Acest flux permite înregistrarea noilor utilizatori într-un mod controlat și securizat, asigurându-se că doar utilizatorii cu rolul de ADMIN pot adăuga noi utilizatori și că informațiile noilor utilizatori sunt validate înainte de a fi salvate în baza de date.

4.1.3 Crearea de facturi

Procesul de creare a unei noi facturi începe cu un request HTTP POST trimis la endpoint-ul `/bill`. Requestul conține un corp care reprezintă un obiect **BillDTO**. Acest obiect conține informațiile necesare pentru crearea unei noi facturi, cum ar fi codul de client, codul de facturare, data, suma totală, lista de produse etc.

După ce requestul este primit de server, acesta este gestionat de metoda **create()** din **BillController**. Această metodă apelează metoda **create()** din **BillService** și îi trimite **BillDTO**-ul primit.

În metoda **create()** din **BillService**, prima operațiune este validarea datelor facturii. Aceasta se face prin **BillValidator**, care verifică dacă toate datele facturii sunt corecte și complete. Dacă validarea eșuează, validatorul va arunca o excepție de tip **BadRequestException** cu un mesaj corespunzător.

```
public BillDTO create(BillDTO bill, HttpServletRequest request) throws Exception {
    validator.validate(bill);
    var userEntity : UserEntity = userRepository.findByUsername(jwtGenerator.getUsernameFromRequest(request))
        .orElseThrow(Exception::new);

    var entity : BillEntity = converter.convertToEntity(bill);
    entity.setUser(userEntity);
    entity.setPrice(Math.round(entity.getUnits() * entity.getCostPerUnit() * 100) / 100.0);
    entity.setIssueDate(
        bill.getIssueDate().atOffset(ZoneOffset.UTC).withHour(12).withMinute(0).withSecond(0)
        .truncatedTo(ChronoUnit.SECONDS).toInstant());
    setDates(bill, entity);

    return converter.convertToDTO(billRepository.save(entity));
}
```

Figura 4.8 Metoda create() din BillService

Dacă validarea trece cu succes, atunci **BillDTO**-ul este convertit într-un obiect **BillEntity** prin **BillConverter**. Acest **BillEntity** este apoi salvat în baza de date prin metoda `save` a **BillRepository**.

În cazul în care apare vreo eroare în timpul procesului de creare a facturii, aceasta va fi gestionată de `ControllerExceptionHandler`. Acesta este un `ControllerAdvice`, ceea ce înseamnă că el va fi aplicat tuturor controlerelor din aplicație.

În **`ControllerExceptionHandler`**, am definit câte un handler pentru excepțiile `AuthenticationException` și `BadRequestException`. În cazul în care o astfel de excepție este aruncată în timpul procesării unui request, handlerul corespunzător va fi apelat automat de Spring și va returna un răspuns HTTP cu un cod de stare corespunzător și cu mesajul excepției ca corp.

Rezumând, procesul de creare a unei noi facturi este următorul:

1. Se primește un request HTTP POST la endpoint-ul `"/bill"` de la partea unui utilizator autentificat.
2. **`BillController`** preia requestul și îi extrage corpul, care este un `BillDTO`.
3. **`BillController`** apelează metoda **`create()`** din **`BillService`** și îi trimite `BillDTO`-ul.
4. **`BillService`** validează `BillDTO`-ul folosind **`BillValidator`**. Dacă validarea eșuează, se aruncă o excepție.
5. Dacă validarea reușește, **`BillService`** convertește `BillDTO`-ul într-un **`BillEntity`**.
6. **`BillRepository`** salvează `BillEntity`-ul în baza de date prin **`JpaBillRepository`**.
7. Dacă apare vreo eroare în timpul procesului, aceasta este gestionată de **`ControllerExceptionHandler`**, care returnează un răspuns HTTP corespunzător.

4.1.4 Istoricul facturilor pe tip

Pentru a vizualiza istoricul facturilor pe tip, procesul începe cu un request HTTP GET trimis la endpoint-ul `"/api/bills"`. Requestul trebuie să fie trimis de un utilizator autentificat, iar filtrul de autentificare al aplicației va asigura că doar utilizatorii autentificați au acces la acest endpoint.

Requestul HTTP GET va avea inclus ca parametri `"billType"`, care specifică tipul de factură dorit, și `"paid"`, care specifică dacă sunt dorite facturile plătite sau nu.

Aceste parametri sunt trimise către metoda **`getPerType`** a serviciului **`BillService`**, împreună cu informații din **`HttpServletRequest`** pentru a determina utilizatorul care face solicitarea.

Metoda **`getPerType`** va începe prin identificarea utilizatorului care face solicitarea. Acest lucru se face prin folosirea **`JWTGenerator`** pentru a obține numele de utilizator din cererea HTTP și apoi interogarea bazei de date pentru a obține **`UserEntity`** asociat cu acel nume de utilizator.

Odată ce a fost identificat utilizatorul, metoda va apela metoda **`getByTypeAndPaid`** a repository-ului **`BillRepository`**, furnizându-i **`BillType`** asociat cu **`billType`** primit ca parametru, starea de plată și **`UserEntity`**. Aceasta va interoga baza de date și va returna o listă cu facturile care corespund criteriilor.

Apoi, lista de entități **BillEntity** este transformată într-o listă de **BillDTO** prin folosirea **Converter**-ului și returnată către client.

În cazul în care apare o eroare (de exemplu, dacă numele de utilizator din tokenul JWT nu corespunde unui utilizator existent), aceasta va fi aruncată și gestionată de către **ControllerExceptionHandler**.

Acest proces de vizualizare a istoricului facturilor pe tip se poate rezuma astfel:

1. Se primește un request HTTP GET la endpoint-ul `"/api/bills"` de la un utilizator autentificat.
2. **BillController** preia requestul și extrage parametrii `"billType"` și `"paid"`.
3. **BillController** apelează metoda **getPerType** din **BillService**, trimițându-i parametrii și obiectul **HttpServletRequest**.
4. **BillService** identifică **UserEntity** asociat cu numele de utilizator din cererea HTTP.
5. **BillService** apelează **getByTypeAndPaid** din **BillRepository** și obține o listă de **BillEntity**.
6. **BillService** convertește lista de **BillEntity** în **BillDTO** și o returnează către **BillController**.
7. **BillController** returnează lista de **BillDTO** ca răspuns la requestul HTTP GET.
8. În cazul în care apare o eroare, aceasta este gestionată de **ControllerExceptionHandler**.

Prin urmare, acest flux permite utilizatorilor autentificați să vizualizeze istoricul facturilor pe tip, asigurându-se că doar utilizatorii autentificați pot accesa aceste informații și că sunt returnate doar facturile asociate cu utilizatorul care face solicitarea.

4.1.5 Crearea de cheltuieli

ExpenseController este un controller REST, care expune un endpoint pentru a crea o nouă cheltuială (expense). Atunci când clientul trimite o cerere POST către `"/api/expense"`, metoda **create** este apelată. Cererea trebuie să conțină un obiect de tip **ExpenseDTO** în format JSON și o listă de atașamente, care sunt opționale.

@RequestPart("expense") String expense - Aceasta este partea de JSON a cererii, care este convertită într-un șir de caractere. JSON-ul reprezintă datele cheltuielii care urmează să fie create.

