

APLICAȚIE WEB PROGRESIVĂ PENTRU STOCAREA FIȘIERELOR

Candidat: Marian-Daian Petrica

Coordonator științific: Șl.dr.ing Raul Robu

Sesiunea: Iunie 2022

CUPRINS

CUPRINS	2
1. INTRODUCERE	4
1.1 Contextul actual	4
1.2 Motivele realizării proiectului	4
1.3 Soluția propusă în raport cu aplicații similare	5
2. FUNDAMENTARE TEORETICĂ	8
2.1 Limbajul de programare Java	8
2.2 Framework-ul Spring	9
2.3 Spring Boot	10
2.4 Spring Data JPA și interfețele Repository	11
2.5 Limbajul de programare TypeScript	13
2.6 Framework-ul Angular	14
2.7 Componente	15
2.8 Servicii	18
2.9 Angular Guards	19
2.10 RxJs	19
2.11 Angular Material	21
2.12 SCSS	22
2.13 Progressive Web Application	23
2.14 Comunicarea între aplicații folosind HTTP	24
2.15 Comunicarea folosind Websockets	24
2.16 Git și GitHub	26
2.17 Găzduirea aplicației pe server	27
3. PROIECTAREA TEHNICĂ A APLICAȚIEI	29
3.1 Diagrame UseCase	29
3.2 Arhitectura generală	31
3.3 Arhitectura server-ului	32
3.3.1 Securitatea în FileStorm	33
3.3.2 Filtre	34
3.3.3 Controlere	39
3.3.4 Servicii	52
3.3.5 Repositories	63
3.4 Arhitectura UI-ului	65
3.4.1 Routes	66
3.4.2 Guards	67
3.4.3 Componente	68
3.4.4 Servicii	76
3.5 Testarea performanței	77
4. UTILIZAREA APLICAȚIEI	80

4.1 Necesități hardware și software	80
4.2 Instalare și găzduire	80
4.3 Autentificarea & logarea	82
4.4 Cererea și atribuirea spațiului de stocare	82
4.5 Operații CRUD cu fișiere	84
4.6 Operații cu directoare	85
4.7 Operatii cu utilizatori	85
5. CONCLUZII	87
BIBLIOGRAFIE	88

1. INTRODUCERE

1.1 Contextul actual

Aplicația propusă, denumită FileStorm, reprezintă o soluție de stocare a fișierelor ce au nevoie să fie accesate sau descărcate de pe diferite dispozitive, caz care în contextul actual nu este unul izolat. Fie că este vorba de încărcarea unei fotografii facută cu telefonul de care urmează să avem nevoie și pe PC, stocarea arhivelor de dimensiuni mari, distribuirea anumitor fișiere sau încarcarea documentelor într-un mediu care să ia în considerare confidențialitatea datelor, nevoia pentru acest gen de platformă este clară.

În momentul de față există multiple metode de stocare și distribuire a fișierelor, însă fiecare dintre acestea are anumite curențe pe care aplicația propusă, le rezolvă.

Printre cele mai notabile caracteristici ale acesteia se enumera : disponibilitatea pe toate platformele, cerințele reduse din punct de vedere al hardware-ului, independența față de sistemele altor corporații, confidențialitatea datelor, independența față de magazinele software, performanța ridicată și design-ul modern.

Unul dintre motivele pentru care atât de multe astfel de servicii de stocare (de ex. GoogleDrive, Dropbox, etc.) nu oferă toate calități înafara unui abonament, este nivelul ridicat în ceea ce privește complexitatea dezvoltării și menținerii unei astfel de platforme, din punct de vedere al serviciilor software.

Bineînțeles, hardware-ul jucă un rol foarte important în genul acesta de servicii, însă acesta devine din ce în ce mai accesibil unui public foarte larg și mi se pare o oportunitate excelentă în a avea un serviciu de stocare și distribuire a datelor complet funcțional și modern care să aparțină în totalitate utilizatorului.

1.2 Motivele realizării proiectului

Aplicația implementată FileStorm, are ca scop realizarea unei soluții de stocare a datelor total independentă față de alte servicii, de ex. Cloud care să asigure intimitate, o interfață modernă disponibilă pe mai multe platforme, ușor de utilizat și administrabilă.

De asemenea, platforma a fost concepută și cu scopul de a utiliza sisteme de calcul și de stocare, fie ele destinate utilizării pe perioade lungi de timp de ex. un home server cât și pentru a oferi un nou scop dispozitivelor scoase din uz. Din punct de vedere al utilizării resurselor programul este foarte eficient astfel, aproape orice computer personal poate fi utilizat ca și gazdă pentru această aplicație.

Totodată, această platformă a reprezentat un efort semnificativ, deoarece în realizarea acesteia am luat în considerare atât performanța cât și designul atractiv, utilizând cele mai moderne tehnologii și metode de dezvoltare a aplicațiilor web.

În final, ca și motiv principal, trebuie luată în calcul și confidențialitatea datelor, care odată cu apariția acestor servicii a fost neglijată, iar aplicația dezvoltată, își propune să funcționeze ca și un spațiu de stocare și distribuire a datelor, 100% confidențial și administrabil, fără să fie necesară intervenția altor servicii.

1.3 Soluția propusă în raport cu aplicații similare

Printre aplicațiile similare pe care le-am luat în considerare se enumeră : Google Drive, DropBox, OneDrive.

Ca și similarități, Google Drive este cel mai apropiat din punct de vedere estetic, lucru care se datorează utilizării standardului Material la nivelul aplicației FileStorm. Acest standard conține diverse componente des utilizate în realizarea interfețelor grafice pe web și este în cea mai mare parte, un proiect susținut chiar de către Google, așa că similitudinile din design nu sunt întâmplătoare.

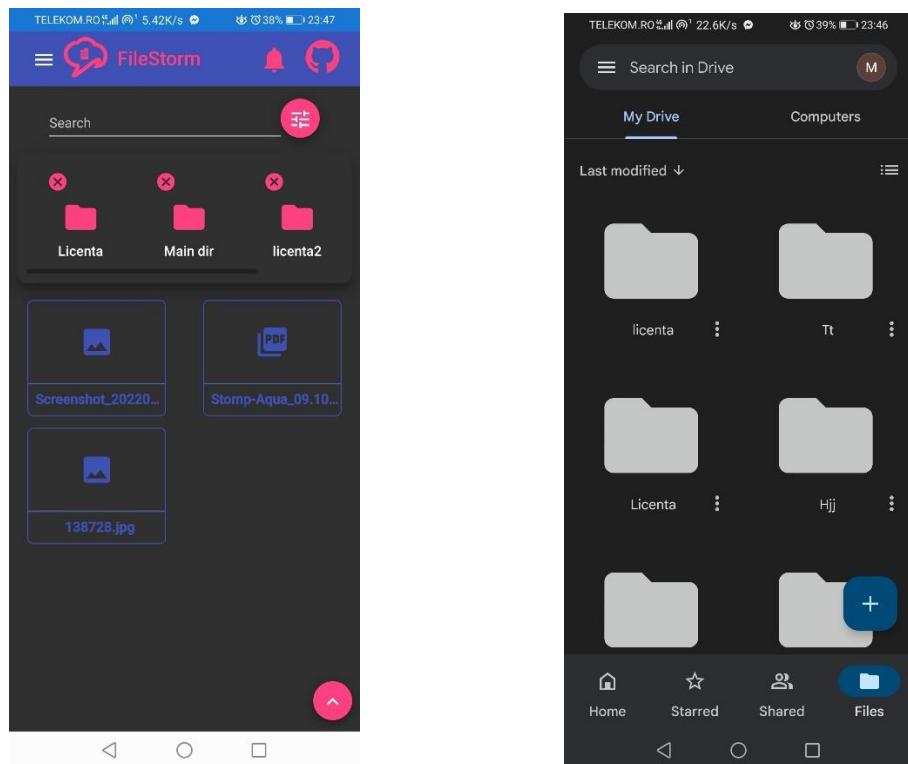


Figura 1.3 – Comparație între FileStorm în varianta PWA și aplicația de android Google Drive.

O caracteristică definitorie ce reiese din captura de ecran din partea stângă o reprezintă ordonarea directoarelor într-un chenar poziționat în aşa fel încât, să nu ocupe prea mult spațiu pe ecran.

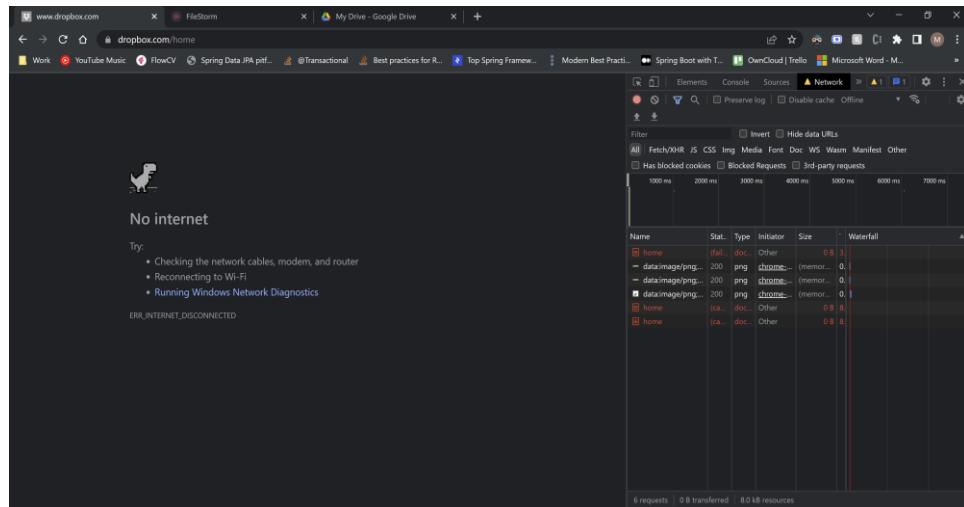


Figura 1.4 – Se observă faptul că pe aplicația web DropBox, în momentul în care conexiunea la internet este sistată, acesta nu mai are nici un fel de funcționalitate. În tab-ul de Network din browser, se observă că este simulație modul „Offline”.

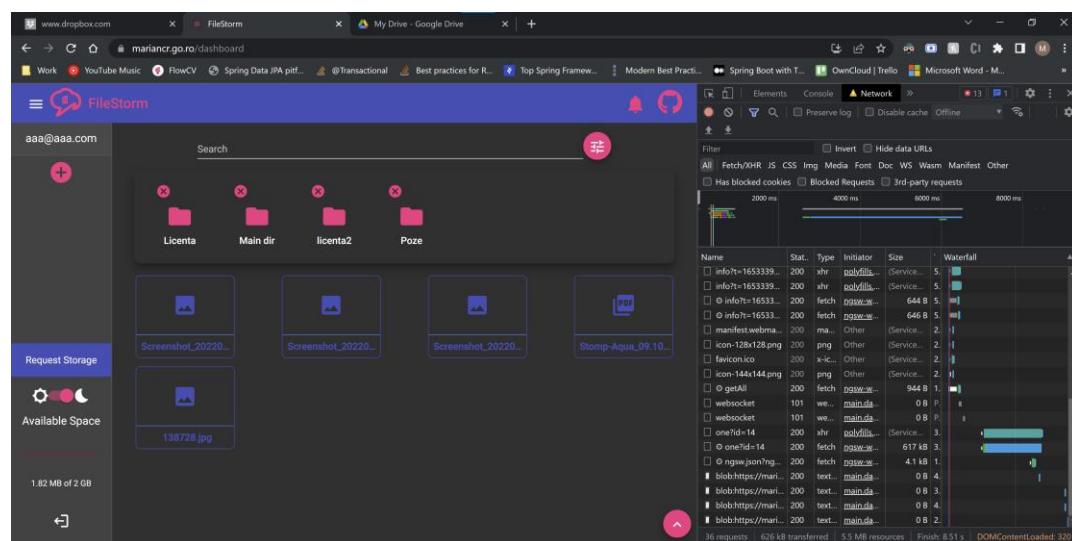


Figura 1.5 – Se observă faptul că, deși aplicația este în modul „Offline”, aceasta încă se încarcă și are disponibile o parte din funcționalități.

În ceea ce privește aplicația DropBox, avantajul pe care FileStorm îl are este abilitatea de a salva resurse în memoria cache a browser-ului, astfel, pe lângă faptul că acest concept aduce și beneficii semnificative de performanță, mai există un beneficiu și

anume faptul că platforma are în continuare o parte din funcționalitățile ei de bază, în momentul în care se întrerupe conexiunea. De exemplu, se pot vedea fișiere deschise anterior, se poate naviga prin directoare, se poate schimba tema paginii, se poate observa cantitatea de stocare utilizată și.a.m.d.

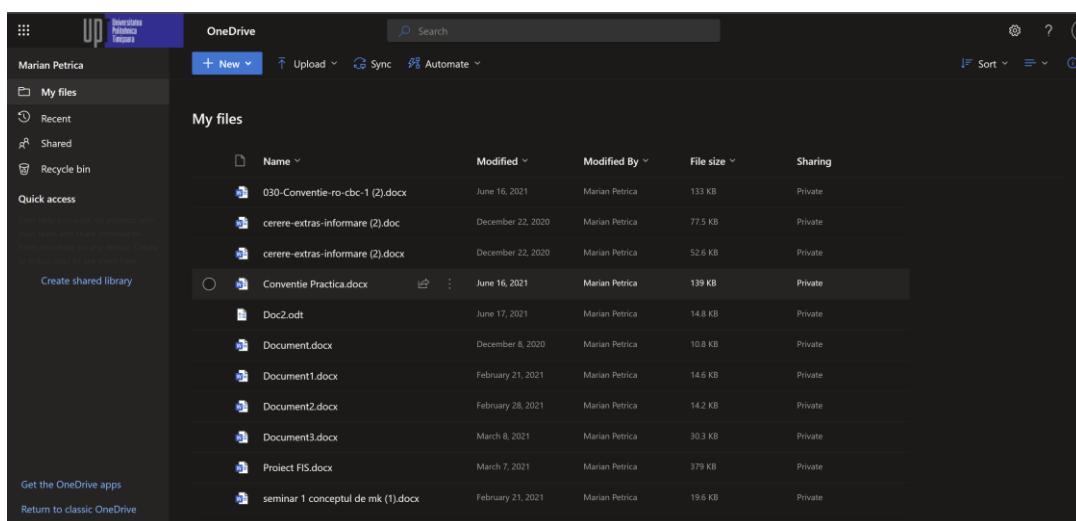


Figura 1.6 – Aplicația web a platformei OneDrive

Platforma OneDrive, pe lângă faptul că nu oferă nici ea la rândul ei, disponibilitate offline, are și un design mai diferit fata de FileStorm.

Pe lângă aceste diferențe, fie ele relevante pentru unii utilizatori sau nu, o funcționalitate lipsește cu desăvârșire acestor aplicații de stocare în cloud, și anume, utilizarea acestor platforme, fără necesitatea unui abonament în cadrul aceluiasi serviciu de Cloud. Toate aceste aplicații, oferă inițial un spațiu limitat de stocare, ce mai apoi poate fi suplimentat prin intermediul unui abonament și nu există o versiune a acestor aplicații care să permită instalarea lor pe diferite alte dispozitive alese de către un potențial utilizator (de ex. laptop personal, computer personal, home server).

Astfel, cea mai importantă caracteristică a aplicației dezvoltate FileStorm, este faptul că e o platformă, destinată în principal, utilizării acestor dispozitive personale ca și gazdă pentru a avea controlul absolut asupra fișierelor, a datelor și a activității personale, având o performanță sporită și un design atractiv. Cu toate acestea, aplicația poate fi găzduită pe orice serviciu de Cloud oferit de majoritatea platformelor, în cazul în care se dorește asta, lucru datorat utilizării platformei Java în componența serverului, limbaj de programare ce poate fi rulat indiferent de sistemul de operare.

2. FUNDAMENTARE TEORETICĂ

2.1 Limbajul de programare Java

Java reprezintă un limbaj de programare orientat pe obiecte, multi-platformă și puternic tipizat, conceput de James Gosling și ulterior lansat în 23 ianuarie 1996, care a fost cumpărat de compania Oracle. Odată cu achiziția unei alte companii, responsabile de realizarea limbajului, numită Sun Microsystems [1].



```

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}

```

Figura 2.1 – Exemplu de program în limbajul Java.

Cel mai important beneficiu al acestui limbaj este portabilitatea, faptul că putem scrie cod pe un dispozitiv, iar respectivul cod poate fi rulat pe alt dispozitiv cu o altfel de arhitectură. Acest lucru se datorează așa-numitului "Java bytecode". Aceasta este un cod intermediar, generat în procesul de compilare care este analog codului mașină, doar că în loc să fie rulat direct, acesta este mai întâi executat de către o mașină virtuală, numită Java Virtual Machine care trebuie să fie instalată inițial pe dispozitiv.

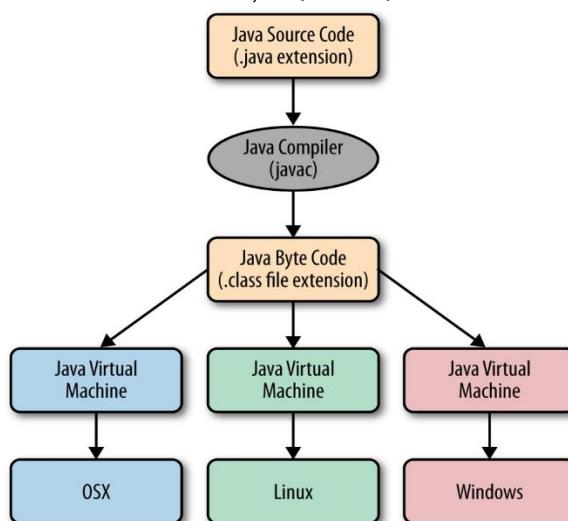


Figura 2.2 – Procesul de rulare a codului în limbajul Java [2]

Din cauza acestui proces mai complex de executare a codului, în general, limbajele bazate pe o mașină virtuală sunt considerate mai încete, însă odată cu aducerea în discuție a

compilatoarelor Just In Time (JIT), capabile să genereze cod mașină la runtime această diferență a timpului de execuție a fost semnificativ redusă.