@RequestPart(value = "attachments", required = false) List<MultipartFile> attachments - Aceasta este partea de atașamente a cererii, care este convertită într-o listă de fișiere MultipartFile. Aceste fișiere sunt opționale și pot fi încărcate împreună cu datele cheltuielii.

După ce datele sunt primite, sunt procesate de metoda **create** a serviciului **ExpenseService**. Acesta este serviciul care se ocupă cu logica pentru cheltuieli. Atunci

când metoda **create** este apelată, ea validează mai întâi datele cheltuielii cu ajutorul validatorului. Apoi, extrage detaliile utilizatorului care a trimis cererea și setează utilizatorul ca proprietar al cheltuielii. Dacă sunt atașamente, fiecare fișier este transformat într-o entitate **AttachmentEntity** și este adăugat la lista de atașamente a cheltuielii.

În final, entitatea **ExpenseEntity** este salvată în baza de date cu ajutorul metodei **save** a repositorului **ExpenseRepository** și este returnat un DTO al entității salvate.

Un **MultipartFile** reprezintă un fișier încărcat într-o cerere multipart/form-data. **@RequestPart** este o anotație Spring care se folosește pentru a asocia o parte a unei cereri multipart cu un parametru de metodă într-un controler.

Clasa **ExpenseConverter** este responsabilă pentru conversia dintre obiectele DTO (Data Transfer Object) și entitățile de baze de date. În acest context, se convertește între **ExpenseDTO** și **ExpenseEntity** și între **AttachmentDTO** și **AttachmentEntity**.

ModelMapper este o bibliotecă care se ocupă automat cu maparea între obiectele de diferite tipuri. În metoda **configureModelMapper**, sunt configurate regulile de mapare între **AttachmentDTO** și **AttachmentEntity**. În acest caz, se evită maparea câmpului **data** în timpul conversiei de la **AttachmentEntity** la **AttachmentDTO**, dar se mapează în timpul conversiei de la **AttachmentDTO** la **AttachmentEntity**.

@Lob este o anotație JPA care indică faptul că un câmp ar trebui să fie persistat ca un tip de date mare (Large Object - LOB). În cazul de față, este utilizat pentru câmpul **data** în **AttachmentEntity**. Acesta indică faptul că datele atașamentului, care sunt în format binar, ar trebui să fie stocate ca un BLOB în baza de date. Acest lucru este necesar deoarece atașamentele pot fi de dimensiuni foarte mari și nu ar putea fi stocate într-un câmp normal.

Entitatea **AttachmentEntity** are câmpurile necesare pentru a stoca un fișier în baza de date: **data** este conținutul fișierului, **fileName** este numele fișierului, iar **fileType** este tipul de media al fișierului (ex: "image/jpeg"). De asemenea, fiecare atașament este legat de o cheltuială prin intermediul câmpului **expense**.

Există o relație bidirecțională între entitățile **ExpenseEntity** și **AttachmentEntity**. Fiecare cheltuială poate avea mai multe atașamente, iar fiecare atașament aparține unei singure cheltuieli.

Există, de asemenea, o relație bidirecțională între entitățile **UserEntity** și **ExpenseEntity**. Fiecare utilizator poate avea mai multe cheltuieli, iar fiecare cheltuială aparține unui singur utilizator.

4.1.6 Istoricul cheltuielilor

Funcționalitatea de preluare a istoricului cheltuielilor este implementată în controllerul **ExpenseController** și în serviciul **ExpenseService**, cu sprijinul **ExpenseConverter**, pentru conversia datelor între formatele DTO și Entity.

În primul rând, **ExpenseController** expune un endpoint HTTP GET la `"/api/expenses/all"`, prin care clienții pot solicita istoricul cheltuielilor. Metoda **getAll** este responsabilă cu primirea cererii și este înzestrată cu parametrul **paid** pentru a filtra cheltuielile după statutul lor de plată și un obiect **HttpServletRequest** pentru a prelua detalii despre cererea HTTP.

Aceasta delegă procesarea cererii către serviciul **ExpenseService**, apelând metoda sa **getAll** cu parametrii cererii. În interiorul acestei metode, se prelucrează cererea primită de la client. Este extras numele de utilizator din cererea HTTP, folosind generatorul JWT, iar apoi se interoghează repository-ul **userRepository** pentru a obține entitatea **UserEntity** asociată acestuia.

După identificarea utilizatorului, metoda **getAll** din **expenseRepository** este apelată pentru a obține lista de cheltuieli asociată acestuia, filtrată pe baza parametrului **paid**. Rezultatul acestei interogări este o listă de entități **ExpenseEntity**, care reprezintă cheltuielile stocate în baza de date.

Pentru a converti această listă de entități **ExpenseEntity** într-un format prietenos pentru client, lista este transformată într-un flux (stream) și apoi este aplicată metoda **map**, care convertește fiecare entitate într-un obiect DTO. Conversia este realizată de către metoda **convertToDTO** a instanței **ExpenseConverter**.

Clasa **ExpenseConverter** este responsabilă cu conversia datelor între formatul DTO și formatul Entity. Pentru a realiza acest lucru, folosește un obiect **ModelMapper**, care este configurat pentru a asocia câmpurile corespunzătoare între cele două formate. De exemplu, în timpul conversiei de la **AttachmentEntity** la **AttachmentDTO**, câmpul **data** este omis, pentru a economisi lățimea de bandă, dar este inclus în timpul conversiei inversă, de la **AttachmentDTO** la **AttachmentEntity**.

În final, metoda **getAll** din **ExpenseService** returnează lista de obiecte **ExpenseDTO**, care este trimisă înapoi către client ca răspuns la cererea inițială. Acest proces asigură că informațiile despre cheltuielile unui utilizator sunt obținute în mod eficient și sunt prezentate într-un format adecvat pentru client.

4.1.7 Pagina principala

4.1.7.1 Secțiunea de facturi

Funcționalitatea de returnare a tuturor facturilor este implementată în controlerul **BillController** și în serviciul **BillService**, cu sprijinul **BillConverter**, pentru conversia datelor între formatele DTO și Entity.

În primul rând, **BillController** expune un endpoint HTTP GET la `"api/bills/all"`, prin care clienții pot solicita toate facturile. Metoda **getAll** este responsabilă cu primirea cererii și este înzestrată cu parametrul **paid** pentru a filtra facturile după statutul lor de plată și un obiect **HttpServletRequest** pentru a prelua detalii despre cererea HTTP.

Aceasta delegă procesarea cererii către serviciul **BillService**, apelând metoda sa **getAll** cu parametrii cererii. În interiorul acestei metode, se prelucrează cererea primită de la client. Este extras numele de utilizator din cererea HTTP, folosind generatorul JWT,

iar apoi se interoghează repository-ul **userRepository** pentru a obține entitatea **UserEntity** asociată acestuia.

După identificarea utilizatorului, metoda **getAll** din **billRepository** este apelată pentru a obține lista de facturi asociată acestuia, filtrată pe baza parametrului **paid**. Rezultatul acestei interogări este o listă de entități **BillEntity**, care reprezintă facturile stocate în baza de date.

Pentru a converti această listă de entități **BillEntity** într-un format prietenos pentru client, lista este transformată într-un flux (stream) și apoi este aplicată metoda **map**, care convertește fiecare entitate într-un obiect DTO. Conversia este realizată de către metoda **convertToDTO** a instanței **BillConverter**.

Clasa **BillConverter** este responsabilă cu conversia datelor între formatul DTO și formatul Entity. Pentru a realiza acest lucru, folosește un obiect **ModelMapper**, care este configurat pentru a asocia câmpurile corespunzătoare între cele două formate.

În final, metoda **getAll** din **BillService** returnează lista de obiecte **BillDTO**, care este trimisă înapoi către client ca răspuns la cererea inițială. Acest proces asigură că informațiile despre facturile unui utilizator sunt obținute în mod eficient și sunt prezentate într-un format adecvat pentru client.

4.1.7.2 Secțiunea de cheltuieli

Funcționalitatea de returnare a tuturor facturilor este aceeași funcționalitate explicată în 4.1.6, singura diferență fiind că valoarea inițială a parametrului **paid** este **false**, returnând astfel cheltuielile neplătite.

4.1.8 Descărcarea atașamentelor

Funcționalitatea de descărcare a atașamentelor este implementată în controlerul **ExpenseController** și în serviciul **ExpenseService**, cu sprijinul **AttachmentRepository**.

În primul rând, **ExpenseController** expune un endpoint HTTP GET la `"/expense/attachment/{expenseId}/{attachmentId}"`, prin care clienții pot solicita descărcarea unui atașament. Metoda **downloadAttachment** este responsabilă cu primirea cererii și este înzestrată cu parametrii **expenseId** și **attachmentId** care reprezintă id-ul cheltuielii și id-ul atașamentului dorit.

Aceasta delegă procesarea cererii către serviciul **ExpenseService**, apelând metoda sa **downloadAttachment** cu parametrii cererii. În interiorul acestei metode, se prelucrează cererea primită de la client. Se interoghează repository-ul **attachmentRepository** pentru a obține entitatea **AttachmentEntity** asociată id-urilor primite.

Repository-ul **attachmentRepository** implementează o interogare JPA custom, **findByIdAndExpenseId**, care returnează atașamentul cu id-ul specificat care este

asociat cheltuielii cu id-ul dat. Dacă atașamentul nu este găsit, se aruncă o excepție **NotFoundException**.

Entitatea **AttachmentEntity** este apoi convertită într-un **ByteArrayResource**, care poate fi trimis ca răspuns la client.

În final, metoda **downloadAttachment** din **ExpenseController** creează un **ResponseEntity** cu **ByteArrayResource** ca și corp al răspunsului, setează tipul media corespunzător datelor atașamentului și adaugă un header **Content-Disposition** pentru a specifica că răspunsul trebuie să fie tratat ca o descărcare de fișier, cu numele specificat în **AttachmentEntity**. Acest răspuns este trimis înapoi către client.

Acest proces asigură că se pot descărca atașamentele asociate cheltuielilor în mod eficient, în formatul original în care au fost încărcate.

4.2 Frontend

4.2.1 Sistemul de logare

Componenta **Login** începe cu definirea a trei variabile de stare: **username**, **password** și **snackbar**. Variabilele **username** și **password** sunt inițializate cu valori implicite și vor fi actualizate de fiecare dată când utilizatorul introduce un nume de utilizator sau o parolă în formular. **snackbar** este un obiect care deține starea unei bare de notificare, care poate afișa mesaje de succes sau de eroare către utilizator.

Funcția **handleLogin** este definită ca o funcție asincronă și este apelată atunci când formularul de login este trimis. Aceasta împiedică comportamentul implicit al formularului (reîncărcarea paginii) prin apelarea **event.preventDefault()**. Apoi, încearcă să trimită o cerere HTTP către server folosind API-ul Fetch nativ al JavaScript.

Cererea HTTP este o cerere POST către endpoint-ul `"/api/auth/login"` și include datele formularului (numele de utilizator și parola) în format JSON. Înainte de a trimite cererea, funcția verifică dacă răspunsul primit de la server este OK (codul de stare HTTP 200).

```
const data = {username, password};
const response = await fetch(input: 'http://localhost:8080/api/auth/login', init: {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
});
```

Figura 4.9 Cererea de logare

Dacă răspunsul este OK, atunci serverul a acceptat cererea de login și a trimis înapoi un token JWT în răspuns. Acest token este salvat în sessionStorage folosind **sessionStorage.setItem('token', responseData.accessToken)**. Token-ul JWT este apoi decodificat folosind biblioteca **jwt-decode** pentru a extrage rolul utilizatorului. În funcție de rolul utilizatorului, funcția redirecționează utilizatorul către o pagină diferită.

Dacă răspunsul nu este OK, atunci serverul a respins cererea de login. Funcția tratează acest caz aruncând o excepție cu un mesaj de eroare corespunzător.

Funcția **handleLogin** include și o clauză **catch** pentru a gestiona orice erori care pot apărea în timpul execuției. În cazul în care apare o eroare, aceasta este afișată în bara de notificare.

Randerarea componentei include un formular cu câmpuri pentru numele de utilizator și parola și două butoane, unul pentru trimiterea formularului și unul pentru înregistrare. Ambele câmpuri au handler **onChange** care actualizează starea componentei cu valoarea introdusă de utilizator.

Formularul are un handler **onSubmit** care apelează funcția **handleLogin** atunci când este trimis.

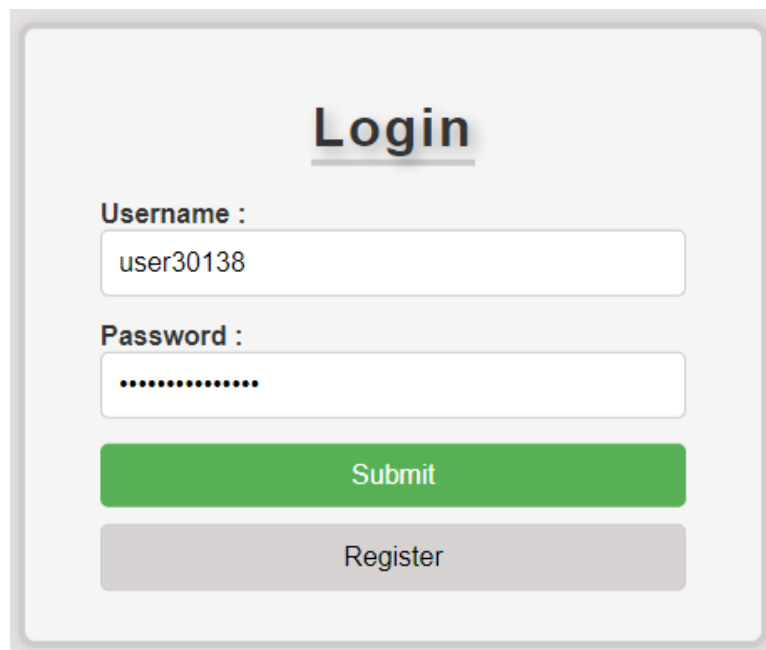
The image shows a login form with a light gray background and rounded corners. At the top, the word "Login" is centered in a bold, dark gray font. Below it, there are two input fields. The first is labeled "Username :" and contains the text "user30138". The second is labeled "Password :" and contains a series of dots, indicating a password field. Below the password field, there are two buttons: a green "Submit" button and a gray "Register" button.

Figura 4.10 Formularul de logare

4.2.2 Crearea de utilizatori

Componenta React pentru înregistrarea unui nou utilizator este un formular care conține câmpuri pentru nume, prenume, username, parolă, confirmare parolă, adresă și email. În plus, are și un mecanism de validare a acestor date, și de a trimite cererea de înregistrare la server.