2.2 Framework-ul Spring

Așa-numitul „Spring Framework” este un cadru de dezvoltare a aplicațiilor în limbajul de programare Java[3]. Un cadru de lucru sau Framework mai bine spus, reprezintă o colecție de biblioteci, programe și/sau diverse metodologii, care au ca și scop principal reducerea nivelului de dificultate în ceea ce privește dezvoltarea aplicațiilor de dimensiuni considerabile. Astfel, Spring pune la dispoziție o întreagă infrastructură pe baza căreia o aplicație se poate dezvolta.

Principalele concepte ale acestui framework sunt injectarea dependințelor și inversarea controlului.

La nivelul unei aplicații simple Java, se creează numeroase obiecte, unele dintre acestea existând și în relații de interdependentă, care, odată cu avansarea în complexitate a aplicației, devin din ce în ce mai greu de controlat. Aceste relații de interdependentă pot duce la îngreunarea procesului de dezvoltare al unei aplicații, așa că Spring utilizează un mecanism, prin care fiecare obiect, denumit „Bean”, poate fi injectat în cadrul altor obiecte astfel încât relațiile de dependentă să fie satisfăcute. Realizarea acestui concept se datorează existenței unei piese esențiale ale framework-ului Spring, așa-numitul „Spring Application Context”.

Prin utilizarea conceptului de injectare a dependințelor, practic tot ce ține de crearea, administrarea și satisfacerea dependințelor dintre obiecte nu mai reprezintă o sarcină a programatorului ci a framework-ului iar această proprietate poartă numele de „Inversion of Control”.

```
@Service
@RequiredArgsConstructor
@Slf4j
public class UserService implements UserDetailsService {
    14 usages
    private final UserRepository userRepository;
    3 usages
    private final UserMapper userMapper;

    1 usage  ↳ mariannpmnd
    public List<UserDTO> getAllUsers() {
        List<UserEntity> all = userRepository.findAll();
        return userMapper.entitiesToDTOs(all);
    }
}
```

Figura 2.3 – Exemplu de clasa ce are injectate 2 dependințe

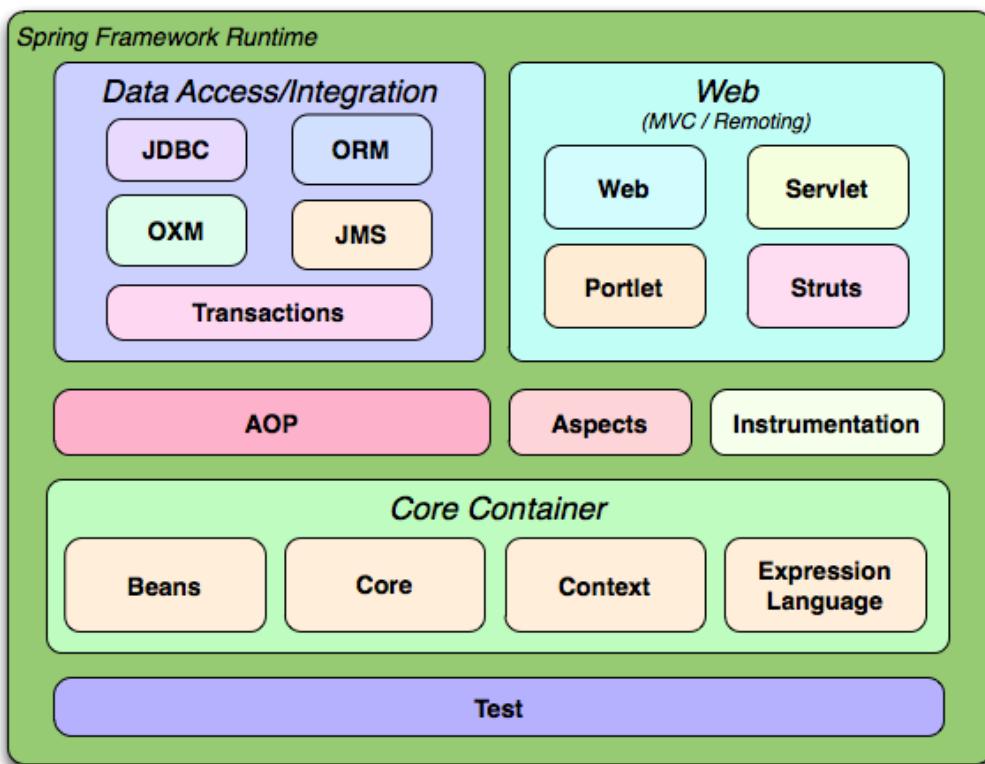


Figura 2.4 – Diagrama tuturor modulelor din arhitectura Framework-ului Spring[3]

2.3 Spring Boot

Luând în considerare toate capabilitatile puse la dispoziție de către Spring, este normal ca la nivelul Framework-ului să apară diverse alte dificultăți. Cea mai problematică parte a acestuia o reprezintă configurarea. Desi în esență sa, Spring este extrem de configurabil, acest aspect poate duce adesea, la diverse alte complicații și astfel, definirea mai strictă a unui mod de lucru și realizarea implicită a unor configurații esențiale devine extrem de dezirabilă.

Astfel, Spring Boot reprezintă un proiect bazat pe Frameworkul Spring ce vine cu diverse îmbunătățiri precum autoconfigurarea și abilitatea de a realiza aplicații web de sine stătătoare fară să se pună prea mult accent pe partea de configurare.

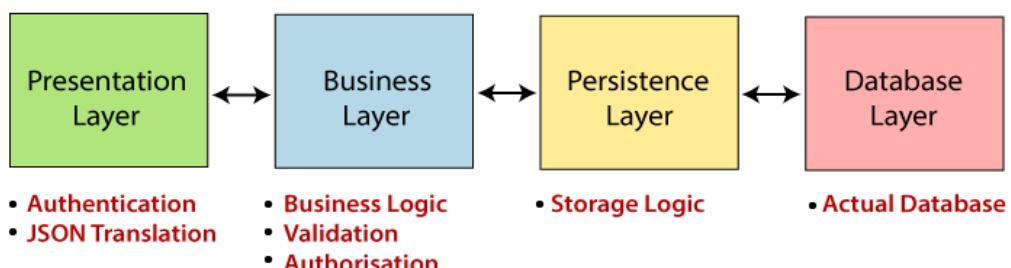


Figura 2.5 – Modelul arhitectural Spring Boot [4]

Arhitectura Spring Boot, la nivelul unei aplicații web se poate defini că având 4 niveluri esențiale : Baza de date, nivelul de persistență, nivelul serviciilor sau „Business Layer” și nivelul de prezentare.

Această abordare structurată pe niveluri este mult mai simplă și mai facilă, fiind astfel posibilă dezvoltarea diverselor aplicații cu un minim de efort față de platforma de baza Spring.

În cadrul acestui proiect am ales să utilizez baza de date PostgreSQL pentru stocarea datelor legate de utilizatori, fișiere, directoare și notificări.

Nivelul de persistență este util în realizarea interogărilor la nivelul bazei de date folosindu-se de un concept numit ORM (Object Relational Mapping) implementat de către Framework-ul Hibernate care este inclus în dependința Spring Data JPA.

Nivelul de servicii este responsabil cu procesarea informațiilor venite de la nivelul anterior, validarea datelor și în general toată logica de business a aplicației.

Nivelul de prezentare are rolul de a expune datele procesate de către servicii realizând în prealabil procesul de marshalling, adică translatarea informațiilor din forma pe care acestea o au în memorie, într-o formă mai utilă pentru aplicațiile care au nevoie de informații, în cazul FileStorm acest format este reprezentat de JSON (JavaScript Object Notation).

2.4 Spring Data JPA și interfețele Repository

Spring Data JPA este o dependință foarte utilizată în ecosistemul Spring. Aceasta aduce o multitudine de facilități pentru lucrul cu baze de date. De asemenea aceasta are la bază un alt framework, denumit Hibernate ce se bazează pe paradigma ORM (Object Relational Mapping).

Spre deosebire de lucrul clasic cu baze de date relationale, ce se bazează pe propoziții SQL pentru a se crea interogări care mai apoi sunt interpretate de baza de date și în baza acestora se returnează date, Spring Data JPA se folosește de Hibernate și Reflection API pentru a ușura aceste proceduri.

În primul rând, Hibernate este o abstractizare a JDBC (Java Database Connectivity), care la randul său, este un API de bază în comunicarea cu o baza de date în limbajul Java. Aceasta este foarte performant, dar greu de folosit din cauza seturilor de date nestructurate, a menținerii conexiunilor cu bazele de date și.a.m.d. Astfel, Hibernate vine cu o metodă mai familiară și mai simplă de reprezentare a seturilor de date rezultate în urma interogărilor. Aceasta se folosește de clase anotate cu @Entity pentru a defini o reprezentare a unui tabel de date.

```
@Entity
@Getter
@Setter
@ToString
@RequiredArgsConstructor
public class UserEntity implements Principal {
    3 usages
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;

    2 usages
    private String email;

    1 usage
    @Transient
    private String username;
    1 usage
    private String password;
    1 usage
    private String role;
    private BigInteger assignedSpace = BigInteger.ZERO;
    private BigInteger occupiedSpace = BigInteger.ZERO;

    @OneToMany(mappedBy = "user")
}
```

Figura 2.6 – Clasa-entitate UserEntity.

Observăm că și exemplu clasa definită în figura de mai sus, aceasta este anotată cu `@Entity` și are diverse attribute. Fiecare atribut reprezintă o coloană din tabelul asociat acestei entități. De asemenea, Hibernate permite definirea a diverse tipuri de relații între tabele (One to Many, One to One ...).

Pe lângă aceste facilități aduse de Hibernate, Spring Data JPA adaugă și interfețe numite Repositories. Acestea trebuie să fie anotate cu `@Repository` și de regulă extind o altă interfață de bază de ex: `JpaRepository`, `PagingAndSortingRepository` și.a.m.d.

```
@Repository
public interface DirectoryRepository extends JpaRepository<DirectoryEntity, Long> {
    1 usage  ▲ Petrica Marian
    List<DirectoryEntity> findByUser(UserEntity user);
    2 usages  ▲ Petrica Marian
    Optional<DirectoryEntity> findByPath(String path);
    1 usage  ▲ marianpmd
    List<DirectoryEntity> findByPathContainsAndUser(String path, UserEntity user);
}
```

Figura 2.7 – Definirea repository-ului DirectoryRepository.

Spring Data se foloseste de aceste interfețe și de Reflection API, pentru a “scana” semnatura unei metode, prin intermediul căreia, va genera o propozitie SQL și va returna datele mapate la obiectul-entitate.

```
2022-06-09 01:23:32.742 DEBUG 34444 --- [nio-8080-exec-9] org.hibernate.SQL : select userentity0_.id as id1_3_, userentity0_.assigned_space as assigned2_3_, userentity0_.email as email3_3_, userentity0_.occupied_space as occupied4_3_, userentity0_.password as password5_3_, userentity0_.role as role6_3_ from user_entity userentity0_ where userentity0_.email=?  
2022-06-09 01:23:32.743 TRACE 34444 --- [nio-8080-exec-9] o.h.type.descriptor.sql.BasicBinder : binding parameter [1] as [VARCHAR] - [admin@admin.com]  
2022-06-09 01:23:32.747 INFO 34444 --- [nio-8080-exec-9] c.m.owncloudbackend.service.UserService : User FOUND in the dbadmin@admin.com  
2022-06-09 01:23:33.046 DEBUG 34444 --- [nio-8080-exec-4] org.hibernate.SQL : select userentity0_.id as id1_3_, userentity0_.assigned_space as assigned2_3_, userentity0_.email as email3_3_, userentity0_.occupied_space as occupied4_3_, userentity0_.password as password5_3_, userentity0_.role as role6_3_ from user_entity userentity0_ where userentity0_.email=?  
2022-06-09 01:23:33.047 TRACE 34444 --- [nio-8080-exec-4] o.h.type.descriptor.sql.BasicBinder : binding parameter [1] as [VARCHAR] - [admin@admin.com]  
2022-06-09 01:23:33.047 DEBUG 34444 --- [nio-8080-exec-4] org.hibernate.SQL : select userentity0_.id as id1_3_, userentity0_.assigned_space as assigned2_3_, userentity0_.email as email3_3_, userentity0_.occupied_space as occupied4_3_, userentity0_.password as password5_3_, userentity0_.role as role6_3_ from user_entity userentity0_ where userentity0_.email=?  
2022-06-09 01:23:33.049 TRACE 34444 --- [nio-8080-exec-8] o.h.type.descriptor.sql.BasicBinder : binding parameter [1] as [VARCHAR] - [admin@admin.com]  
2022-06-09 01:23:33.056 DEBUG 34444 --- [nio-8080-exec-8] org.hibernate.SQL : select directory0_.id as id1_0_, directory0_.name as name2_0_, directory0_.path as path3_0_, directory0_.user_id as user_id4_0_ from directory_entity directory0_ where directory0_.path=?  
2022-06-09 01:23:33.057 TRACE 34444 --- [nio-8080-exec-8] o.h.type.descriptor.sql.BasicBinder : binding parameter [1] as [VARCHAR] - [/home/marian/OwnCloud/admin@admin.com]
```

Figura 2.8 – Exemplu de propozitii SQL generate automat de Spring Data JPA.

2.5 Limbajul de programare TypeScript.

TypeScript este un limbaj de programare open-source, menținut de către Microsoft care funcționează ca un superset al limbajului JavaScript, oferindu-i un sistem optional de tipuri. Acesta este adesea utilizat în aplicații mai mari, unde este compilat în JavaScript iar mai apoi rulat în browser[5].

De asemenea, conform unui sondaj realizat de Stack Overflow în 2021, TypeScript se afla pe locul 3 la secțiunea de „Cele mai îndrăgite limbi de programare”[6].

```
class Person{  
    constructor(public firstName:string,  
               public lastName:string) {  
    }  
}  
  
const person:Person = new Person("John","Doe")  
const fullName:string = `${person.firstName}, ${person.lastName}`;  
console.log(`Hello ${fullName}!`) // "Hello John, Doe!"
```

Figura 2.9 – Exemplu de cod in TypeScript

JavaScript este un limbaj de programare, cu tipuri dinamice, care rulează în browser și are ca și scop principal introducerea de funcționalități într-o pagină web. De asemenea acesta poate controla DOM-ul (Document Object Model), pentru a realiza aplicații web interactive.

Pe lângă facilitățile pe care le oferă limbajul, acesta nu este puternic tipizat, ceea ce reprezintă un detriment în cazul aplicațiilor de dimensiuni mari, deoarece pot apărea foarte multe inadvertențe de tip.

Astfel, ca și alternativă pentru JavaScript, în 2012 a fost făcut pentru prima dată public limbajul TypeScript care avea posibilitatea atașării unui sistem de tipuri limbajului JavaScript.

Pe lângă sistemul de tipuri în sine, au fost adăugate și alte îmbunătățiri precum interfețe, clase (inexistente în JavaScript până la versiunea ECMAScript 2015), uniuni de tipuri, diversi alți operatori și.a.m.d.

Totodată, sistemul de tipuri este optional, deci se poate opta ca acesta să nu intervină în anumite cazuri și de regula se face prin specificarea tipului „any”, însemnând „orice”.

2.6 Framework-ul Angular

Angular este un Framework open-source, bazat pe TypeScript care este dezvoltat în principal de echipa Angular de la Google.[7]

Că și platformă, aceasta include : o arhitectură de dezvoltare bazată pe componente, o colecție de biblioteci integrate pentru diverse funcționalități precum comunicarea cu diverse API-uri, rutare, securitate și o suita de instrumente ce au scopul de a ușura dezvoltarea aplicațiilor frontend de ex, Angular CLI.[8]

```
@Component({
  selector: 'app-file-upload-dialog',
  templateUrl: './file-upload-dialog.component.html',
  styleUrls: ['./file-upload-dialog.component.scss']
})
export class FileUploadDialogComponent implements OnInit {

  allFiles: File[] = [];

  constructor(
    @Inject(MAT_DIALOG_DATA) private data: File[],
  ) {
  }

  ngOnInit(): void {
    this.allFiles = [...this.data];
  }
}
```

Figura 2.10 – Exemplu de cod TypeScript în Angular.

O parte fundamentală a acestui framework o reprezintă Angular CLI (Command line Interface). Aceasta reprezintă un instrument destinat creării și dezvoltării unui proiect în Angular care se folosește de o interfață în linia de comandă.

```
PS C:\Users\ZZ03FL826\Documents\delete\untitled1> ng generate component example/component
CREATE src/app/example/component/component.html (24 bytes)
CREATE src/app/example/component/component.component.spec.ts (647 bytes)
CREATE src/app/example/component/component.component.ts (287 bytes)
CREATE src/app/example/component/component.component.css (0 bytes)
UPDATE src/app/app.module.ts (416 bytes)
PS C:\Users\ZZ03FL826\Documents\delete\untitled1> [ ]
```

Figura 2.11 – Exemplu de comanda către Angular CLI folosindu-se directiva „ng” pentru crearea unui nou component

Adăugarea dependințelor suplimentare, fară o legătură directă cu Angular se poate face folosind managerul de pachete npm, dar în cazul unor biblioteci care nu au fost dezvoltate în TypeScript este adesea utilă și instalarea unor biblioteci ce conțin definiții de tip specifice acelei biblioteci.