Când componenta este încărcată, starea inițială a formularului este setată cu ajutorul hook-ului **useState** de la React. Starea inițială a tuturor câmpurilor este un string gol, iar starea inițială a tuturor erorilor este **false**.

Formularul conține un câmp de input pentru fiecare dată necesară înregistrării, și fiecare câmp de input este legat de o variabilă de stare corespunzătoare. Când valoarea unui câmp de input se schimbă, funcția de update a variabilei de stare corespunzătoare este apelată cu noua valoare, actualizând astfel starea componentei.

The image shows a registration form titled "Register". It contains the following fields and labels:

- First Name:** Input field with placeholder text "First Name".
- Last Name:** Input field with placeholder text "Last Name".
- Username:** Input field with placeholder text "Username".
- Password:** Input field with placeholder text "Password".
- Repeat Password:** Input field with placeholder text "Confirm Password".
- Address:** Input field with placeholder text "Address".
- Email:** Input field with placeholder text "Email".

At the bottom of the form is a green button labeled "Register".

Figura 4.11 Formularul de înregistrare

Când formularul este trimis, se apelează funcția **handleRegister**. Aceasta începe prin a reseta toate erorile și prin a construi obiectul de date care va fi trimis la server.

Apoi, funcția validează datele. Dacă vreuna dintre validări eșuează, funcția setează eroarea corespunzătoare ca **true** și afișează un mesaj în **Snackbar** pentru a informa utilizatorul despre eroare, și întrerupe execuția funcției.

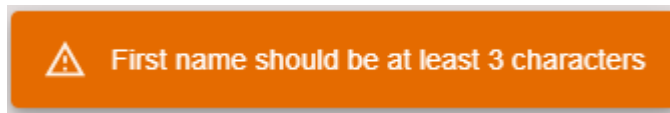


Figura 4.12 Exemplu eroare validare

Dacă toate datele sunt valide, funcția trimite o cerere HTTP POST la endpoint-ul `"/admin/register"` al serverului. Cererea include un header **'Authorization'** cu valoarea `'Bearer '` urmat de tokenul de sesiune al utilizatorului curent, și un corp care este o reprezentare JSON a obiectului de date.

Răspunsul serverului este tratat în funcție de codul său de stare. Dacă răspunsul are codul de stare 200 (OK), atunci înregistrarea a reușit și starea formularului este resetată la valorile inițiale. Dacă răspunsul are codul de stare 400, atunci înregistrarea a eșuat pentru că username-ul sau email-ul sunt deja utilizate, și se afișează un mesaj corespunzător în **Snackbar**. Orice alt cod de stare este considerat o eroare și se afișează un mesaj generic de eroare.

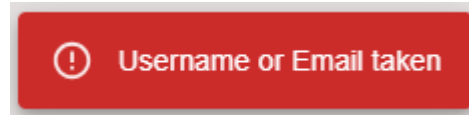


Figura 4.13 Exemplu eroare server

Dacă apare o excepție în timpul trimiterii cererii sau a procesării răspunsului, aceasta este prinsă și se afișează un mesaj de eroare în **Snackbar**.

În rezumat, procesul de înregistrare din perspectiva frontend-ului este următorul:

1. Utilizatorul completează formularul cu datele necesare înregistrării.
2. Când formularul este trimis, datele sunt validate.
3. Dacă datele sunt valide, se trimite o cerere de înregistrare la server.
4. Răspunsul serverului este procesat și utilizatorul este informat despre rezultatul înregistrării.

4.2.3 Crearea de facturi

Componenta React pentru crearea facturilor este un formular care conține câmpuri pentru tipul de factură, unitatea de măsură, costul pe unitate, numărul de unități, datele și dacă factura a fost plătită sau nu.

Funcția **handleSubmit** este responsabilă de gestionarea trimerii formularului. Validează informațiile introduse în formular și, dacă sunt valide, efectuează un request HTTP POST către backend, utilizând funcția **authFetch**, pentru a trimite informațiile facturii. Dacă requestul este reușit, toate câmpurile formularului sunt resetate și se afișează un mesaj de succes. Dacă apar erori în timpul procesării cererii, se afișează un mesaj de eroare.

The image shows a web form titled "Create new Bill". It contains the following fields and controls:

- Bill Type**: A dropdown menu with the placeholder text "-- select type --".
- Index**: A text input field containing the value "0".
- Unit of Measurement**: An empty text input field.
- Units**: A text input field containing the value "0".
- Cost Per Unit**: A text input field containing the value "0".
- From Date**: A date picker field showing "mm/dd/yyyy".
- To Date**: A date picker field showing "mm/dd/yyyy".
- Issue Date**: A date picker field showing "mm/dd/yyyy".
- Due Date**: A date picker field showing "mm/dd/yyyy".
- Paid**: A dropdown menu with the value "No".
- Submit**: A green button at the bottom of the form.

Figura 4.14 Formularul pentru crearea unei noi facturi

Funcția **handleBillTypeChange** gestionează schimbarea tipului de factură. În funcție de tipul selectat, setează unitatea de măsură și costul per unitate corespunzător, și resetează alte câmpuri dacă este necesar.

Interfața utilizator a formularului de facturi este construită folosind JSX, cu elemente de formular pentru fiecare câmp, cu etichete corespunzătoare. În funcție de tipul de factură selectat, unele câmpuri pot fi dezactivate.

Componenta **TopBarNavBar** implementează bara de navigație din partea de sus și din lateral a aplicației. Aceasta include un meniu desfășurabil pentru a naviga între diferite pagini ale aplicației. Componenta **BillCreation** este o componentă de nivel înalt care încorporează atât bara de navigație cât și formularul de factură, constituind astfel pagina de creare a facturilor.

4.2.4 Istoricul facturilor pe tip

Aplicația are structurat un set de pagini (sau „view-uri”), fiecare dedicată unui anumit tip de factură. Componenta **WaterView**, de exemplu, este o pagină dedicată afișării facturilor legate de apă.

În interiorul componentei **WaterView**, există două componente principale:

1. **TopBarNavBar**: Aceasta este componenta de navigație de nivel superior a aplicației. În funcție de selecția utilizatorului, aceasta permite schimbarea tipului de factură vizualizată (apă, electricitate, altele).
2. **BillsList**: Aceasta este componenta care afișează lista de facturi pentru un anumit tip specificat. În contextul componentei **WaterView**, tipul facturilor este stabilit la 'Water'.

Structura de mai sus permite afișarea tuturor facturilor de apă într-un mod organizat și ușor de înțeles pentru utilizator.

Similar, avem și alte pagini dedicate pentru diferite tipuri de facturi, cum ar fi **ElectricityView**, **GasView**, **InternetView** etc. Acestea au o structură similară cu **WaterView**, însă diferența esențială este dată de valoarea parametrului **billType** furnizat componentei **BillsList**.

De exemplu, în **ElectricityView**, componenta **BillsList** este invocată cu **billType** setat ca 'Electricity'. Aceasta semnifică faptul că, atunci când utilizatorul selectează 'Electricitate' din bara de navigație, componenta **ElectricityView** este afișată și astfel, sunt prezentate doar facturile legate de electricitate.

Aceeași abordare este urmată pentru toate tipurile de facturi pe care le gestionăm în aplicație. Astfel, avem o modularizare eficientă și o segregare clară a responsabilităților între diferitele componente ale aplicației, ceea ce duce la o mai bună întreținere a codului și la o experiență superioară pentru utilizator.