2.7 Componente

Componentele reprezintă unitățile funcționale de bază ale unei aplicații în Angular, deoarece prin intermediul lor se generează interfața și se tine cont de starea aplicației.

```
@Component({
  selector: 'app-admin-dashboard',
  templateUrl: './admin-dashboard.component.html',
  styleUrls: ['./admin-dashboard.component.scss'],
  animations: [...],
})
export class AdminDashboardComponent implements OnInit {
  tableDataSource: UserInfo[] = [];
  displayedColumns: string[] = ['id', 'email', 'role', 'occupiedSpace', 'assignedSpace'];
  expandedElement: UserInfo | null | undefined = undefined;

  sysTotalSpace!: number;
  sysUsableSpace!: number;

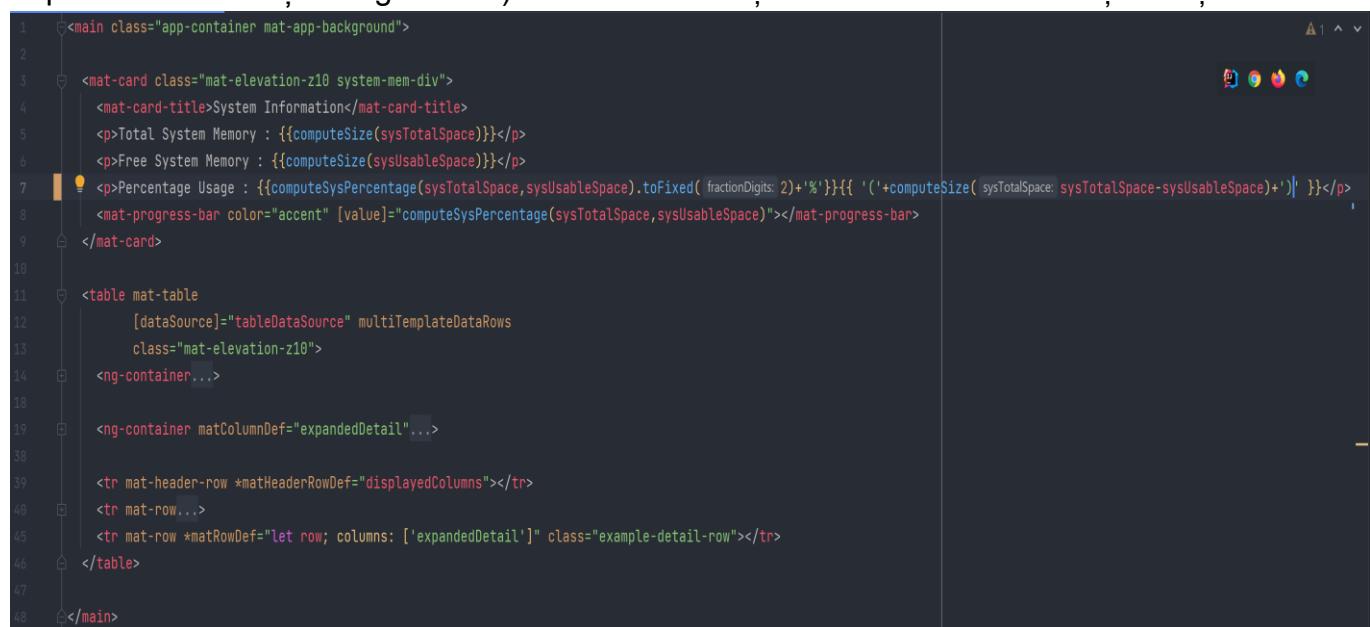
  constructor(private userService: AuthService,
    private dialog: MatDialog,
    private fileService: FileService
  ) {...}

  ngOnInit(): void {
    this.loadSysInfo();
    this.loadUserInfo();
  }
}
```

Figura 2.12 – Exemplu de componentă în Angular

Un component se face remarcat având deasupra definiției de clasa, decoratorul `@Component`. Acesta trebuie să conțină un parametru „selector” care reprezintă numele componentului și poate fi folosit ca și selector CSS, un parametru „templateURL” care reprezintă calea către şablonul HTML folosit pentru generarea interfeței grafice și un alt parametru „styleUrls” care reprezintă un sir de căi către resursele CSS pentru stilizarea

componentului. Bineînțeles, există și alți parmetri la nivelul decoratorului @Component (după cum se vede și în Figura 2.9) dar cei enumerați anterior sunt considerați esențiali.



```

1 <main class="app-container mat-app-background">
2
3   <mat-card class="mat-elevation-z10 system-mem-div">
4     <mat-card-title>System Information</mat-card-title>
5     <p>Total System Memory : {{computeSize(sysTotalSpace)}}</p>
6     <p>Free System Memory : {{computeSize(sysUsableSpace)}}</p>
7     <p>Percentage Usage : {{computeSysPercentage(sysTotalSpace,sysUsableSpace).toFixed( fractionDigits: 2)+'%'}}{{ ('+'+computeSize( sysTotalSpace: sysTotalSpace-sysUsableSpace)+')' }}</p>
8     <mat-progress-bar color="accent" [value]="computeSysPercentage(sysTotalSpace,sysUsableSpace)"></mat-progress-bar>
9   </mat-card>
10
11   <table mat-table
12     [dataSource]="tableDataSource" multiTemplateDataRows
13     class="mat-elevation-z10">
14     <ng-container>...</ng-container>
15
16     <ng-container matColumnDef="expandedDetail" ...>
17
18       <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
19       <tr mat-row ...>
20         <tr mat-row *matRowDef="let row; columns: ['expandedDetail']" class="example-detail-row"></tr>
21       </ng-container>
22     </table>
23
24   </main>

```

Figura 2.13 – Şablonul HTML asociat componentei din figurile anterioare

Şablonul HTML folosit în Angular reprezintă cod HTML pentru care s-au adus diverse îmbunătățiri pentru a spori viteza de dezvoltare. Un prim exemplu le reprezintă directivele condiționale și repetitive *ngIf și *ngFor.



```

<section class="directory-section mat-elevation-z10" *ngIf="!isLoading">
  <div class="directory-div">
    <p>...</p>
    <div matRipple class="dir-item" ...>

      <div class="dir-item" *ngFor="let dir of directories">
        <button mat-button (click)="onDirDeleteClick(dir)" class="dir-delete-button" ...>
        <button mat-button ...>
      </div>
    </div>
  </section>

```

Figura 2.14 – Exemplu de utilizare a directivelor *ngIf și *ngFor

Directiva *ngIf face în aşa fel încât, dacă condiția din definiția acesteia este adevărată, atunci elementul pe care se află va fi vizibil. Astfel putem ascunde/afisa detalii în funcție de o anumită stare a respectivului component.

Directiva *ngFor este una iterativă, ea generând dinamic taguri HTML la fel, pentru fiecare element din colecția la care se face referire în interiorul acesteia. În acest fel, se pot genera dinamic elemente cu aceeași structură dar cu date diferite în funcție de o anumită colecție definită.

Există și alte directive precum *ngSwitch, însă am considerat doar *ngIf și *ngFor relevante pentru a fi explicate și folosite în cadrul FileStorm.

Legăturile între şablonul HTML și componenta efectivă sunt de două tipuri : unidirectionale și bidirectionale.

Un exemplu de legătură unidirectională este operatorul „{{}}”, în interiorul căruia putem scrie date din interiorul componentului (stare), care mai apoi vor fi afișate și în interfață.

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
```

Figura 2.15 – Exemplu legătura bidirectională [9]

Un exemplu de legătură bidirectională este vizibil în Figura 2.12 și iese în evidență prin utilizarea operatorului „[()]”.

Acest tip de legare înseamnă că starea aplicației poate fi schimbată de către interfață, dar și interfața poate fi schimbată în momentul schimbării stării.

De regulă, aplicațiile în Angular au un component de bază, denumit AppComponent și este poziționat în tag-ul „body” al fișierului HTML auto-generat de către Angular CLI. În general, restul componentelor se aşază în interiorul acestui component.

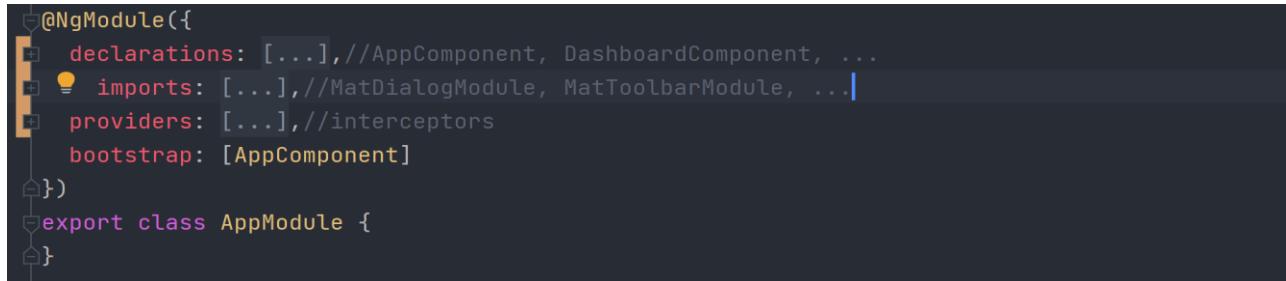
```
@import "../../assets/mixins/base-colors";  
  
.main {  
    margin-top: 40px;  
    width: 100%;  
}  
  
.container {  
    width: 100%;  
    height: 100%;  
    padding-top: 54px;  
  
    .infinite-container {  
        overflow: auto;  
    }  
  
    .sidenav {  
        padding-top: 54px;  
        max-width: 147px;  
    }  
}
```

Figura 2.16 – Exemplu de fișier SCSS din cadrul unui component

Un alt element important din cadrul unui component este fișierul CSS. În cadrul acestui fișier putem stiliza elementele din interiorul componentului.

Tipul de fișier CSS este ales la crearea proiectului cu Angular CLI, și există opțiunea de fișiere simple CSS, SCSS (Sassy CSS) și SASS. În cadrul platformei propuse FileStorm am ales să utilizez fișiere SCSS deoarece au o sintaxă mai plăcuta, permit imbricarea și alte noutăți față de CSS cum ar fi definirea de variabile. Fișierul CSS atașat unui anumit component are efect numai asupra acestuia, deci nu se pot accesa prin diversi selectori, elementele altui component.

Faptul că există această structurare între componenta vizuală, cea funcțională și şablon reprezintă unul dintre caracteristicile definitorii ale Framework-ului Angular și ajuta enorm în dezvoltarea aplicațiilor și ulterior la scalarea acestora.



```
@NgModule({
  declarations: [...], // AppComponent, DashboardComponent, ...
  imports: [...], // MatDialogModule, MatToolbarModule, ...
  providers: [...], // interceptors
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Figura 2.17 – Exemplu de modul Angular

Ar mai fi de menționat, ca și funcționalitate utilă la nivelul framework-ului, existența modulelor. Acestea au rolul de a defini ce alte module sunt necesare la nivelul unor anumite componente pentru a nu încărca toate celelalte module dacă nu sunt necesare.

2.8 Servicii



```
const CREATE_DIRECTORY = environment.baseUrl + '/dir/create';
const DELETE_DIRECTORY = environment.baseUrl + '/dir/delete';
const ALL_DIRECTORIES = environment.baseUrl + '/dir/getAll';

@Injectable({
  providedIn: 'root'
})
export class DirectoryService {

  constructor(private http: HttpClient) { }

  createDirectory(dirName:string,pathsFromRoot:string[]){...}

  getAllDirectories(pathsFromRoot:string[]){...}

  deleteDirectory(id: number) {...}
}
```

Figura 2.18 – Exemplu de Serviciu în Angular

Angular, similar cu Framework-ul Spring, are la bază, pe lângă arhitectura bazată pe componente și injectarea dependințelor. Astfel, există posibilitatea injectării unei clase în interiorul altrei clase în aşa fel încât, programatorul nu mai este responsabil de crearea și managementul instanțelor respective.

Definirea unui serviciu se face cu ajutorul decoratorului `@Injectable` deasupra definiției de clasă. Acest decorator are și o proprietate: „`providedIn`” care oferă posibilitatea de a alege ce injector să se folosească la nivelul serviciului.

```
| constructor(private auth: AuthService) {  
| }  
|
```

Figura 2.19 – Exemplu de injectare a unui serviciu

După cum se poate observa și din figura de mai sus, serviciile se pot injecta în constructorul altor clase, astfel putem răspândi funcționalitatea respectivului serviciu în alte servicii/componente pentru o experiență de dezvoltare a aplicațiilor mult simplificată.

2.9 Angular Guards

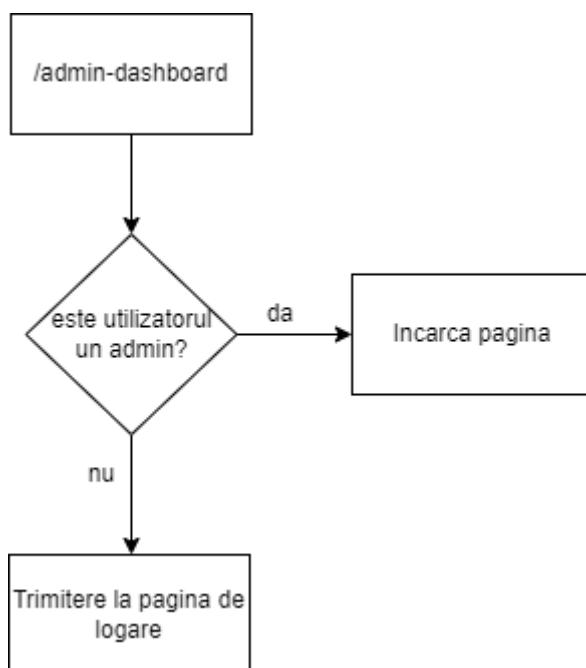


Figura 2.20 – Diagrama de activitate a unui Angular Guard pentru pagina aferentă unui utilizator cu rol de administrator

Un Angular Guard este un tip de serviciu care are ca și rol principal restricționarea accesului unui utilizator la o anumită pagină. Aceștia se diferențiază față de celelalte servicii datorită faptului că implementează interfața CanActivate.

Funcțiile acestui tip de serviciu sunt apelate în momentul în care un utilizator dorește să acceseze o resursă protejată. Scopul Guard-ului este de a verifica informațiile necesare ale utilizatorului, pentru a permite sau a restricționa accesul.

2.10 RxJs

Reactive Extensions for JavaScript reprezintă o biblioteca bazată pe programarea reactivă ce se folosește de Observables pentru a ușura folosirea operațiilor asincrone.[10]

Programarea reactivă este o paradigmă de programare care se bazează pe fluxuri de date și pe propagarea schimbărilor în cadrul acestuia și este folosită cu predilecție în

aplicațiile cu interfață grafică ce interacționează cu un utilizator și trebuie să managerizeze diverse acțiuni într-o manieră asincronă.

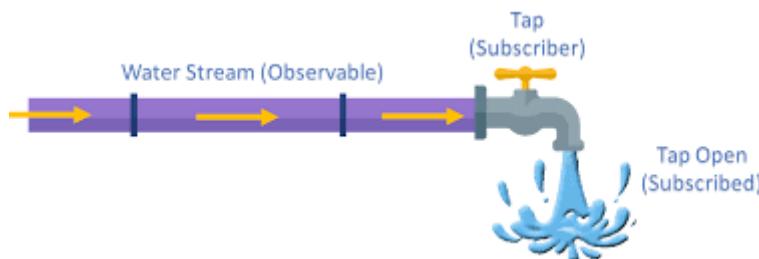


Figura 2.21 – Descrierea funcționalității unui flux de date folosind ca și exemplu un robinet.[11]

Un flux de date se poate reprezenta teoretic asemenea unui flux de apă printr-o conductă. Fluxul în sine se numește Observable iar la capătul acestuia există un robinet supranumit Subscriber. Când un Subscriber este atașat fluxului, acesta curge dând voie informațiilor să se propage mai departe. La nivelul unui flux reactiv de date, se mai pot adăuga diverse filtre pentru a permite doar anumite date să treacă, se pot realiza operații înainte ca un Subscriber se fie atașat, ba chiar se poate schimba cu totul conținutul fluxului sau atașarea unui alt flux.

În cadrul Angular, RxJS este utilizat în efectuarea tuturor operațiilor asincrone și pune la dispoziție diverse obiecte pentru a simplifica lucrul cu acestea.

În aplicația propusă FileStorm, RxJS este cel mai des utilizat pentru a procesa fluxul de date venite de la server într-o manieră asincronă pentru a putea salva datele venite în starea aplicației și a le afișa mai apoi în interfață.

```
onRegister() {
  this.isLoading = true;
  this.auth.onRegister(this.email.value, this.password.value)
    .subscribe( observer: {
      next: (response :{email: string, password: string}) => {
        console.log(response);
        this.isLoading = false;
        this.loginAfterRegister(response);
        this.dialogRef.close();
      },
      error: err => {...}
    });
}
```

Figura 2.22 – Exemplu de procesare a răspunsului primit în urma unui request HTTP la server.

2.11 Angular Material

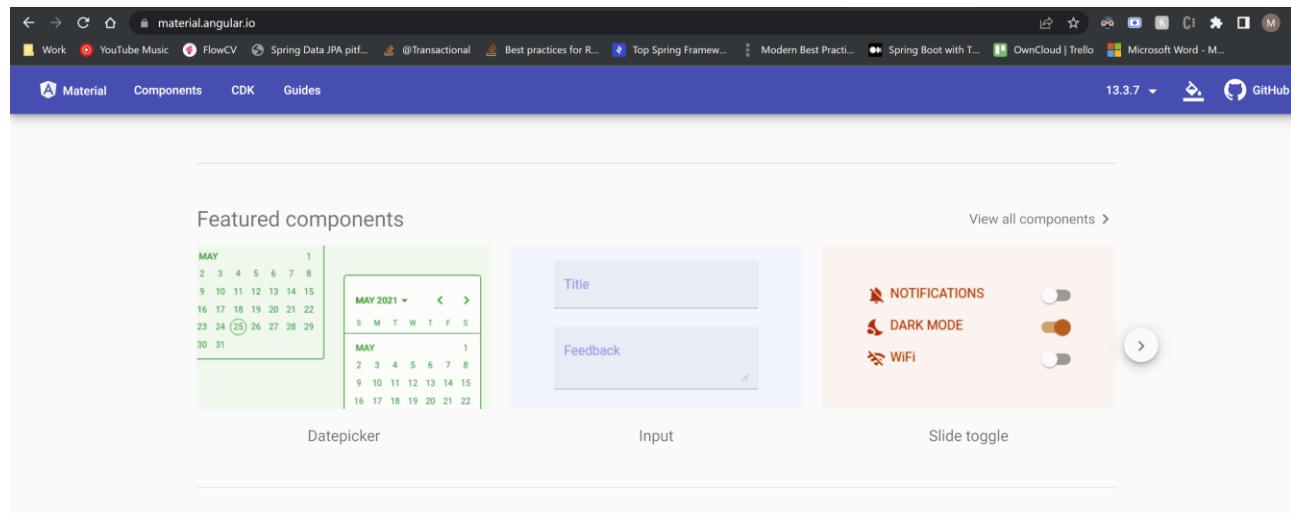


Figura 2.23 – Site-ul oficial Angular Material, arătând câteva din componente de bază.

Angular Material reprezintă o bibliotecă de componente, instalată separat față de dependințele de bază ale framework-ului Angular ce conține elemente stilizate conform standardului Material întreținut de Google.

Această bibliotecă oferă diverse componente reutilizabile și cu un design modern, care oferă consistență interfeței de utilizator. Printre acestea se enumeră : carduri, butoane, casete de text, tabele, iconițe și multe altele. Ele au fost create în aşa fel încât să funcționeze armonios împreună și să ofere aplicației un design deosebit dar oferă totuși și accesibilitate.

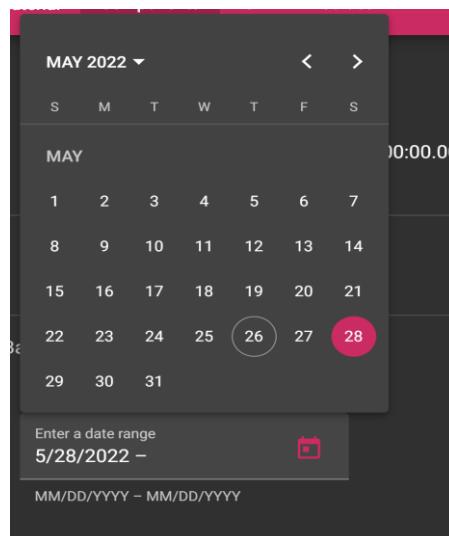


Figura 2.24 – Exemplu de selector de date din biblioteca Angular Material[12]

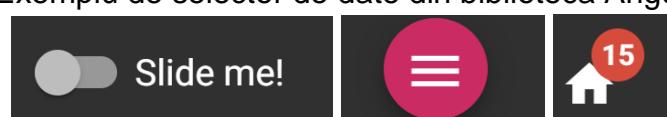


Figura 2.25 – Exemplu de slider, buton și iconiță cu indicator[12]

Pentru introducerea bibliotecii Angular Material într-un proiect angular, este nevoie de Angular CLI, prin care putem rula comanda „ng add @angular/material”.

În urma rulării comenzi se va cere efectuarea unor alegeri pentru alte funcționalități pe care biblioteca le aduce precum fontul specific sau o temă de culori.

Totodată, aceste elemente sunt ușor stilizabile și oferă și câteva animații pentru a se îmbunătății experiența oferită utilizatorului.

2.12 SCSS

Sass(Syntactically Awesome Style Sheets) este un limbaj de script preprocesor care este interpretat sau compilat în fișiere CSS. Acesta este format din două sintaxe SASS și SCSS[13].

SCSS este un tip de sintaxă pentru Sass care sporește deosebită de cealaltă sintaxă care este bazată pe indentare, folosește accolade pentru definirea domeniului unui selector anume.



```
1  @import "src/assets/mixins/base-colors";  
2  
3  .main {  
4      height: 100%;  
5      display: flex;  
6      justify-content: center;  
7      align-items: center;  
8      mat-card {  
9          min-width: 300px;  
10         form {...}  
11         .footer-content{...}  
12     }  
13     }  
14 }  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26 }
```

Figura 2.26 – Exemplu de cod SCSS.

Acest mod de a scrie CSS vine cu câteva îmbunătățiri majore precum definirea variabilelor, definirea portiunilor de stiluri similare denumite „mixins” care pot fi incluse în altele, imbricarea selectorilor și multe altele.

```
$primary: mat.get-color-from-palette($angular-primary,500);  
$accent: mat.get-color-from-palette($angular-accent,A200);  
$warn : mat.get-color-from-palette($angular-warn,A200);
```

Figura 2.27 – Exemplu de definire de variabile in SCSS.

Prin intermediul variabilelor, putem memora diverse date, precum culori, lucru care este foarte util având în vedere că în general acestea se exprimă în hexazecimal sau alte

formate greu de reținut. Ulterior ele pot fi incluse în alte fișiere SCSS pentru o experiență de dezvoltare mai bună și o structurare mai reușită a stilurilor.

2.13 Progressive Web Application

Aplicațiile web progresive sunt o tehnologie, apărută în 2015, care permit dezvoltarea aplicațiilor asemănătoare celor native, dar folosind limbaje de dezvoltare web precum JavaScript, HTML și CSS.[14]

Această tehnologie conferă unei aplicații web un aspect similar cu o aplicație dezvoltată pe o platformă nativă fară să nevoie să se creeze o aplicație nouă, lucru ce poate scădea drastic costurile de producție.

Aplicațiile de tip PWA sunt instalabile din browser, iar instalarea acestora durează în general foarte puțin, cu mult mai puțin decât o aplicație nativă.

Asemănător cu unele aplicații clasice, dar invers fata de aplicațiile web, PWA-urile oferă posibilitatea de a trimite notificări push, dar și o parte de funcționalități offline.

Spre deosebire de aplicațiile native, acestea nu au nevoie de un anumit magazin ca să fie descărcate și instalate ulterior, ci ele se pot instala în urma aprobării unui mic pop-up la intrarea pe platforma web.

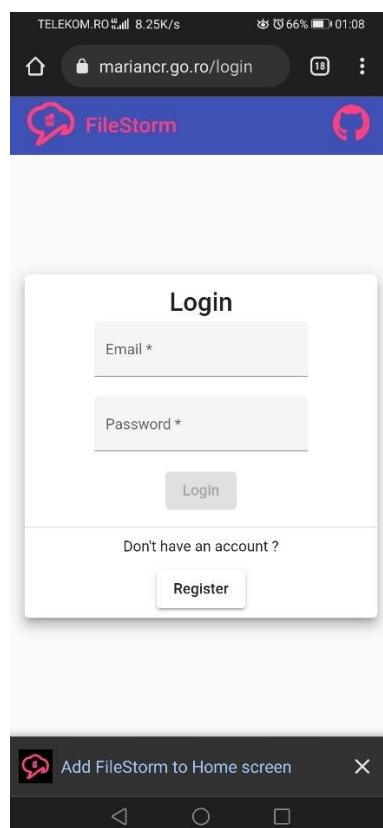


Figura 2.28 – Exemplu de pop-up pentru instalarea aplicației FileStorm.

2.14 Comunicarea între aplicații folosind HTTP

Hypertext Transfer Protocol este metoda cea mai des utilizată pentru accesarea resurselor pe Internet care sunt salvate pe servere WWW (World Wide Web). Acesta este un protocol de tip text și este protocolul implicit al WWW [15].

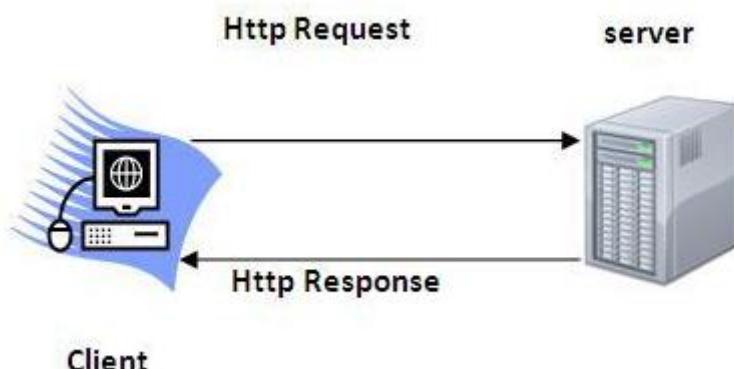


Figura 2.29 – Diagrama protocolului HTTP[16]

Protocolul HTTP reprezintă o modalitate de comunicare între un client și server, bazat pe TCP pentru stabilirea conexiunii între cele două dispozitive, pentru clasificarea și structurarea tipurilor de cereri, diverse metode. Printre metodele HTTP cele mai importante se enumerează : GET – utilizat în accesarea datelor de la server, POST – utilizat în transmiterea de date către server, PUT – utilizat în transmiterea și/sau modificarea datelor de pe server și DELETE – utilizat în ștergerea anumitor resurse la nivelul serverului.

Acest protocol este utilizat în aplicație ca și mediu de comunicare într-un Web API (Application Programming Interface). Acest Web API pune la dispozitiv diverse endpoint-uri utilizate de către aplicația de frontend.

2.15 Comunicarea folosind Websockets

Websocket este un protocol de comunicații, care oferă canale de comunicare full-duplex printr-o singură conexiune TCP[17].

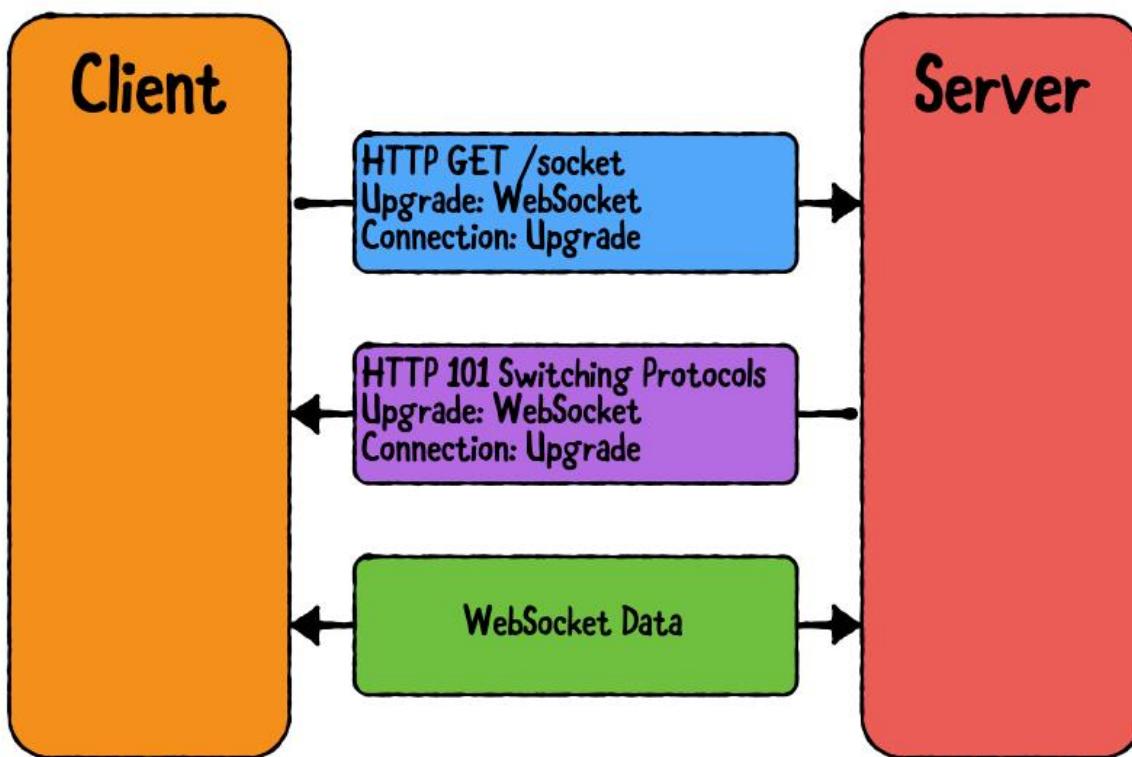


Figura 2.30 – Diagrama protocolului Websocket[18]

Acest protocol este distinct față de HTTP, deoarece este o conexiune full-duplex care facilitează transferul de date în timp real către și dinspre server. Spre deosebire de HTTP polling, care este o tehnică ce presupune trimiterea de request-uri HTTP către server la un anumit interval de timp pentru a comunica date ce trebuie actualizate în timp real, Websocket-urile sunt mult mai utile în facilitarea acestui proces. Datorită existenței unui standard bine definit de a transmite date către un client, fără să fie necesară în prealabil o cerere, se poate permite astfel o comunicare în ambele direcții, atât timp cât conexiunea nu este opriță.

Transferul mesajelor se face pe anumite canale, numite adesea „topic-uri” ce permit segregarea tipurilor de informații transmise prin canalul TCP.

Ca și proces de comunicare, acest standard este explicitat în Figura 2.27, unde se observă că inițial se realizează un request HTTP specific, adesea numit „handshake”, care are rolul de a stabili legătura dintre client și server. Mai apoi este deschis un canal de comunicare TCP, care permite acest tip de comunicare full-duplex.

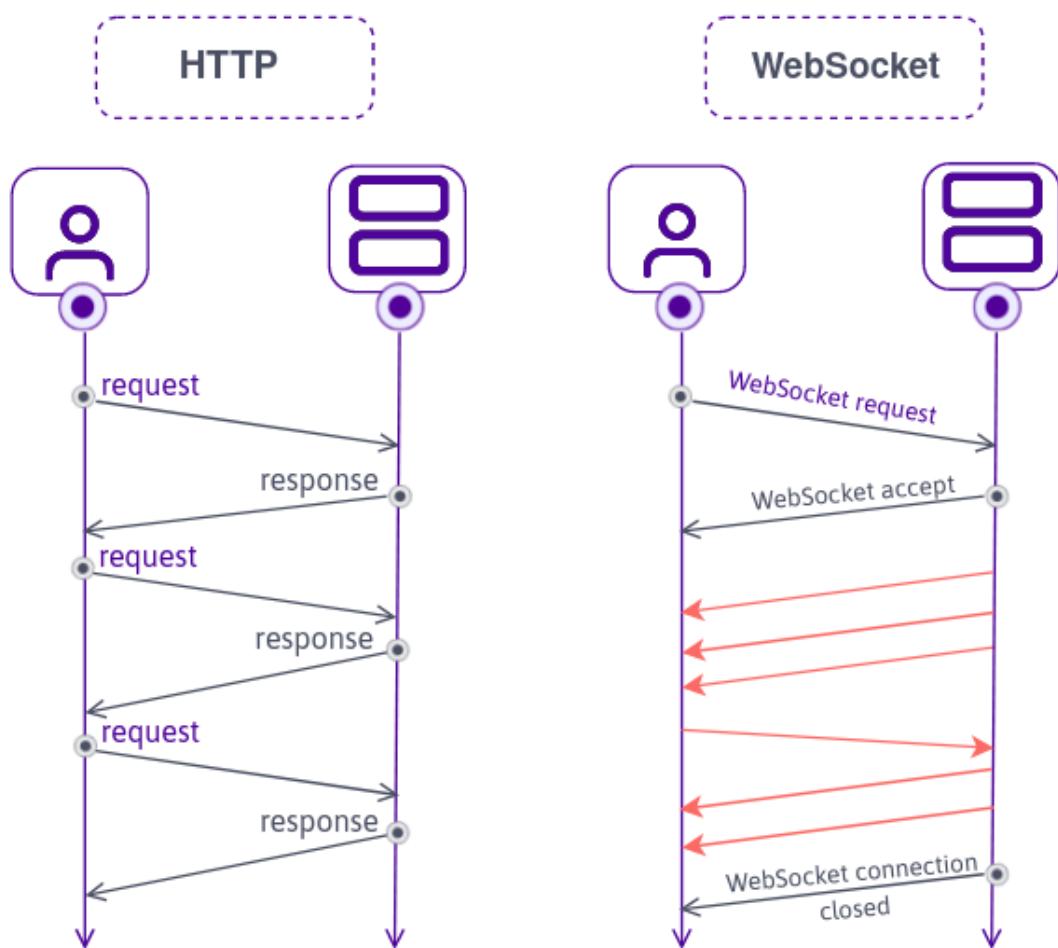


Figura 2.31 – Diagrama din care reiese diferența dintre protocolul HTTP și Websocket[19]

2.16 Git și GitHub

Git este un sistem de versionare multi-platformă, ce aduce multiple beneficii la nivelul unui proiect, de exemplu : un istoric al modificărilor, posibilitatea de a se întoarce la un anumit moment în cadrul proiectului, posibilitatea de a crea diverse alte căi cu propriul lor istoric denumite „branch-uri” și multe altele.

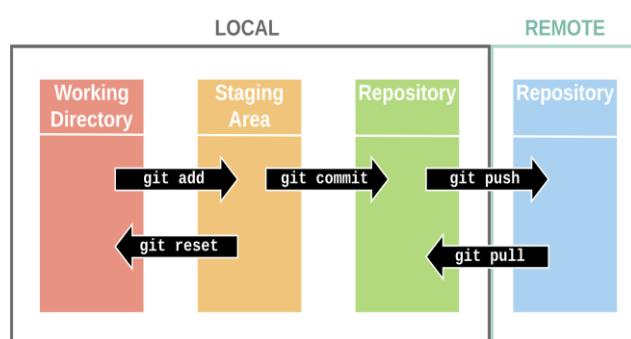


Figura 2.32 – Exemplu de funcționare a sistemului de versionare Git[20]

Din punct de vedere al funcționării, Git are o zona locală prezentă pe mașina pe care se află proiectul, și una remote sau „la distanță” care este de regulă disponibilă pentru mai mulți utilizatori și reprezintă zona unde se concentrează toate funcționalitățile unui anumit proiect.