Componenta **BillsList** se ocupă de preluarea facturilor din backend și afișarea acestora în tabel. De asemenea, include logica pentru filtrarea facturilor plătite și neplătite prin intermediul unui checkbox. Ea trimite o solicitare GET la server, autenticată cu tokenul salvat în sesiune, pentru a prelua facturile de un anumit tip și în funcție de starea lor (plătită sau neplătită). De asemenea, gestionează afișarea unui mesaj de eroare dacă există probleme în comunicarea cu serverul.

Componenta **BillTable** se ocupă de afișarea facturilor într-un tabel. Tabelul include mai multe detalii despre facturi, inclusiv tipul facturii, prețul, dacă a fost plătită sau nu, numărul de unități, costul pe unitate, data scadenței, data emiterii, intervalul pentru care se aplică factura și un index. Include de asemenea o iconiță pentru a schimba starea unei facturi de la neplătită la plătită. Utilizează o funcție de paginare pentru a afișa doar un număr limitat de facturi pe pagină.

Water Bills History									
<input type="checkbox"/> Show Paid									
Bill Type	Price	Paid	Units	Cost Per Unit	Due Date	Issue Date	From Date	To Date	Index
Water	101.15	Yes	4111	0.02462	July 1, 2023, 00:00	June 15, 2023, 00:00	June 1, 2023, 00:00	July 1, 2023, 00:00	127679
Water	103.37	Yes	4202	0.02462	June 1, 2023, 00:00	May 15, 2023, 00:00	May 1, 2023, 00:00	June 1, 2023, 00:00	123477
Water	83.22	Yes	3381	0.02462	May 1, 2023, 00:00	April 15, 2023, 00:00	April 1, 2023, 00:00	May 1, 2023, 00:00	120096
Water	84.22	Yes	3422	0.02462	April 1, 2023, 00:00	March 15, 2023, 01:00	March 1, 2023, 01:00	April 1, 2023, 00:00	116674
Water	101.46	Yes	4123	0.02462	March 1, 2023, 01:00	February 15, 2023, 01:00	February 1, 2023, 01:00	March 1, 2023, 01:00	112551
Water	94.1	Yes	3822	0.02462	February 1, 2023, 01:00	January 15, 2023, 01:00	January 1, 2023, 01:00	February 1, 2023, 01:00	108729
Water	87.39	Yes	3551	0.02462	January 1, 2023, 01:00	December 15, 2022, 01:00	December 1, 2022, 01:00	January 1, 2023, 01:00	105178
Water	71.23	Yes	2893	0.02462	December 1, 2022, 01:00	November 15, 2022, 01:00	November 1, 2022, 01:00	December 1, 2022, 01:00	102285
Water	87.27	Yes	3548	0.02462	November 1, 2022, 01:00	October 15, 2022, 00:00	October 1, 2022, 00:00	November 1, 2022, 01:00	98737
Water	98.46	Yes	4001	0.02462	October 1, 2022, 00:00	September 15, 2022, 00:00	September 1, 2022, 00:00	October 1, 2022, 00:00	94736

Figura 4.15 Istoricul facturilor de apă

4.2.5 Crearea de cheltuieli

Componenta **ExpenseCreation** este o componentă simplă care afișează bara de navigație de sus și formularul de creare a cheltuielilor.

ExpenseForm este componenta principală care gestionează formularul de creare a cheltuielilor. Începe prin setarea unor valori de stare inițiale pentru diversele câmpuri ale formularului, cum ar fi tipul de cheltuială, titlul, atașamentele, dacă a fost plătită, prețul, nota și data. De asemenea, setează o serie de erori de validare la false.

Componenta folosește **useEffect** pentru a prelua tipurile de cheltuieli de la server atunci când componenta este montată. Acestea sunt stocate în starea **expenseTypes** pentru a fi utilizate în formular.

În funcția **handleSubmit**, se face validarea datelor introduse de utilizator și, în caz de erori, se setează starea de eroare corespunzătoare și se afișează un mesaj de eroare. Dacă toate datele sunt valide, se creează un obiect **FormData** care este apoi trimis la server cu o cerere HTTP POST pentru a crea noua cheltuială.

Create new Expense

Expense Type

Title

Price

Paid

Date

Note

Attachments

Figura 4.16 Formularul pentru crearea unei cheltuieli noi

Funcția **handleFileChange** gestionează adăugarea de noi atașamente la lista de atașamente atunci când utilizatorul alege fișiere utilizând butonul "Choose Files". Fișierele selectate sunt adăugate la starea **attachments**.

Funcția **removeFile** permite utilizatorului să elimine un atașament din lista de atașamente. Aceasta este apelată atunci când utilizatorul dă clic pe butonul "X" lângă numele fișierului.

Attachments

attachment1.txt attachment2.txt img1.png

Figura 4.17 Exemplu afișare atașamente

Formularul afișează câmpuri pentru utilizator să completeze diferitele detalii ale cheltuielii, precum tipul cheltuielii, titlul, prețul, dacă a fost plătită, data și nota. De asemenea, există o secțiune de atașamente unde utilizatorul poate adăuga fișiere.

În concluzie, acest formular de creare a cheltuielilor oferă un mod interactiv și ușor de utilizat pentru utilizatori să adauge noi cheltuieli. Acesta gestionează validarea datelor introduse de utilizator, afișarea mesajelor de eroare, adăugarea și eliminarea de atașamente, și trimiterea datelor către server pentru a crea noua cheltuială.

4.2.6 Istoricul cheltuielilor

Avem cinci componente principale: **ExpensesView**, **ExpensesOverview**, **ExpensesTable**, **ExpensesPieChart** și **ExpensesBarChart**.

Componenta **ExpensesView** este o componentă simplă care returnează un container ce include alte două componente, **TopBarNavBar** și **ExpensesOverview**.

În **ExpensesOverview** se gestionează starea aplicației, cu un focus pe gestionarea cheltuielilor și a statusului acestora, dacă sunt plătite sau nu. De asemenea, această componentă este responsabilă de afișarea mesajelor de eroare către utilizator în cazul în care interacțiunea cu serverul întâmpină probleme.

Pentru a realiza funcționalitatea de comutare între cheltuielile plătite și cele neplătite, se utilizează o bifă prin intermediul căreia starea este schimbată, determinând reîmprospătarea componentei și aducerea noilor date de la server.

În componenta **ExpensesTable** se realizează afișarea cheltuielilor într-un tabel, precum și paginarea acestora pentru a evita supraîncărcarea vizuală a utilizatorului. De asemenea, în această componentă se gestionează și atașamentele cheltuielilor, precum și schimbarea statusului cheltuielilor ca fiind plătite sau neplătite.