Aceste zone remote, pot fi diversi clienți precum GitHub, GitLab, BitBucket, etc. Dintre acestea, aplicația FileStorm a fost dezvoltată utilizând GitHub.

GitHub este un serviciu de găzduire pentru proiecte dezvoltate folosind ca sistem de versionare Git. Aceasta este un serviciu gratuit pentru aplicații open-source și a fost achiziționat în 2018 de către Microsoft.

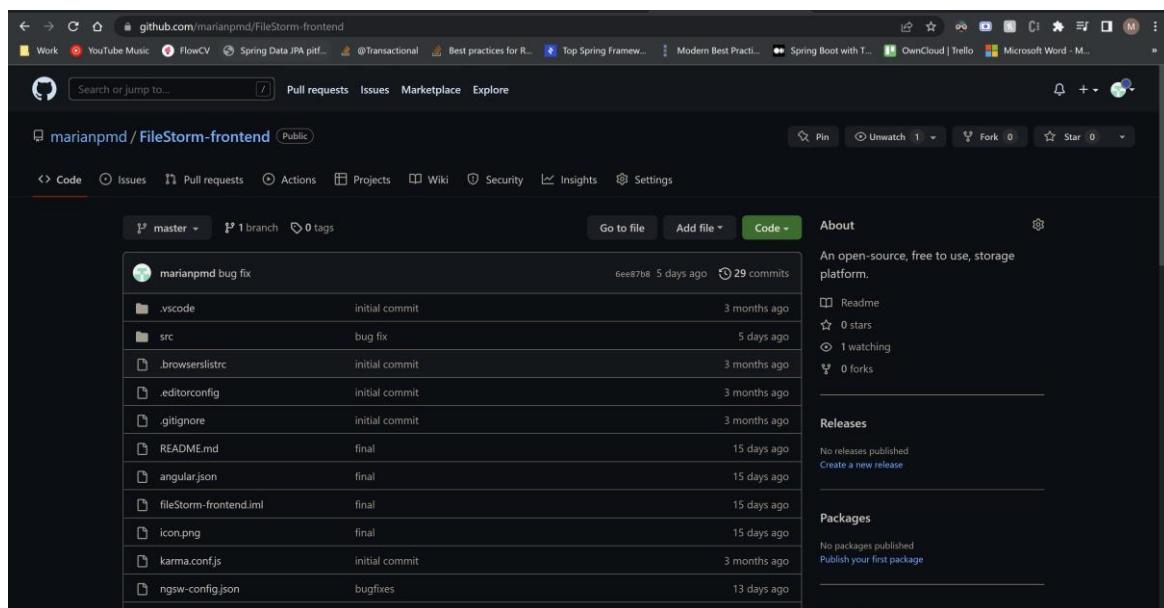


Figura 2.33 – Pagina de GitHub a proiectului, mai exact aplicația de frontend

2.17 Găzduirea aplicației pe server

FileStorm poate fi folosit fie într-o rețea internă, fie poate fi făcut accesibil pentru oricine pe Internet. Acest fapt depinde de felul în care este configurat serverul și este realizabil cu minime schimbări de configurație în cadrul proiectului.

Pentru găzduirea proiectului am folosit un dispozitiv de tip mini PC Lenovo ThinkCentre M700, protocolul ssh, domeniul mariancr.go.ro, și un server nginx.

SSH sau „Secure Shell” este un protocol de rețea criptografic ce permite ca datele să fie transferate utilizând un canal securizat.[21] Pe dispozitivul gazdă este necesară instalarea acestui protocol, iar pe orice alt dispozitiv de pe care dorim accesul, este nevoie de un client SSH. În cadrul acestui proiect am utilizat clientul Solar-PuTTY.

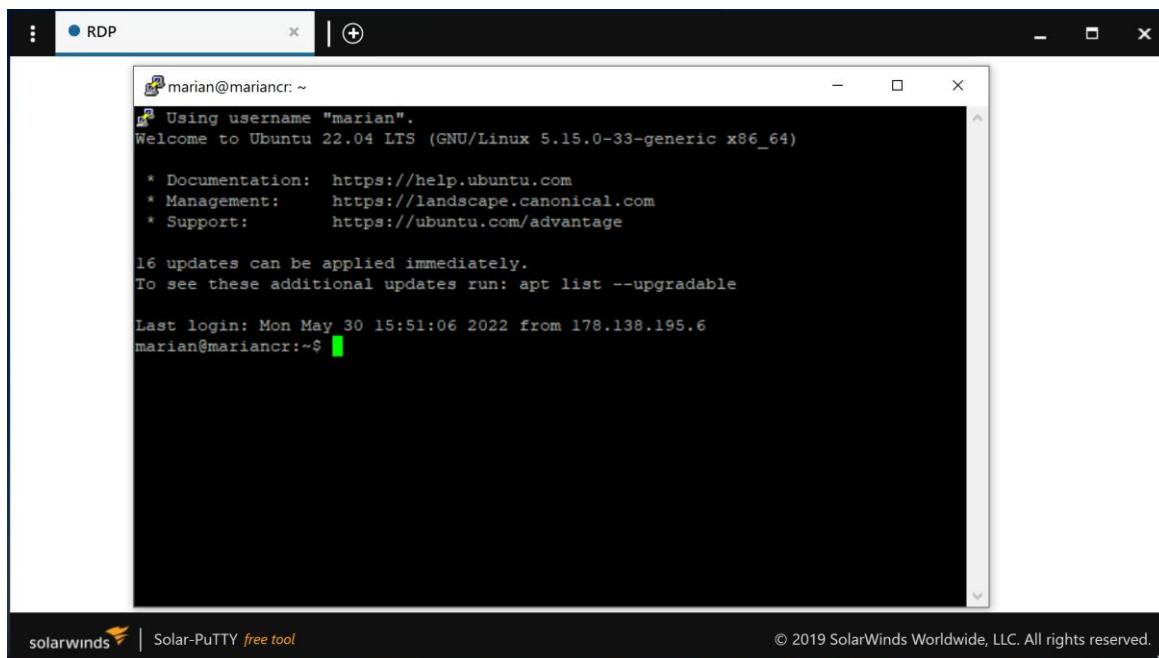


Figura 2.34 – Captura de ecran cu shell-ul deschis în urma conectării prin SSH la dispozitivul gazdă

Dispozitivul gazdă este accesibil prin SSH, prin adresa mariancr.go.ro la portul 22.

Nginx este un server web, open-source utilizat pentru a găzdui fișierele generate de aplicația de Angular, făcând astfel posibil accesul tuturor conexiunilor la aplicația de frontend.

Pentru backend, este pur și simplu suficient să pornim aplicația Spring pe portul potrivit, iar serverul Tomcat integrat se va ocupa de procesarea request-urilor.

```
nginx.service - A high performance web server and a reverse proxy server
   Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2022-05-30 12:16:52 EEST; 1h ago
     Docs: man:nginx(8)
 Process: 912 ExecStartPre=/usr/sbin/nginx -t -q -g daemon on; master_process on; (code=exited, status=0/SUCCESS)
 Process: 987 ExecStart=/usr/sbin/nginx -g daemon on; master_process on; (code=exited, status=0/SUCCESS)
 Main PID: 988 (nginx)
    Tasks: 5 (limit: 9329)
   Memory: 10.3M
      CPU: 346ms
     CGroup: /system.slice/nginx.service
             ├─988 "nginx: master process /usr/sbin/nginx -g daemon on; master_process on;"
             ├─989 "nginx: worker process" " "
             ├─990 "nginx: worker process" " "
             ├─991 "nginx: worker process" " "
             └─992 "nginx: worker process" " "

mai 30 12:16:47 mariancr.go.ro systemd[1]: Starting A high performance web server and a reverse proxy server...
mai 30 12:16:52 mariancr.go.ro nginx[912]: nginx: [warn] conflicting server name "mariancr.go.ro" on 0.0.0.0:80
mai 30 12:16:52 mariancr.go.ro nginx[912]: nginx: [warn] conflicting server name "mariancr.go.ro" on [::]:80, >
mai 30 12:16:52 mariancr.go.ro nginx[987]: nginx: [warn] conflicting server name "mariancr.go.ro" on 0.0.0.0:80>
mai 30 12:16:52 mariancr.go.ro nginx[987]: nginx: [warn] conflicting server name "mariancr.go.ro" on [::]:80, >
mai 30 12:16:52 mariancr.go.ro systemd[1]: Started A high performance web server and a reverse proxy server.
-
-
-
lines 1-23/23 (END)
```

Figura 2.35 – Rezultatul rulării comenzi „systemctl status nginx” pentru observarea stării serviciului.

3. PROIECTAREA TEHNICĂ A APLICAȚIEI

3.1 Diagrame UseCase

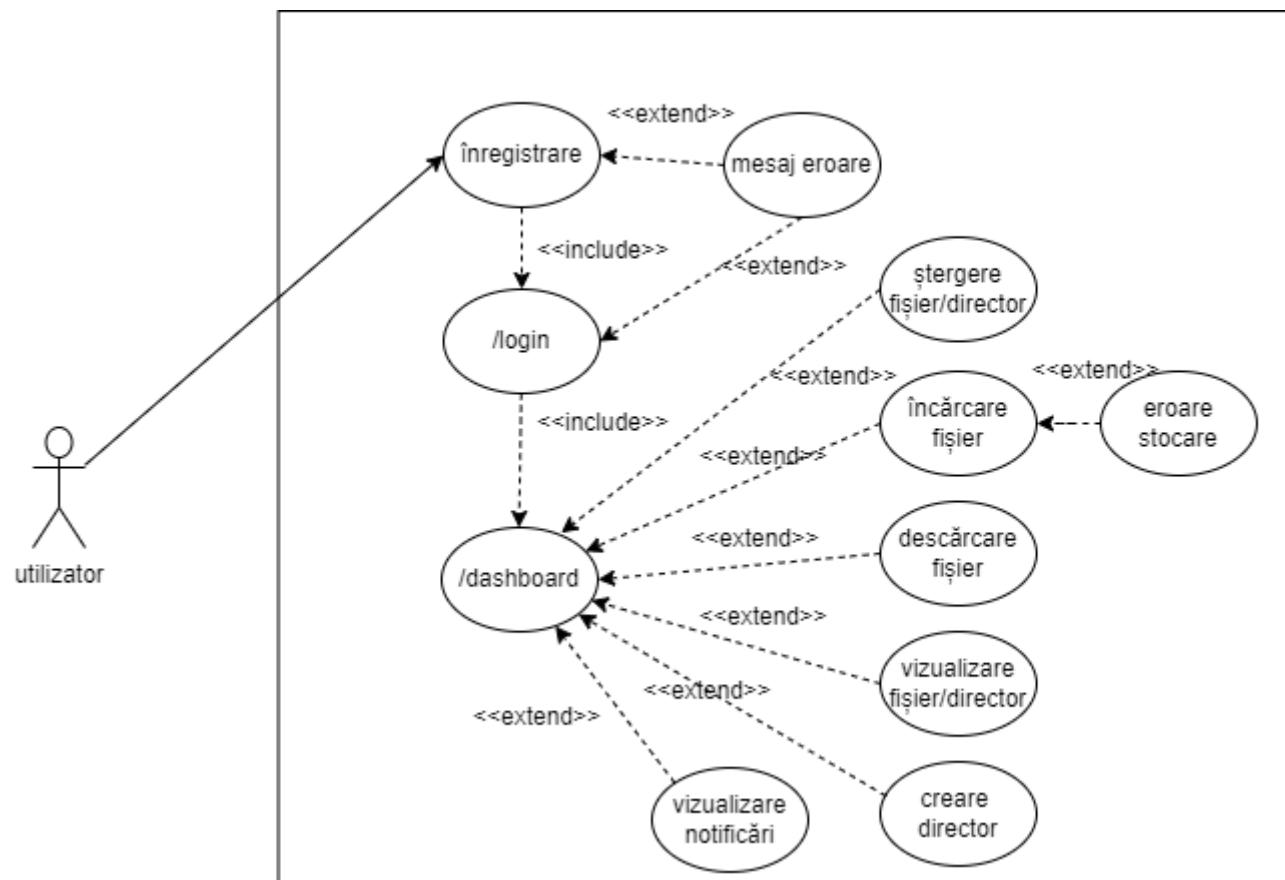


Figura 3.1 – Diagrama UseCase pentru un utilizator simplu

Această diagramă are rolul de a prezenta procesul prin care un utilizator obișnuit percurge aplicația. De la procesul de înregistrare, la procesul de logare și redirectionarea către interfață de baza denumita "dashboard" sunt procese mandatorii, iar toate celelalte procese la nivel de simplu utilizator precum vizualizarea fișierelor și a directoarelor, ștergerea sau crearea acestora sunt la latitudinea utilizatorului. De asemenea aceste procese pot avea asociare mesaje de eroare în cazuri precum : utilizatorul există deja, credentialele sunt greșite, utilizatorul nu are sufficient spațiu de stocare ș.a.m.d.

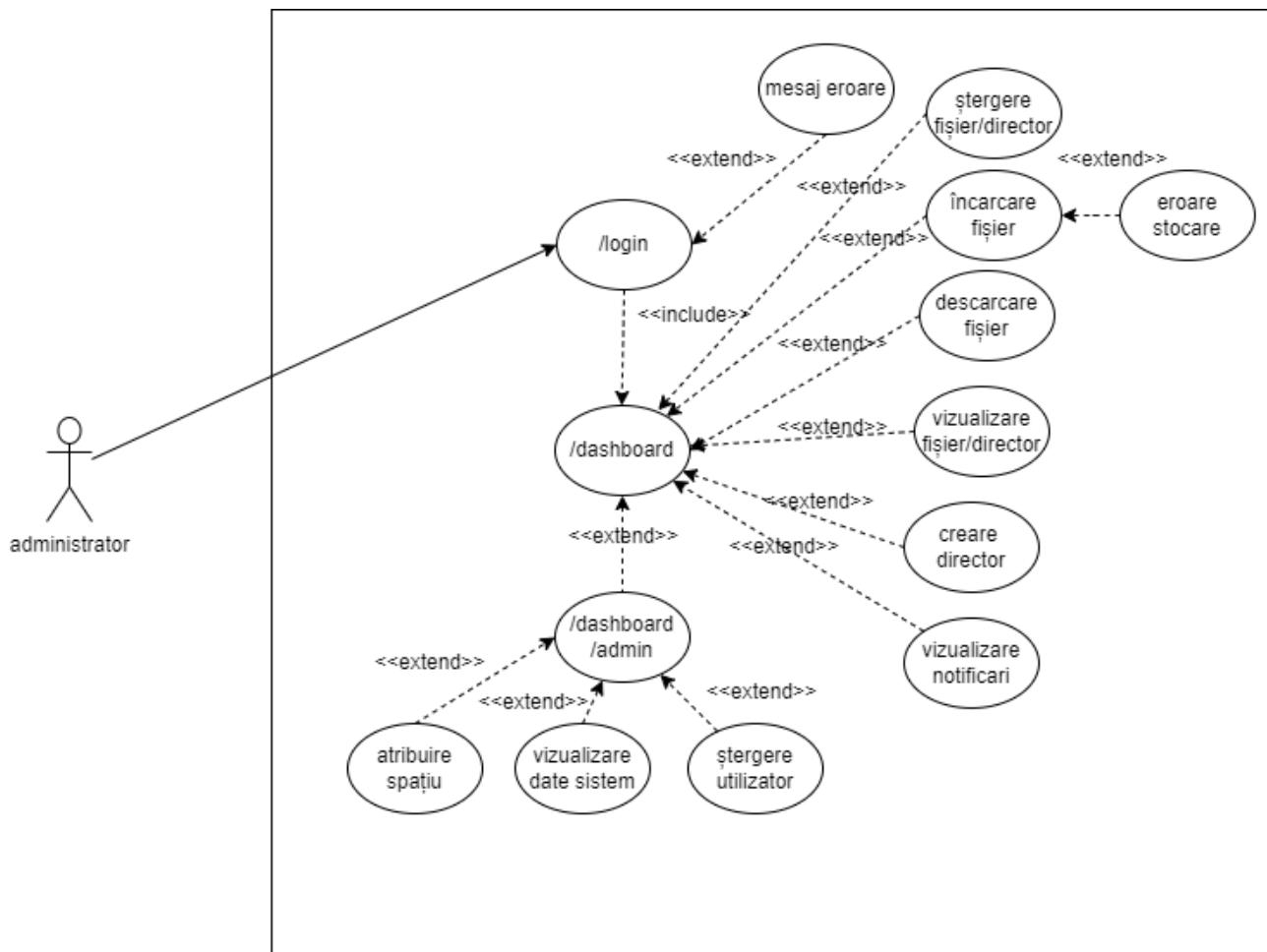


Figura 3.2 – Diagrama UseCase pentru un administrator

Această diagramă prezintă funcționalitățile pe care un administrator al platformei FileStorm le are. Se observă că acesta poate funcționa ca un utilizator obișnuit, doar că are câteva funcții în plus. Aceste funcții sunt : atribuirea de spațiu, vizualizarea datelor de sistem (se includ aici datele despre ce cantitate de date este încărcate de fiecare ușer în parte) și ștergerea utilizatorilor.

3.2 Arhitectura generală

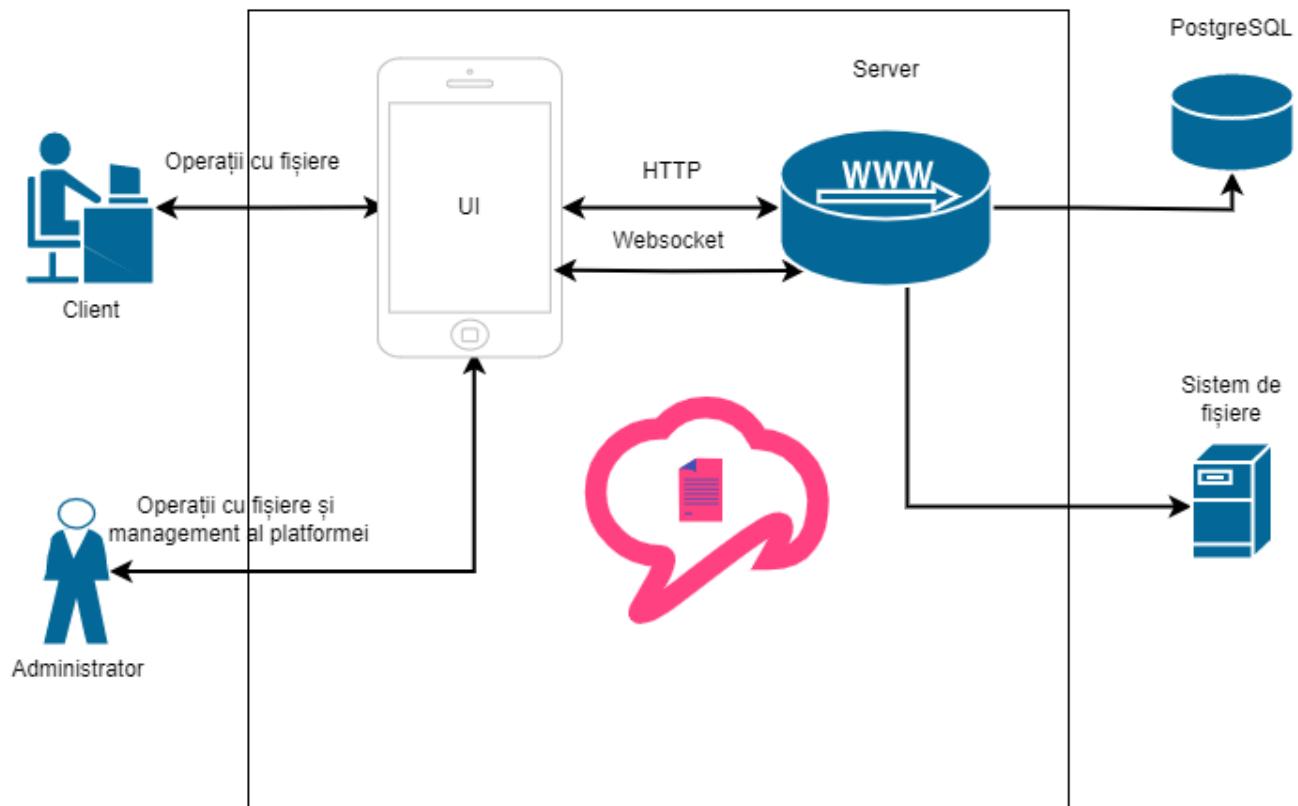


Figura 3.3 – Diagrama întregii arhitecturi FileStorm

Arhitectura aplicației FileStorm, privită din ansamblu este simplu de urmărit, având totuși un grad de complexitate relativ ridicat datorită diversității tehnologiilor utilizate și a capacitațiilor sistemului.

Aplicația presupune existența a două tipuri de utilizatori cu atribuții diferite, un utilizator obișnuit ce poate vizualiza, stoca, șterge și modifica fișiere aflate pe sistem și un administrator, care pe lângă atribuțiile de baza ale unui utilizator simplu, se mai ocupă și de management-ul platformei.

Partea de UI (User Interface), reprezintă interfața grafică a aplicației, care are un design modern și este disponibilă pe orice fel de dispozitiv.

Partea de server funcționează ca și un intermediar între sistem, baza de date și client (UI în acest caz), având rolul de a interoga baza de date că mai apoi să proceseze datele și să transmită fișierele aflate pe sistem către utilizator.

Baza de date utilizată este PostgreSQL, aceasta fiind o baza de date relațională și foarte performantă.

3.3 Arhitectura server-ului

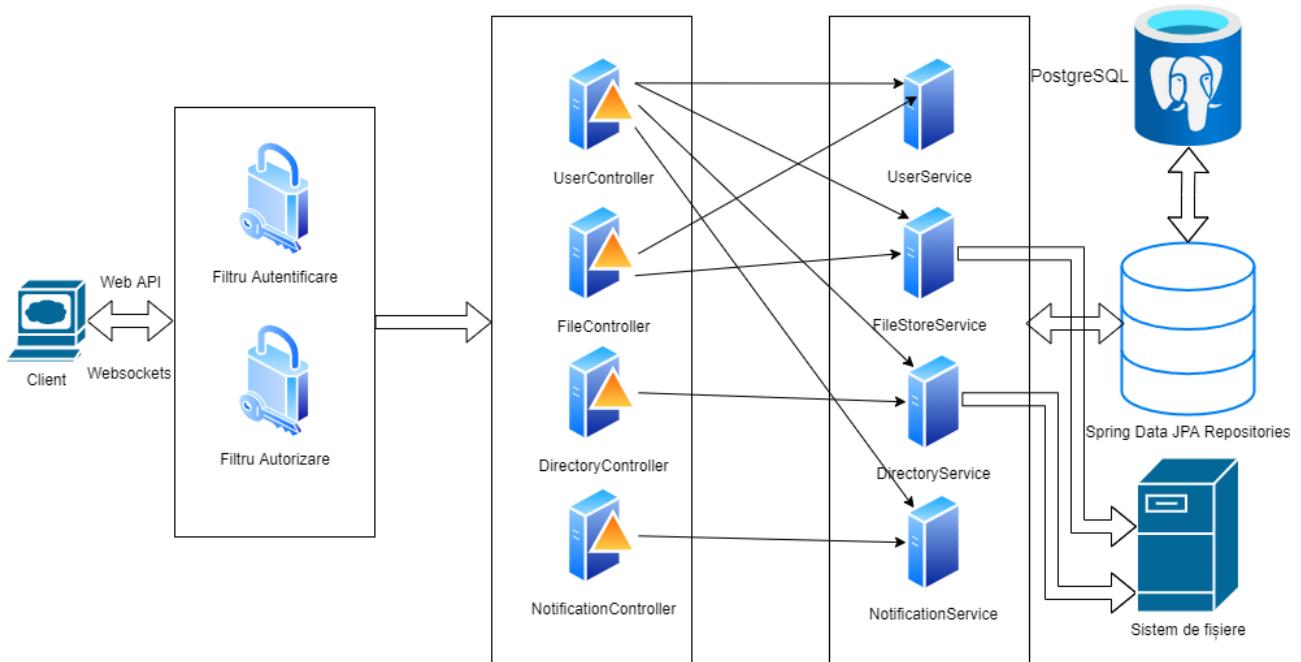


Figura 3.3.1 – Diagrama arhitecturii pe partea de server.

Arhitectura pe partea de server este destul de complexă, dar bine structurată și optimizată. Este reprezentată prin 5 straturi: Client, filtre, controlere, servicii și repositories (depozite de date din Spring).

Clientul, în acest caz reprezintă aplicația în Angular, care prin intermediul Web API-ului și al Websockets comunica cu serverul.

Această comunicare trebuie să fie securizată, astfel cel de-al doilea strat este cel cu filtrele de securitate. Filtrul de autentificare are rolul de a procesa cererile de înregistrare și logare iar cel de autorizare are rolul de a verifica toate cererile pentru autorizare.

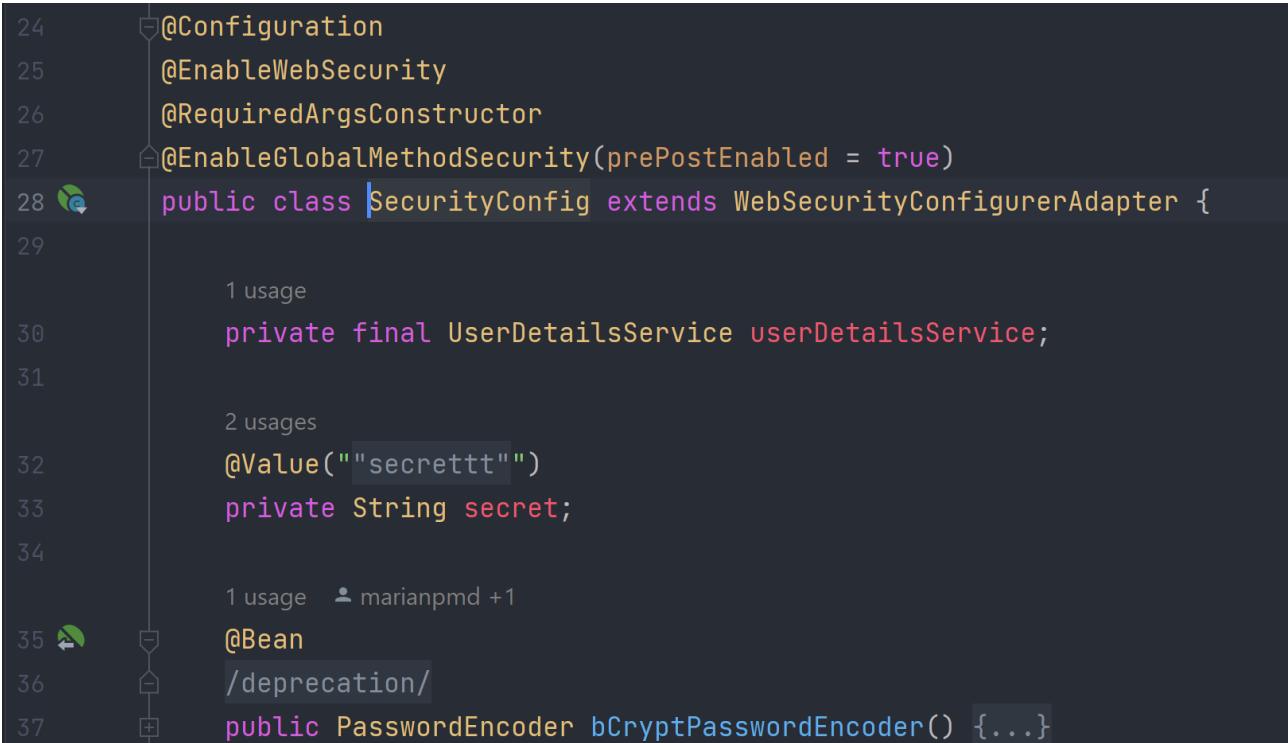
Următorul strat este cel de controller, el având aşa numite „endpoint-uri” la care sunt direcționate cererile, care mai apoi sunt procesate de servicii, mapate la un format convenabil (JSON) și returnează un răspuns.

Stratul cu servicii se ocupă de logica sistemului și are rolul de a prelua datele de la repository sau sistemul de fișiere și a efectua procesele necesare.

Repository-ul este stratul care se ocupă cu comunicarea efectivă cu baza de date, folosindu-se mecanisme de Object Relational Mapping și Query Methods.

3.3.1 Securitatea în FileStorm

Spring Security este o dependență separată de Spring Boot, ce aduce cu ea diverse configurații pentru securizarea unei aplicații web.



```

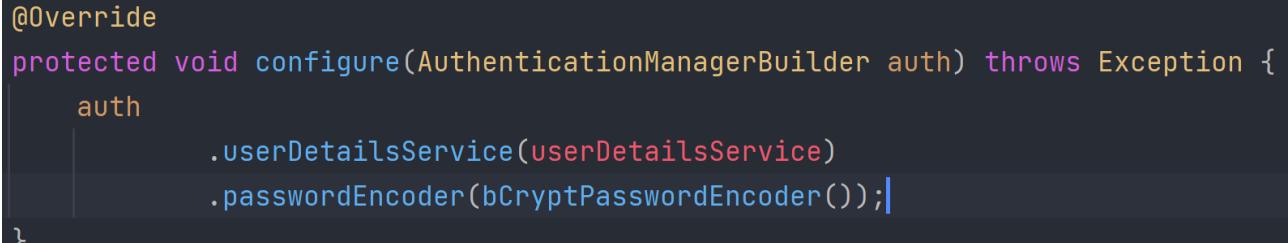
24     @Configuration
25     @EnableWebSecurity
26     @RequiredArgsConstructor
27     @EnableGlobalMethodSecurity(prePostEnabled = true)
28     public class SecurityConfig extends WebSecurityConfigurerAdapter {
29
30         1 usage
31         private final UserDetailsService userDetailsService;
32
33         2 usages
34         @Value("secrettt")
35         private String secret;
36
37         1 usage  ↳ marianpmd +1
38         @Bean
39         /deprecation/
40         public PasswordEncoder bCryptPasswordEncoder() {...}

```

Figura 3.3.2 – Captura de ecran cu clasa SecurityConfig.

Principala structură esențială pentru activarea securității în Spring Boot, este o clasă de configurație, cum se observă în figura de mai sus. Această clasa are rolul de a seta felul în care este gestionată securitatea, a înregistră filtre pentru diferite cereri și multe altele.

De asemenea tot în cadrul acestei clase se declară utilizarea unui serviciu, specializat în interogarea bazei de date, sau a altrei structuri de stocare a credentialelor (ex. LDAP), ce are rolul de a furniza datele necesare pentru verificarea automată a credentialelor.



```

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth
        .userDetailsService(userDetailsService)
        .passwordEncoder(bCryptPasswordEncoder());
}

```

Figura 3.3.3 – Metoda responsabilă pentru declararea serviciului specializat.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    http.cors();

    var customAuthenticationFilter = new CustomAuthenticationFilter(authenticationManagerBean());
    var customAuthorisationFilter = new CustomAuthorisationFilter();

    customAuthorisationFilter.setSecret(this.secret);
    customAuthenticationFilter.setSecret(this.secret);

    http
        .csrf().disable()
        .authorizeRequests()
        .antMatchers( ...antPatterns: "/login", "/user/register", "/ws")
        .permitAll()
        .antMatchers(HttpMethod.OPTIONS, ...antPatterns: "/**")
        .permitAll()
        .anyRequest()
        .authenticated()
        .and()
        .exceptionHandling()
        .and()
        .addFilter(customAuthenticationFilter);

    http.addFilterBefore(customAuthorisationFilter, UsernamePasswordAuthenticationFilter.class);
}
```

Figura 3.3.4 – Metoda principală pentru configurarea securității.

În metoda configure (supra-scrierea acesteia care primește ca și parametru un obiect de tipul HttpSecurity), se realizează toate aceste configurații. Cele mai importante dintre acestea sunt setarea filtrelor, setarea secretului utilizat pentru criptarea token-ului, setarea de permișii pentru endpoint-urile ce nu trebuie să fie securizate („/login”, „/user/register”, „/ws”) și ordonarea filtrelor.

La nivelul obiectului primit ca și parametru se aplică aceste schimbări prin procedeul de „method chaining” (înlănțuirea metodelor). Schimbările se aplică de sus în jos, astfel toate cererile care nu trebuie să fie securizate sunt permise, la fel și pentru cererile ce sunt de tip „OPTIONS” deoarece acestea sunt realizate de către browser pentru a obține niște informații în legătură cu cererea pe care urmează să o facă. Restul cazurilor, trebuie neapărat să treacă prin filtrele înregistrate.

3.3.2 Filtre

Fiecare cerere securizată din platforma FileStorm, trece prin cele două filtre înregistrate, ele având rolul de a procesa datele venite odată cu cererea și de a se asigura că sunt valide în cazul înregistrării de credențiale noi sau în cazul logării, folosindu-se de un așa-numit „JWT”.

JWT sau JSON Web Token reprezintă un standard pentru crearea de date cu semnătură optională și/sau criptare optională, care poate să conțină în componența acestuia și alte date. În cazul FileStorm, JWT-ul este criptat cu algoritmul HMAC256 și conține un secret utilizat în criptare iar rolul și adresa de email a utilizatorului că și date suplimentare.

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbkBhZG1pbj20iLCJyb2xlcyI6WyJhZG1pbjJdLCJpc3Mi0iJodHRwczovL21hcmlhbmcNyLmdvLnJv0jgwODAvbG9naW4iLCJleHAi0jE2NTQzMjI1NzJ9.PudIU6n0fCXMQgpqjm5MjqeGsuXQ6fxibZzn5Q2zRbM|
```

Figura 3.3.5 – Exemplu de JWT folosit pe platforma FileStorm(forma encodată).[22]

Un JWT este împărțit în 3 componente : antet, conținut și semnătură. Aceste parti sunt encodeate individual folosindu-se Base64URL iar mai apoi sunt concatenate folosindu-se „.” că și separator. Antetul conține în principal date despre tipul de token și algoritmul folosit de acesta. În conținutul token-ului se află datele suplimentare, în acest caz, email-ul și rolul utilizatorului ce deține acest token.

Semnătura este calculată după un proces ce cuprinde următorii pași : se encodează atât antetul cât și conținutul folosind Base64URL, se concatenează folosindu-se „.” că și separator împreună cu o cheie secretă de 256 de biți iar mai apoi se aplică algoritmul HMACSHA256.

```
HEADER:  
  
{  
  "typ": "JWT",  
  "alg": "HS256"  
}  
  
PAYLOAD:  
  
{  
  "sub": "admin@admin.com",  
  "roles": [  
    "admin"  
  ],  
  "iss": "https://mariancr.go.ro:8080/login",  
  "exp": 1654302572  
}  
  
VERIFY SIGNATURE  
  
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  your-256-bit-secret  
)  secret base64 encoded
```

Figura 3.3.6 – Exemplu de JWT (forma decodată)

În cadrul aplicației, acest token este creat la nivelul filtrului de autorizare, accesat printr-o cerere la „/login” cu credentialele necesare și mai apoi este setat că și Cookie, având cheia „app-jwt”. Acest token este mai apoi citit din listă de Cookie-uri și validat la momentul inițierii oricărei cereri securizate la nivelul filtrului de autentificare.