Expenses History						
<input checked="" type="checkbox"/> Show Paid						
Expense Type	Title	Price	Paid	Note	Date	Attachments
Other	Gift	60	Yes	Bought a gift for a friend.	June 20, 2023, 10:30	0
Entertainment	Theme Park Tickets	70	Yes	Went to a theme park.	June 5, 2023, 09:05	0
Food	Birthday Dinner	100	Yes	Celebrated a friend's birthday.	May 23, 2023, 11:00	0
Maintenance	Plumber Services	180	Yes	Fixed a leak in the bathroom.	May 10, 2023, 14:45	0
Travel	Flight Tickets	500	Yes	Purchased flight tickets for a trip.	April 20, 2023, 12:30	0
Education	Online Course	50	Yes	Bought a new course online.	April 8, 2023, 15:10	0
Shopping	Sneakers	120	Yes	Bought a new pair of sneakers.	March 22, 2023, 12:45	0
Healthcare	Doctor's Fee	70	Yes	Doctor's appointment.	March 1, 2023, 07:15	0
Entertainment	Concert Tickets	80	Yes	Went to a concert.	February 19, 2023, 11:05	0
Food	Restaurant Dinner	120	Yes	Enjoyed a great meal at a new restaurant.	February 12, 2023, 14:20	0

Figura 4.18 Istoricul cheltuielilor

Componentele **ExpensesPieChart** și **ExpensesBarChart** se ocupă de crearea și afișarea graficelor pie și bar. Aceste componente utilizează biblioteca **recharts**, care permite crearea de grafice SVG ușor de configurat și stilizat.

ExpensesPieChart afișează o reprezentare vizuală a cheltuielilor în funcție de tipul acestora. Componenta colectează datele necesare din lista de cheltuieli și le transformă într-un format adecvat pentru a fi afișate într-un grafic pie.

Pe de altă parte, **ExpensesBarChart** afișează un grafic bar care arată costurile cheltuielilor de-a lungul timpului. Componenta permite selectarea anului pentru care se dorește vizualizarea datelor, default fiind anul curent. Pentru generarea datelor

necesare graficului, se utilizează o funcție care agregă totalul cheltuielilor pentru fiecare lună a anului selectat.

4.2.7 Pagina principala

4.2.7.1 Secțiunea de facturi

Componenta **Dashboard** este pagina principală în care utilizatorii sunt redirecționați după ce s-au autentificat în aplicație. Această componentă conține alte trei componente importante - **TopBarNavBar**, **BillsOverview** și **ExpensesOverview**.

În acest moment, ne vom concentra asupra componentei **BillsOverview**, care afișează o prezentare generală a facturilor utilizatorului.

Începem cu definirea a câtorva stări ale componentei: **bills** pentru a stoca facturile primite de la server, **paid** pentru a stoca valoarea căsuței de selectare care controlează dacă se afișează facturile plătite sau nu, și **snackbar** pentru a afișa notificări către utilizator.

Metoda **useEffect** este folosită pentru a apela funcția **fetchData** la prima încărcare a componentei și oricând se schimbă starea **paid**. Această funcție este responsabilă pentru a trimite o cerere HTTP GET către server pentru a prelua toate facturile, filtrate pe baza valorii **paid**. Răspunsul este convertit în JSON și stocat în starea **bills**. Dacă se întâmplă o eroare, o notificare este afișată utilizatorului.

În ceea ce privește interfața cu utilizatorul, în partea de sus a componentei se afișează un titlu care se schimbă în funcție de valoarea **paid**, împreună cu o căsuță de selectare care permite utilizatorului să comute între facturile neplătite și istoricul facturilor. Atunci când utilizatorul face clic pe această căsuță, valoarea **paid** se schimbă, ceea ce declanșează reîncărcarea facturilor de la server.

În continuare, se afișează componenta **BillTable**, care primește ca proprietăți **bills** și **setBills**. Aceasta afișează facturile într-un tabel și permite utilizatorului să marcheze facturile ca plătite. După tabel, sunt prezentate două grafice - unul de tip bar și unul de tip pie - care oferă o reprezentare vizuală a facturilor.



Figura 4.19 Grafice facturi TODO

La sfârșit, avem componenta **CustomSnackbar**, care este responsabilă pentru afișarea notificărilor către utilizator.

În componenta **BillTable**, se folosește o paginare pentru a afișa facturile într-un mod mai organizat și ușor de navigat pentru utilizator. Facturile sunt afișate într-un tabel, cu un rând pentru fiecare factură și coloane pentru fiecare proprietate relevantă a facturii. Fiecare rând include, de asemenea, o pictogramă care permite utilizatorului să marcheze o factură ca plătită. Atunci când acest lucru se întâmplă, o cerere HTTP POST este trimisă către server pentru a actualiza statutul facturii și starea locală **bills** este actualizată pentru a reflecta această schimbare.

4.2.7.2 Secțiunea de cheltuieli

Componenta **ExpensesOverview** este foarte similară cu **BillsOverview**, cu unele mici diferențe specifice cheltuielilor.

Precum **BillsOverview**, **ExpensesOverview** are trei stări importante: **expenses** pentru stocarea cheltuielilor primite de la server, **paid** pentru a stoca valoarea căsuței de selectare care controlează dacă se afișează cheltuielile plătite sau nu, și **snackbar** pentru afișarea notificărilor către utilizator.

Metoda **useEffect** este folosită pentru a apela funcția **fetchData** la prima încărcare a componentei și oricând se schimbă starea **paid**. Această funcție este responsabilă pentru a trimite o cerere HTTP GET către server pentru a prelua toate cheltuielile, filtrate pe baza valorii **paid**. Răspunsul este convertit în JSON și stocat în starea **expenses**. Dacă se întâmplă o eroare, o notificare este afișată utilizatorului.

În ceea ce privește interfața cu utilizatorul, în partea de sus a componentei se afișează un titlu care se schimbă în funcție de valoarea **paid**, împreună cu o căsuță de selectare care permite utilizatorului să comute între cheltuielile neplătite și istoricul cheltuielilor. În continuare, se afișează componenta **ExpensesTable**, care primește ca proprietăți **expenses** și **setExpenses**. Aceasta afișează cheltuielile într-un tabel și permite utilizatorului să marcheze cheltuielile ca plătite. Apoi, sunt prezentate două grafice - unul de tip bar și unul de tip pie - care oferă o reprezentare vizuală a cheltuielilor. La sfârșit, avem componenta **CustomSnackbar**, care este responsabilă pentru afișarea notificărilor către utilizator.

În **ExpensesTable**, paginarea este folosită pentru a afișa cheltuielile într-un mod mai organizat și ușor de navigat pentru utilizator. Cheltuielile sunt afișate într-un tabel, cu un rând pentru fiecare cheltuială și coloane pentru fiecare proprietate relevantă a cheltuielii. Fiecare rând include, de asemenea, o pictogramă care permite utilizatorului să marcheze o cheltuială ca plătită. Atunci când acest lucru se întâmplă, o cerere HTTP POST este trimisă către server pentru a actualiza statutul cheltuielii și starea locală **expenses** este actualizată pentru a reflecta această schimbare. Fiecare rând include și o pictogramă care deschide un popup cu atașamentele asociate cheltuielii, care permite vizualizarea și descărcarea acestora..

4.2.8 Descarcarea atasamentelor

Componenta **AttachmentsPopup** este o componentă care se deschide atunci când un utilizator dă clic pe pictograma din coloana de atașamente. Acesta afișează un popup cu atașamentele asociate cheltuielii respective și oferă opțiunea de a descărca fiecare atașament.

Componenta are două stări principale: **snackbar**, care controlează notificările către utilizator, și **token**, care este folosit pentru a autoriza descărcarea atașamentelor.

Funcția **handleDownload** este declanșată atunci când utilizatorul dă clic pe butonul de descărcare al unui atașament. Aceasta trimite o cerere HTTP GET către server, solicitând atașamentul specificat prin **expenseId** și **attachmentId**. Dacă răspunsul este OK, răspunsul este convertit într-un blob și un URL este generat pentru acest blob. Apoi, un link ascuns este creat în documentul HTML și este declanșat un clic pe acesta, care duce la descărcarea atașamentului.