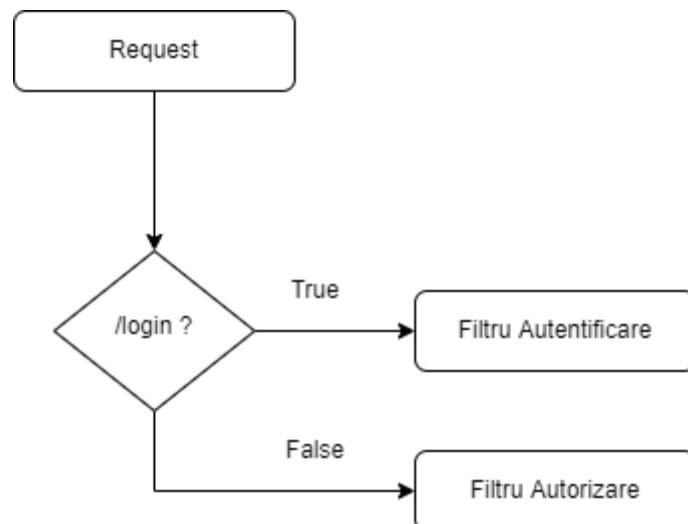


Figura 3.3.7 – Diagrama de activitate pentru cele două filtre.

Filtrele sunt acționate în funcție de calea la care face referire cererea HTTP. Astfel dacă cererea este la „/login”, se va acționa filtrul de autentificare, iar în caz contrar, cel de autorizare.

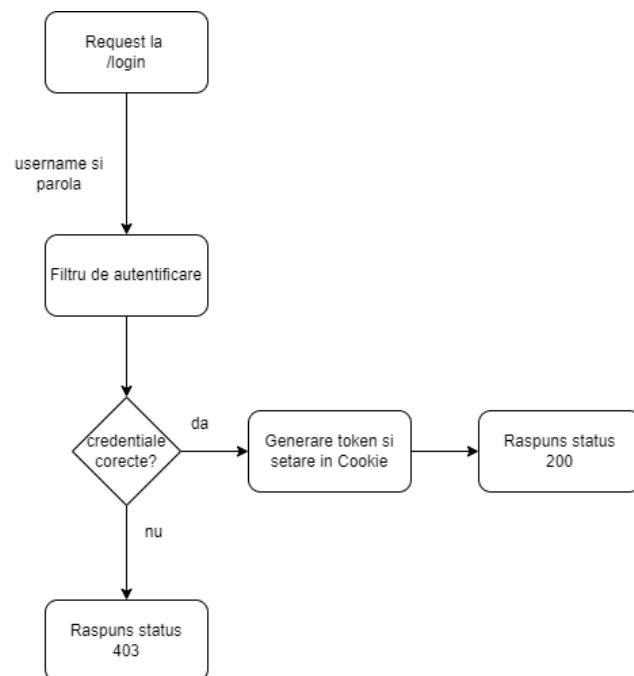


Figura 3.3.8 – Diagrama filtrului de autentificare.

Filtrul de autentificare, interceptează cererile HTTP și le procesează doar pe acele care au calea „/login”.

```
@Override  
public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {  
    String email = request.getParameter("email");  
  
    byte[] decodedPassword = decoder.decode(  
        request.getParameter("password"));  
  
    String password = new String(decodedPassword);  
  
    log.info("Email is {}", email);  
    log.info("Password after decode is {}", password);  
  
    UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(email, password);  
    return authenticationManager.authenticate(authenticationToken);  
}
```

Figura 3.3.9 – Metoda attemptAuthentication(...).

Metoda attemptAuthentication de mai sus, este cea apelată inițial în momentul în care este acționat filtrul. Se iau email-ul și parola din cererea de logare care mai apoi vor fi validate. Validarea se realizează automat, folosindu-se un manager de autentificări definit în clasa de configurare a Spring Security.

```
@Override  
protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain, Authentication authentication) {  
    User user = (User)authentication.getPrincipal();  
    Algorithm algorithm = Algorithm.HMAC256(secret.getBytes());  
  
    String accessToken = createAccessToken(request, user, algorithm);  
    Cookie cookie = new Cookie("app-jwt", accessToken);  
    response.addCookie(cookie);  
}  
  
1 usage  ▾ mariannpmd  
private String createAccessToken(HttpServletRequest request, User user, Algorithm algorithm) {  
    return JWT.create()  
        .withSubject(user.getUsername())  
        .withExpiresAt(new Date(System.currentTimeMillis() + 3000 * 60 * 1000)) //3000mins  
        .withIssuer(request.getRequestURL().toString())  
        .withClaim("roles", user.getAuthorities().stream().map(GrantedAuthority::getAuthority).collect(Collectors.toList()))  
        .sign(algorithm);  
}
```

Figura 3.3.10 – Metodele successfulAuthentication și createAccessToken.

După cum reiese și din figura 3.9, în cazul în care credentialele nu sunt corecte, filtrul nu va mai continua și va returna un răspuns ce vă avea statusul 403 (Forbidden).

În cazul în care, datele din cerere sunt corecte, se continuă procesarea cererii cu metoda successfulAuthentication. Se ia utilizatorul din contextul de securitate al aplicației (aceste fiind tot un beneficiu adus de Spring Security) și se definește algoritmul care urmează să fie folosit, în acest caz HMAC256, la care se trimite că și parametrul secretul aflat în fișierul de proprietăți al aplicației(application.properties).

JWT-ul este creat în metoda createAccessToken și conține numele utilizatorului (în acest caz este defapt email-ul), timpul de expirare (3000 de minute), cine a inițiat cererea, rolurile și la final este creată semnatura.

După crearea token-ului acesta este setat că și Cookie, pentru a fi valabil la orice altă cerere care urmează.

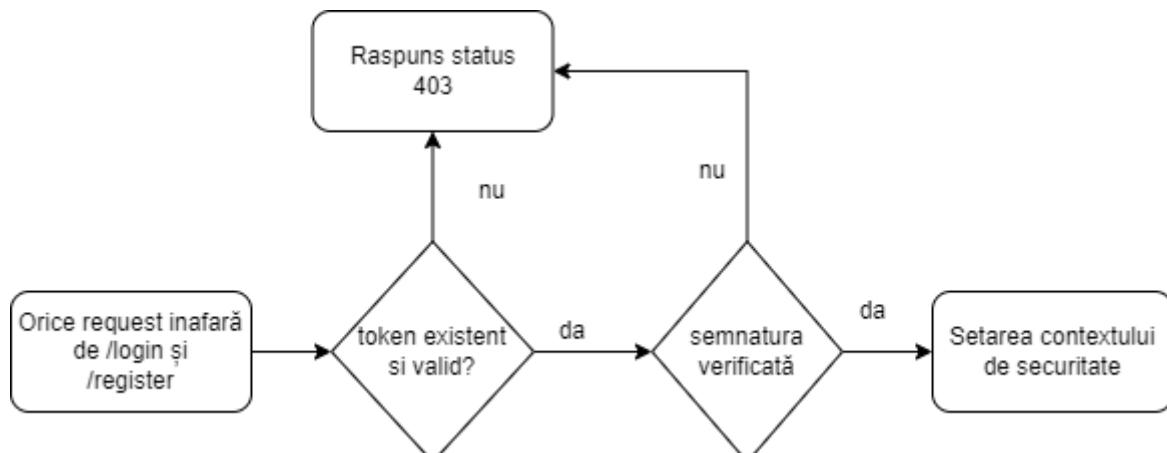


Figura 3.3.11 – Diagrama de activitate pentru filtrul de autorizare.

Filtrul de autorizare este apelat de fiecare dată pentru fiecare cerere care este securizată și nu una de logare sau înregistrare. Acesta are scopul de a verifica token-ul prezent în Cookie-uri.

```
@Override  
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {  
    if (request.getServletPath().equals("/Login") ||  
        request.getServletPath().equals("/user/register")) {  
        filterChain.doFilter(request, response);  
    } else {  
        Cookie[] cookies = request.getCookies();  
        Cookie jwtCookie = null;  
        if (!ArrayUtils.isEmpty(cookies))  
            for (Cookie cookie : cookies) {  
                if (cookie.getName().equals("app-jwt")) {  
                    jwtCookie = cookie;  
                    break;  
                }  
            }  
            if (jwtCookie != null && !StringUtils.isEmpty(jwtCookie.getValue())) {...} else {  
                filterChain.doFilter(request, response);  
            }  
    }  
}
```

Figura 3.3.12 – Metoda doFilterInternal din filtrul de autorizare (partea de verificare a existenței token-ului).

Metoda doFilterInternal, similar cu attemptAuthentication din filtrul de autentificare, este apelată în momentul în care este acționat filtrul.

De asemenea, este de menționat faptul că filtrul de autorizare, se află în fața filtrului de autentificare, astfel el este primul care interceptează cererile, și deci este nevoie să sară peste cererile la căile „/login” și „/user/register” astfel încât să se acționeze celălalt filtru.

Metoda preia inițial token-ul, accesând lista de Cookie-uri, și selectându-l pe acela care are cheia „app-jwt”, după care este verificat dacă există și nu este null sau gol.

```
if (jwtCookie != null && !StringUtils.isEmpty(jwtCookie.getValue())) {
    try {
        String token = jwtCookie.getValue();
        Algorithm algorithm = Algorithm.HMAC256(secret.getBytes());

        JWTVerifier verifier = JWT.require(algorithm).build();
        DecodedJWT decodedJWT = verifier.verify(token);
        String username = decodedJWT.getSubject();
        String[] roles = decodedJWT.getClaim("roles").asArray(String.class);
        Collection<SimpleGrantedAuthority> authorities = new ArrayList<>();
        Arrays.stream(roles)
            .forEach(value -> {
                authorities.add(new SimpleGrantedAuthority(value));
            });

        UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(username, credentials: null, authorities);
        SecurityContextHolder.getContext().setAuthentication(authenticationToken);
        filterChain.doFilter(request, response);
    } catch (Exception exception) {
        log.error("Error logging in: {}", exception.getMessage());
        throw new LoginErrorException("Login has failed");
    }
}

else {
    filterChain.doFilter(request, response);
}
```

Figura 3.3.13 - Metoda doFilterInternal din filtrul de autorizare (partea de validare a tokenului).

După ce este confirmat că token-ul există, se preia din nou secretul din fișierul de proprietăți și se verifică semnătura tokenului decodat. Dacă verificarea eșuează, se aruncă o excepție ce returnează un răspuns cu statusul 403. Dacă verificarea a funcționat, se setează contextul de securitate a aplicației pentru utilizatorul respectiv, astfel se vă poate să ști ce utilizator este autentificat în momentul apelării unui anumit endpoint.

În urma procesării cererii la nivelul filtrului de autorizare, execuția programului continua cu Controller-ul în care este definit endpoint-ul apelat.

3.3.3 Controllere

Un controller reprezintă o clasă care are rol de punct de intrare într-un Web API și este nivelul la care se definesc endpoint-urile. Acestea se disting de restul claselor prin adnotarea @Controller sau @RestController.

Controllerele prezente în aplicația FileStorm sunt : UserController, FileController, DirectoryController, NotificationController.

```
• Petrica Marian +1
@RestController
@RequestMapping("/user")
@RequiredArgsConstructor
public class UserController {
    4 usages
    private final UserService userService;
    1 usage
    private final DirectoryService directoryService;
    2 usages
    private final FileStoreService fileStoreService;
    1 usage
    private final NotificationService notificationService;
```

Figura 3.3.14 – Declaratia de clasa, adnotarile si dependințele din UserController.

Pe lângă adnotarea @RestController de regula se mai scrie și @RequestMapping care are rolul de a defini calea de baza de la nivelul respectivului controller, astfel toate căile către UserController vor începe neapărat cu „/user”.

În figură se mai observă și faptul că sunt declarate câteva bean-uri (dependințe) care sunt injectate în acest controller.

La nivelul acestui controller se realizează operații ce au la bază utilizatorii, de exemplu returnarea tuturor utilizatorilor înregistrați în sistem, setarea de spațiu de stocare, ștergerea și.a.m.d. Endpoint-urile din acest controller corespund următoarelor căi : “/all”, “/info”, “/register”, “/delete”, “/assign”.

```
@PreAuthorize("hasAuthority('admin')")
@GetMapping("/all")
public ResponseEntity<List<UserDTO>> getAllUsers() {
    List<UserDTO> allUsers = userService.getAllUsers();
    return ResponseEntity.ok(allUsers);
}
```

Figura 3.3.15 – Metoda care corespunde endpoint-ului “/all”.

Metoda getAllUsers este cea apelată în momentul în care se face o cerere către calea “/all” și este de tip GET. De asemenea se poate observa existența adnotării @PreAuthorize care are ca și parametru expresia : “hasAuthority(‘admin’)”. Această adnotare face ca acest endpoint să fie disponibil doar utilizatorilor logați cu un cont de administrator. În cazul în care este accesat de către un user normal, se va returna un răspuns cu statusul 403.

Acest endpoint face un apel la UserService ce returnează o lista ce conține informații despre toți utilizatorii platformei. Lista este apoi încapsulată într-un ResponseEntity (deși nu este neapărat necesar) iar mai apoi vă fi convertită automat la formatul JSON.

```
[ {  
    {  
        "id": 74,  
        "email": "adelin_murdeala@yahoo.ro",  
        "role": "user",  
        "assignedSpace": 1000000000,  
        "occupiedSpace": 3357566  
    },  
    {  
        "id": 60,  
        "email": "admin@admin.com",  
        "role": "admin",  
        "assignedSpace": 2000000000,  
        "occupiedSpace": 14762830  
    },  
    {  
    }  
}
```

Figura 3.3.16 – Exemplu de răspuns după apelarea endpointului „/all”

```
@GetMapping("/info")  
public ResponseEntity<?> getUserInfo(String email){  
    String authenticatedUserEmail = SecurityContextHolder.getContext()  
        .getAuthentication()  
        .getPrincipal()  
        .toString();  
    if (!authenticatedUserEmail.equals(email)){  
        return new ResponseEntity<>(body: "Not allowed", HttpStatus.FORBIDDEN);  
    }  
    UserDTO userDTOByEmail = userService.getUserDTOByEmail(email);  
    return ResponseEntity.ok(userDTOByEmail);  
}
```

Figura 3.3.17 – Metoda care corespunde endpoint-ului „/info”

Metoda getUserInfo corespunde endpoint-ului „/info”, este de tip GET și are rolul de a returna informații cu privire la utilizatorul logat.

Înțial se preia adresa utilizatorului logat, se compară cu cea primită că și parametru în cerere, iar în cazul în care acestea nu corespund, se aruncă o excepție. Ulterior, se apelează UserService pentru returnarea de date referitoare la utilizator și se crează răspunsul care va fi de tip JSON.

```
{ □
    "id": 60,
    "email": "admin@admin.com",
    "role": "admin",
    "assignedSpace": 2000000000,
    "occupiedSpace": 14762830
}
```

Figura 3.3.18 – Răspunsul JSON returnat de endpointul „/info”.

```
@PostMapping("register")
public ResponseEntity<?> registerNewUser(@RequestBody UserAuthDTO user) {
    UserEntity userEntity = userService.registerNewUser(user.email(), user.password(), role: "user");
    boolean wasSuccessful = this.fileStoreService.createUserDirectory(userEntity);
    return ResponseEntity.ok(user);
}
```

Figura 3.3.19 – Metoda ce corespunde endpoint-ului „/register”.

Metoda registerNewUser se așteaptă la existența unui RequestBody la nivelul cererii HTTP care să conțină detaliile utilizatorului ce urmează să fie logat și este de tip POST.

Se face un apel către UserService pentru adăugarea unui nou utilizator în sistem, și la DirectoryService pentru crearea unui director pentru acel utilizator.

Sunt returnate ca și răspuns datele utilizatorului proaspăt înregistrat.

```
{ □
    "id": 69,
    "email": "test@test.com",
    "role": "user",
    "assignedSpace": 0,
    "occupiedSpace": 0
}
```

Figura 3.3.20 – Răspunsul JSON returnat de endpoint-ul „/register”

```

@PreAuthorize("hasAuthority('admin')")
@DeleteMapping("/delete")
public ResponseEntity<UserDTO> deleteUserById(@RequestParam Long userId) throws IOException {
    UserDTO userDTO = directoryService.deleteUserById(userId);

    return ResponseEntity.ok(userDTO);
}
  
```

Figura 3.3.21 – Metoda care corespunde endpoint-ului „/delete”.

Metoda deleteUserById se așteaptă la existența unui parametru „userId” în cerere, corespunde endpoint-ului „/delete”, și este de tip DELETE. De asemenea, este necesar ca utilizatorul să fie logat cu contul de administrator pentru a avea acces.

Se inițiază un apel la DirectoryService care se va ocupa de ștergerea tuturor directoarelor, a fișierelor și a tuturor datelor referitoare la acel utilizator.

La final se returnează datele utilizatorului șters.

```

{
    "id": 55,
    "email": "testab@testab.com",
    "role": "user",
    "assignedSpace": 1000000000,
    "occupiedSpace": 0
}
  
```

Figura 3.3.22 – Răspunsul JSON returnat de endpoint-ul „/delete”

```

@PreAuthorize("hasAuthority('admin')")
@PostMapping("/assign")
public ResponseEntity<UserDTO> assignToUser(
    @RequestParam Long userId,
    @RequestBody AssignRequestDTO assignRequest
) {
    Long usableSpace = fileStoreService.getSystemInfo().usableSpace();
    UserDTO userDTO = userService.assignToUser(userId, assignRequest.amount(), usableSpace);

    notificationService.notifyUserForAssignment(userDTO, assignRequest.amount(), assignRequest.description());

    return ResponseEntity.ok(userDTO);
}
  
```

Figura 3.3.23 – Metoda care corespunde endpoint-ului „/assign”.