Funcția **handleOutsideClick** este folosită pentru a închide popupul atunci când se face clic în afara acestuia. Aceasta este adăugată ca un listener de evenimente la document atunci când componenta este montată și este înlăturată când componenta este demontată, pentru a preveni scurgerile de memorie.

În ceea ce privește interfața cu utilizatorul, componenta afișează un titlu, urmat de o listă de atașamente. Fiecare atașament este reprezentat de un nume de fișier și un buton de descărcare. La sfârșit, există un buton de închidere care închide popupul. Dacă se întâmplă o eroare în timpul descărcării, se afișează o notificare către utilizator.

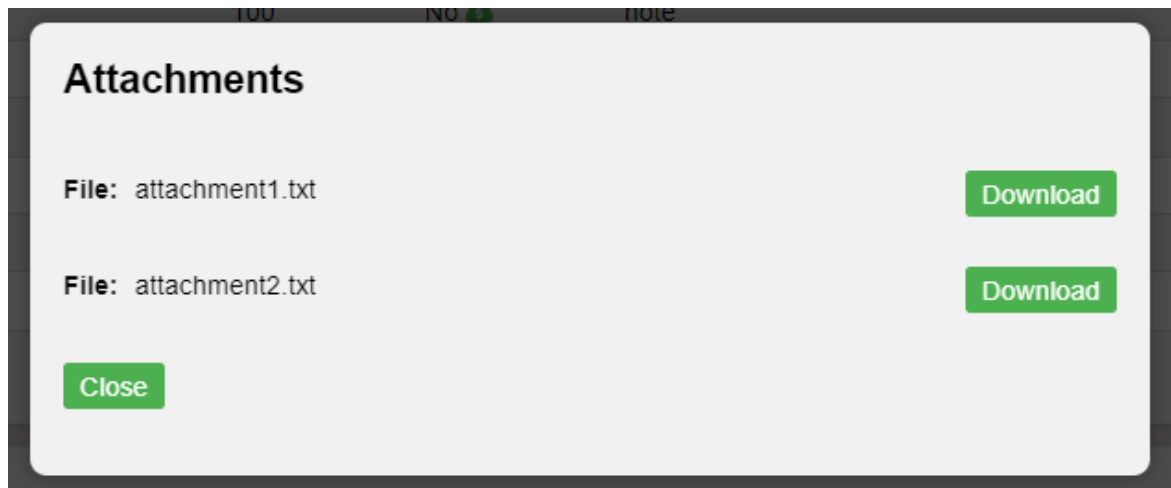


Figura 4.20 Popup atașamente

4.3 Baza de date

Un aspect fundamental al aplicației este reprezentat de modelul de date și de modul în care sunt stocate și interconectate diferitele entități. Aplicația noastră utilizează o bază de date relațională pentru a stoca și a gestiona datele, bazându-se pe șase tabele cheie pentru a reprezenta entitățile sistemului: Users, Roles, Bills, Expenses și Attachments.

- Tabela **“users”** reprezintă inima sistemului și stochează informațiile despre fiecare utilizator al aplicației. Informațiile pentru fiecare utilizator includ un ID unic, prenume, nume, nume de utilizator, parolă, codul clientului, codul de facturare, adresa și adresa de e-mail. În plus, fiecare utilizator are un set de roluri asociate, care sunt reprezentate și stocate în tabela **Roles**.
- Tabela **“roles”** conține diferitele roluri pe care le poate avea un utilizator în cadrul sistemului. Rolurile pot fi **“ADMIN”** sau **“USER”**. Fiecare rol este identificat printr-un ID unic și are un nume descriptiv. Pentru a realiza asocierea între utilizatori și rolurile lor, se utilizează o tabelă de legătură numită **“user_roles”**.
- Tabela **“bills”** este responsabilă pentru stocarea informațiilor despre facturile adăugate de fiecare utilizator. Fiecare înregistrare (sau "rând") din această tabelă reprezintă o factură unică și include un ID unic, tipul facturii, numărul de unități facturate, costul pe unitate, intervalul de timp pentru care este emisă factura (dată de început și de sfârșit), data emiterii facturii, data scadenței, un index, un indice de utilizator, un câmp pentru a indica dacă factura a fost sau nu plătită, și prețul total.
- Tabela **“expenses”** reține informațiile despre cheltuielile efectuate de fiecare utilizator. Asemenea tabelii **“bills”**, fiecare înregistrare în această tabelă reprezintă o cheltuială unică și include un ID unic, tipul cheltuielii, titlul, un câmp pentru a indica dacă cheltuiala a fost sau nu plătită, prețul, o notă legată de cheltuială, data la care a fost efectuată cheltuiala și orice fișiere atașate legate de cheltuială.
- Tabela **“attachments”** servește ca depozit pentru fișierele atașate cheltuielilor. Fiecare înregistrare din această tabelă reprezintă un atașament unic, fiind identificat printr-un ID unic. Alte informații despre atașament includ datele efective ale fișierului (stocate ca un array de bytes), numele fișierului, tipul fișierului și ID-ul cheltuielii cu care este asociat.

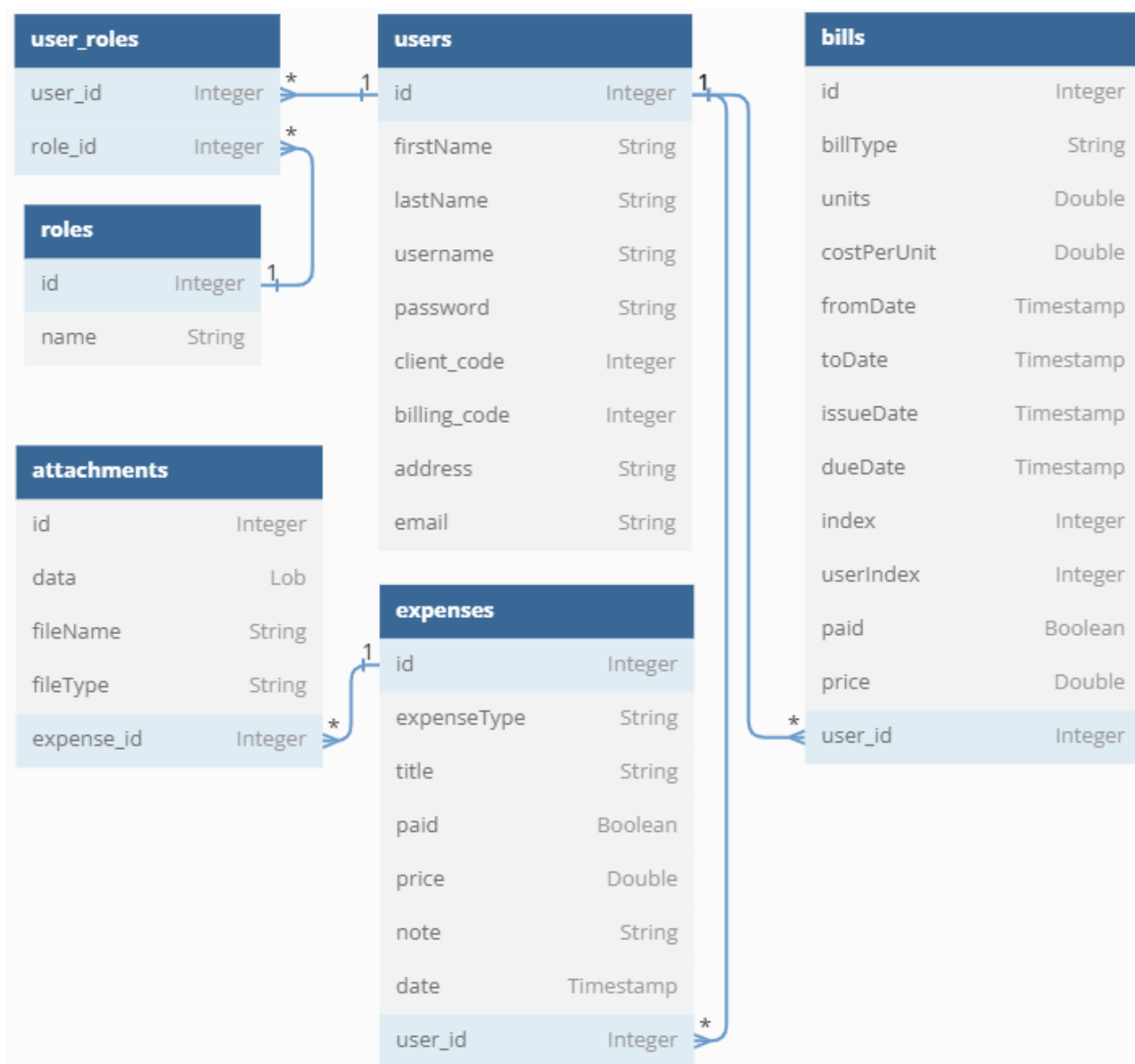


Figura 4.21 Schema bazei de date

5 Concluzii

5.1 Rezultate obținute

Finalizarea acestei lucrări a condus la dezvoltarea unui sistem robust de gestionare a facturilor și cheltuielilor. Aplicația este capabilă să ofere o imagine detaliată și actualizată asupra istoricului financiar al utilizatorului, informațiile fiind prezentate într-o manieră accesibilă și ușor de înțeles.

Sistemul este capabil să stocheze, să prelucreze și să prezinte datele utilizatorului într-un mod eficient și securizat, fie că acesta alege să utilizeze aplicația local sau de la distanță.

În pofida eforturilor de securitate, o provocare a fost securitatea datelor. Deși se folosesc tehnici moderne de securizare, precum JWT pentru autentificare și HTTPS pentru comunicația securizată, există mereu riscul ca anumiți actori rău intenționați să încerce să acceseze informațiile utilizatorilor. Cu toate acestea, nu sunt stocate date cu caracter personal sau informații sensibile, ceea ce asigură că utilizatorii nu sunt expuși unor riscuri semnificative.

5.2 Direcții de dezvoltare

Există numeroase oportunități de dezvoltare și îmbunătățire a sistemului. O direcție esențială ar fi consolidarea securității, prin introducerea unor funcționalități suplimentare de autentificare, precum autentificarea în doi pași.

O altă direcție de dezvoltare ar fi posibilitatea de a suporta mai mulți utilizatori cu conturi diferite. Aceasta ar face ca aplicația să fie utilă pentru familii sau grupuri de colegi de apartament, care doresc să își gestioneze cheltuielile comune într-un mod eficient.

Un alt aspect interesant ar fi integrarea cu alte sisteme financiare, cum ar fi băncile sau platformele de plată online. Acest lucru ar putea oferi funcționalități suplimentare, cum ar fi posibilitatea de a plăti facturi direct din aplicație sau de a importa date despre tranzacții în mod automat.

De asemenea, sistemul ar putea fi extins cu un modul de raportare mai avansat, care să ofere analize mai detaliate asupra datelor. Acesta ar putea include grafice sau diagrame care să ilustreze evoluția cheltuielilor pe categorii, sau să compare cheltuielile din diferite perioade de timp.

În concluzie, această aplicație oferă o bază solidă pentru gestionarea eficientă a facturilor și cheltuielilor și are un potențial semnificativ de dezvoltare și îmbunătățire.

6 Bibliografie

- R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures,” University of California, Irvine, 2000.
- 1] G. Gitonga, "LinkedIn," [Online]. Available:
2] <https://www.linkedin.com/pulse/understanding-rest-architecture-gabriel-gitonga/>.
- G. E. Krasner și S. T. Pope, „A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80,” *Journal of Object-Oriented Programming*, vol. 1, nr. 3, pp. 26-49, 1988.
- 3] A. Maurício, B. Gabriele și T. Christoph, „Code smells for Model-View-Controller architectures,” *Empirical Software Engineering*, nr. 23, pp. 2021-2157, 2018.
- 4] L. Richardson și R. Sam, „RESTful Web Services,” O'Reilly Media, Inc., 2007.
- 5] Oracle Corporation, "The Java® Language Specification," *Java SE 17 Edition*.
- 6] Oracle Corporation, "The Java™ Virtual Machine Specification," *Java SE 17 Edition*.
- 7] Oracle Corporation, "Java Garbage Collection Basics".
- 8] Oracle Corporation, „Garbage Collector Tuning”.
- 9] Oracle Corporation, "Java Development Kit (JDK)".
- 10] Oracle Corporation, "Java Runtime Environment (JRE)".
- 11] Mozilla Developer Network, "HTML basics," [Online]. Available:
12] https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics.
- C. Walls, „Spring in Action,” Manning Publications, 2019.
- 13] P. Priya, "EDUCBA," [Online]. Available: <https://www.educba.com/spring-architecture/>.
- 14]

- G. Turnquist, in *Learning Spring Boot 2.0*, Packt Publishing, 2017.
- 15] "InterviewBit," [Online]. Available:
16] <https://www.interviewbit.com/blog/spring-boot-architecture/>.
- Apache Maven Project, 2023. [Online]. Available:
17] <https://maven.apache.org/guides/>.
- M. Fowler, "Continuous Integration," martinfowler.com, 15 2006. [Online].
18] Available: <https://martinfowler.com/articles/continuousIntegration.html#ConfigurationManagement>.
- H. Kabutz, "Java Specialists' Newsletter," 2023. [Online]. Available:
19] <https://www.javaspecialists.eu/archive/archive.jsp>.
- Facebook, "React - A JavaScript library for building user interfaces,"
20] reactjs.org, [Online]. Available: <https://reactjs.org>.
- "Understanding the React Component Lifecycle," [Online]. Available:
21] <https://www.telerik.com/kendo-react-ui/react-introduction-guide/>.
- IBM, "<https://developer.ibm.com/tutorials/wa-react-intro/>," [Online].
22]
- T. Kadlec, "React.js in Patterns," [Online]. Available:
23] <https://github.com/krasimir/react-in-patterns>.
- PostgreSQL Global Development Group, "What is PostgreSQL," [Online].
24] Available: <https://www.postgresql.org/about/>.
- PostgreSQL Global Development Group, "PostgreSQL 13.3 Documentation,"
25] [Online]. Available: <https://www.postgresql.org/docs/13/index.html>.
- JetBrains, "IntelliJ IDEA," [Online]. Available:
26] <https://www.jetbrains.com/idea/>.
- JetBrains, "WebStorm," [Online]. Available:
27] <https://www.jetbrains.com/webstorm/>.
- Postman, "What is Postman?," [Online]. Available:
28] <https://www.postman.com/product/api-client/>.