Metoda assignToUser se așteaptă la existența unui parametru „userId” și a unor date de atribuire a spațiului de stocare, poate fi utilizată doar de către administrator și este de tip POST.

Inițial se preia spațiul utilizabil din FileStoreService pentru a fi pasat către UserService astfel încât, să se asigure că nu se atribuie mai mult spatiu decât este disponibil.

De asemenea se apelează și serviciul de notificări pentru a trimite o notificare utilizatorului care a primit spațiul de stocare.

La final sunt returnate datele utilizatorului asupra căruia s-au făcut aceste modificări.

```
{ □  
  "id":71,  
  "email":"test1@test.com",  
  "role":"user",  
  "assignedSpace":1500000000,  
  "occupiedSpace":0  
}
```

Figura 3.3.24 – Răspunsul JSON returnat de endpoint-ul „/assign”.

```
@RestController  
@RequestMapping(“/file”)  
@Slf4j  
@RequiredArgsConstructor  
public class FileController {  
  
    7 usages  
    private final FileStoreService fileStoreService;  
    3 usages  
    private final UserService userService;
```

Figura 3.3.25 - Declarația de clasă, adnotările și dependințele din FileController.

Clasa FileController, asemănător cu UserController are o cale de bază, în acest caz, toate cererile către acest controller trebuie să înceapă cu „/file”. De asemenea aceasta are două dependințe, către FileStoreService și UserService.

Endpoint-urile din acest controller sunt : „/upload”, „/check”, „/all”, „/one”, „/delete/one”, „/byKeyword” și „/systemInfo”.

```
@PostMapping("/upload")
public ResponseEntity<FileEntityDTO> uploadFile(@RequestParam final MultipartFile file,
                                                 @RequestParam final ArrayList<String> pathFromRoot,
                                                 @RequestParam(required = false) final boolean shouldUpdate)
    log.info("New file to be uploaded : {}", file.getOriginalFilename());

    FileEntityDTO fileEntityDTO = fileStoreService.uploadNewFile(file, pathFromRoot, shouldUpdate);

    return ResponseEntity.ok(fileEntityDTO);
}
```

Figura 3.3.26 – Metoda care corespunde endpoint-ului „/upload”.

Această metodă se așteaptă să primească un fișier și doi parametrii : calea la care se află utilizatorul în aplicație exprimată sub forma unui sir și relativă la directorul de bază denumit „pathFromRoot” și o valoare booleană denumita „shouldUpdate”.

Endpoint-ul citește acest fișier într-un buffer datorită utilizării MultipartFile, acest lucru se datorează faptului că am vrut că aplicația să poată să încarce fișiere de dimensiuni considerabile. Ulterior, după ce fișierul este salvat în sistem și în baza de date ca urmare a apelului la FileStoreService, se returnează detaliile fișierului nou adăugat.

De menționat este faptul că tot acest endpoint se folosește și în modificarea unui fișier deja existent (cu același nume la aceeași cale) prin trimitera valorii true ca și parametru în variabila shouldUpdate.

```
{
    "id": 100,
    "name": "arhitectura backend.drawio (3).png",
    "path": "/home/marian/OwnCloud/admin@admin.com/arhitectura backend.drawio (3).png",
    "size": 107896,
    "fileType": "IMAGE"
}
```

Figura 3.3.27 - Răspunsul JSON returnat de endpoint-ul „/upload”.

```
@GetMapping("/check")
public ResponseEntity<Boolean> checkFile(@RequestParam ArrayList<String> pathFromRoot,
                                            @RequestParam String filename) {
    boolean fileExists = fileStoreService.checkIfExists(filename, pathFromRoot);
    if (fileExists) {
        return ResponseEntity.ok().body(true);
    }

    return ResponseEntity.ok().body(false);
}
```

Figura 3.3.28 - Metoda care corespunde endpoint-ului „/check”.

Metoda checkFile primește 2 parametrii : calea relativă la directorul de bază la care se află utilizatorul și numele unui fișier. De asemenea endpoint-ul este de tip GET.

Este apelat FileStoreService pentru a verifica dacă fișierul dat ca parametru există deja la calea primită că și parametru.

Astfel, în funcție de caz, este returnat ca și răspuns o valoare boolean (true sau false).

```
@GetMapping("all")
public ResponseEntity<Page<FileEntityDTO>> getAllFilesForUser(String sortBy,
                                                               int page,
                                                               int size,
                                                               boolean asc,
                                                               @RequestParam ArrayList<String> pathFromRoot) {
    var userEmail = (String) SecurityContextHolder.getContext().getAuthentication().getPrincipal();

    Page<FileEntityDTO> allFilesForUser = fileStoreService
        .getAllFilesForUser(userEmail, sortBy, page, size, asc, pathFromRoot);

    return ResponseEntity.ok(allFilesForUser);
}
```

Figura 3.3.29 - Metoda care corespunde endpoint-ului „/all”.

Metoda getAllFilesForUser primește următorii parametrii : „sortBy” – se referă la câmpul după care să se sorteze datele, „page” – se referă la pagina din care vrem să citim datele, „size” – numărul de entități de pe o pagină , „asc” – sortare crescătoare sau descrescătoare și „pathFromRoot” – calea relativă la care se află utilizatorul.

Acest endpoint este de tip GET și apelează serviciul FileStoreService cu toți acești parametrii pentru a returna o pagină (listă) cu informații despre fișiere sortate și ordonate într-o anumită ordine.

```
{
  "content": [
    {
      "id": 100,
      "name": "arhitectura backend.drawio (3).png",
      "path": "/home/marian/OwnCloud/admin@admin.com/arhitectura backend.drawio (3).png",
      "size": 107896,
      "fileType": "IMAGE"
    },
    {
      "id": 84,
      "name": "4afc7814346114c102a1da9eaa82597c.mp4",
      "path": "/home/marian/OwnCloud/admin@admin.com/4afc7814346114c102a1da9eaa82597c.mp4",
      "size": 4170297,
      "fileType": "VIDEO"
    }
  ]
}
```

Figura 3.3.30 - Răspunsul JSON returnat de endpoint-ul „/all” sortate după ultima modificare în ordine descrescătoare.

```
@GetMapping("one")
public StreamingResponseBody getFileFromUserAndId(HttpServletRequest response, @RequestParam Long id) {
    String userEmail = (String) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    UserEntity userByEmail = userService.getUserByEmail(userEmail);
    File file = fileStoreService.getFileByIdAndUser(id, userByEmail);
    String fileName = file.getName();

    response.setHeader(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=" + fileName);
    response.setHeader(HttpHeaders.ACCESS_CONTROL_EXPOSE_HEADERS, HttpHeaders.CONTENT_DISPOSITION);

    return outputStream -> {
        FileInputStream inputStream = new FileInputStream(file);
        IOUtils.copyLarge(inputStream, response.getOutputStream());
        inputStream.close();
    };
}
```

Figura 3.3.31 - Metoda care corespunde endpoint-ului „/one”.

Metoda getFileFromUserAndId primește doi parametrii, un id de tip Long și un obiect response de tip HttpServletRequest și este de tip GET.

Înțial se ia din contextul de securitate, email-ul utilizatorului logat, astfel încât UserService-ul să returneze entitatea crespunzatoare utilizatorului. Serviciul dedicat pentru fișiere se folosește de acest ușer că și parametru și returnează o referință la fișierul indicat prin id. Ulterior se setează două valori în antetul răspunsului care au rolul de a da nume fișierului și a-i oferi acces în cerere.

La final este returnat fișierul sub forma unui flux, în aşa fel încât acesta nu este încărcat în memorie în întregime ci se descarcă dintr-un buffer.

Acest endpoint nu returnează JSON ci doar octeți din care este compus fișierul, până când acesta este complet descărcat.

```
@DeleteMapping("delete/one")
public ResponseEntity<String> deleteFileFromUserAndId(@RequestParam Long id) {
    String userEmail = (String) SecurityContextHolder.getContext()
        .getAuthentication().getPrincipal();
    UserEntity userByEmail = userService.getUserByEmail(userEmail);

    if (fileStoreService.deleteFileByIdAndUser(id, userByEmail)) {
        userService.recomputeUserStorage(userByEmail);
        return ResponseEntity.ok().build();
    }

    return ResponseEntity.status(HttpStatus.EXPECTATION_FAILED)
        .body("File was not deleted");
}
```

Figura 3.3.32 - Metoda care corespunde endpoint-ului „/delete/one”.

Metoda deleteFileFromUserAndId primește ca și parametru un id de tip Long și este de tip DELETE.

Se începe prin a selecta utilizatorul folosind contextul de securitate al aplicației.

Ulterior, se apelează serviciul dedicat pentru fișiere pentru a iniția ștergerea fișierului indicat prin id.

La final, se recalculează informațiile referitoare la spațiul de stocare al utilizatorului și este returnat un simplu răspuns gol cu statusul 200 în cazul în care această operație este realizată cu succes. În caz contrar se returnează un răspuns cu statusul 417.

```
@GetMapping("/byKeyword")
public ResponseEntity<?> findAllLike(String keyword) {
    List<FileEntityDTO> allFilesLike = fileStoreService.getAllFilesLike(keyword);

    return ResponseEntity.ok(allFilesLike);
}
```

Figura 3.3.33 - Metoda care corespunde endpoint-ului „/byKeyword”.

Metoda findAllLike primește ca și parametru un sir de caractere denumit „keyword” și este de tip GET.

Acest endpoint are rolul de a returna toate datele despre fișierele care au un keyword în componența numelui lor.

```
[{"id": 72, "name": "Untitled Diagram.drawio.png", "path": "/home/marian/OwnCloud/admin@admin.com/Untitled Diagram.drawio.png", "size": 17402, "fileType": "IMAGE"}, {"id": 100, "name": "arhitectura backend.drawio (3).png", "path": "/home/marian/OwnCloud/admin@admin.com/arhitectura backend.drawio (3).png", "size": 107896, "fileType": "IMAGE"}]
```

Figura 3.3.34 - Raspunsul JSON returnat de endpoint-ul „/byKeyword” cu keyword = 'png'.

```
@GetMapping("/systemInfo")
@PreAuthorize("hasAuthority('admin')")
public ResponseEntity<?> getSystemInfo(){
    SystemInfoDTO systemInfo = fileStoreService.getSystemInfo();
    return ResponseEntity.ok(systemInfo);
}
```

Figura 3.3.35 - Metoda care corespunde endpoint-ului „/systemInfo”.

Metoda `getSystemInfo` poate fi accesată doar de către un administrator logat și este de tip GET.

Aceasta are rolul de a returna date referitoare la sistem precum spațiul disponibil sau spațiul utilizabil.

```
{ "totalSpace" : 490574913536,  
  "usableSpace" : 446428904960  
}
```

Figura 3.3.36 - Răspunsul JSON returnat de endpoint-ul „/systemInfo”.

```
@RestController  
@RequestMapping("/dir")  
@RequiredArgsConstructor  
public class DirectoryController {  
    private final DirectoryService directoryService;
```

Figura 3.3.37 - Declarația de clasă, adnotările și dependințele din DirectoryController.

Clasa `DirectoryController`, la fel ca restul controllerelor are o cale de bază, în acest caz, toate cererile către acest controller trebuie să înceapă cu „/dir”. Această clasă are o singură dependință și anume `DirectoryService`.

Endpoint-urile din acest controller sunt : „/create”, „/getAll”, „/delete”.

```
@PostMapping("/create")  
public ResponseEntity<DirectoryDTO> createDirectory(@RequestBody ArrayList<String> pathsFromRoot) {  
    var directory : DirectoryDTO = directoryService.createDirectory(pathsFromRoot);  
    return ResponseEntity.ok(directory);  
}
```

Figura 3.3.38 - Metoda care corespunde endpoint-ului „/create”.

Metoda `createDirectory` acceptă ca și parametru în corpul cererii calea curentă a utilizatorului exprimată ca și un sir de cuvinte.

Aceasta are rolul de a apela serviciul de directoare pentru a crea un nou director, la calea primită ca și parametru și returnează detaliile referitoare la acesta.

```
{
    "id": 3,
    "path": "C:\\\\Users\\\\ZZ03FL826\\\\OwnCloud\\\\admin@admin.com\\\\main",
    "name": "main"
}
```

Figura 3.3.39 - Răspunsul JSON returnat de endpoint-ul „/create”.

```
@PostMapping("/getAll")
public ResponseEntity<DirectoriesWithParentDTO> getAllDirectoriesInPath(@RequestBody ArrayList<String> pathsFromRoot){
    DirectoriesWithParentDTO allDirectoriesInPath = directoryService.getAllDirectoriesInPath(pathsFromRoot);
    return ResponseEntity.ok(allDirectoriesInPath);
}
```

Figura – 3.3.40 - Metoda care corespunde endpoint-ului „/getAll”.

Metoda getAllDirectoriesInPath acceptă ca și parametru în corpul cererii calea curentă a utilizatorului exprimată că și un sir de cuvinte.

Aceasta are rolul de a apela serviciul de directoare pentru a lua date referitoare la toate directoarele de la calea primită ca și parametru și le returnează. De asemenea este menționat separat directorul părinte.

```
{
    "parent": null,
    "directories": [
        {
            "id": 3,
            "path": "C:\\\\Users\\\\ZZ03FL826\\\\OwnCloud\\\\admin@admin.com\\\\main",
            "name": "main"
        }
    ]
}
```

Figura 3.3.41 - Răspunsul JSON returnat de endpoint-ul „/getAll”.

```
@DeleteMapping("/delete")
public ResponseEntity<?> deleteDirById(@RequestParam Long id) throws IOException {
    DirectoryDTO directoryDTO = directoryService.deleteDirById(id);
    return ResponseEntity.ok(directoryDTO);
}
```

Figura 3.3.42 - Metoda care corespunde endpoint-ului „/delete”.

Metoda deleteDirById acceptă ca și parametru id-ul care corespunde directorului care urmează să fie șters.

Aceasta are rolul de a apela serviciul de directoare pentru a șterge directorul cu id-ul primit că și parametru dar și toate fișierele din interiorul acestuia. Se returnează informații despre directorul șters.

```
{
  "id": 3,
  "path": "C:\\\\Users\\\\ZZ03FL826\\\\OwnCloud\\\\admin@admin.com\\\\main",
  "name": "main"
}
```

Figura 3.3.43 - Răspunsul JSON returnat de endpoint-ul „/delete”.

```

● Petrica Marian *
@RestController
@RequestMapping("/notification")
@RequiredArgsConstructor
public class NotificationController {
    3 usages
    private final NotificationService notificationService;
}
```

Figura 3.3.44 - Declarația de clasă, adnotările și dependințele din NotificationController.

Clasa NotificationController, are o calea de baza „/notification”. De asemenea aceasta are o singură dependință și anume către NotificationService.

Endpoint-urile din acest controller sunt : „/admin”, „/all”, „/all/update”.

```

@PostMapping("/admin")
public ResponseEntity<?> notifyAdmin(@RequestBody UserStorageRequest userStorageRequest) {
    notificationService.notifyAdminForUserRequest(userStorageRequest);

    return ResponseEntity.ok( body: "SENT");
}
```

Figura 3.3.45 - Metoda care corespunde endpoint-ului „/admin”.

Metoda notifyAdmin acceptă ca și parametru în corpul cererii un obiect de tipul UserStorageRequest, ce conține detalii despre o eventuală cerere de atribuire a spațiului de stocare și este de tip POST.

Aceasta are rolul de a apela serviciul de notificări pentru a trimite o notificare către administrator. La final , se raspunde cu statusul 200 și textul „SENT”.

```

@GetMapping("/all")
public ResponseEntity<List<NotificationDTO>> getAllNotifications(){
    List<NotificationDTO> allNotificationsOrdered = notificationService.getAllNotificationsOrdered();
    return ResponseEntity.ok(allNotificationsOrdered);
}
```

Figura 3.3.46 - Metoda care corespunde endpoint-ului „/all”.

Metoda getAllNotifications nu are nevoie de nici un fel de parametru și este de tip GET.

Aceasta are rolul de a apela serviciul de notificări pentru a returna toate notificările utilizatorului logat.

```
[ {  
    {  
        "id": 5,  
        "description": "admin@admin.com:test:5 GB",  
        "dateTime": "2022-06-08T01:17:26.538547",  
        "notificationState": "UNREAD"  
    },  
    {  
        "id": 4,  
        "description": "admin@admin.com:notify pls\n:2 GB",  
        "dateTime": "2022-06-08T01:17:14.974524",  
        "notificationState": "READ"  
    }  
}
```

Figura 3.3.47 - Răspunsul JSON returnat de endpoint-ul „/all”.

```
@GetMapping("/all/update")  
public ResponseEntity<?> updateNotificationsState(){  
    notificationService.updateNotificationsState();  
    return ResponseEntity.ok( body: "");  
}
```

Figura 3.3.48 - Metoda care corespunde endpoint-ului „/all/update”.

Metoda updateNotificationsState nu are parametrii și este de tip GET.

Aceasta are rolul de a apela serviciul de notificări pentru a schimba starea tuturor notificărilor utilizatorului logat din „UNREAD” în „READ”. La final se trimită un răspuns cu statusul 200.

3.3.4 Servicii

Serviciile, după cum au mai fost explicate și în capitolul trecut sunt niste bean-uri (obiecte ce se folosesc de proprietățile de Dependency Injection din Spring) ce se ocupă strict de partea de logica a unei operații. În cazul FileStorm, ele funcționează de asemenea și ca un intermediar între repository-uri și controlere care realizează operațiile propriu-zise iar mai apoi returnează un rezultat.

Serviciile din cadrul aplicației sunt : UserService, FileStoreService, DirectoryService și NotificationService.

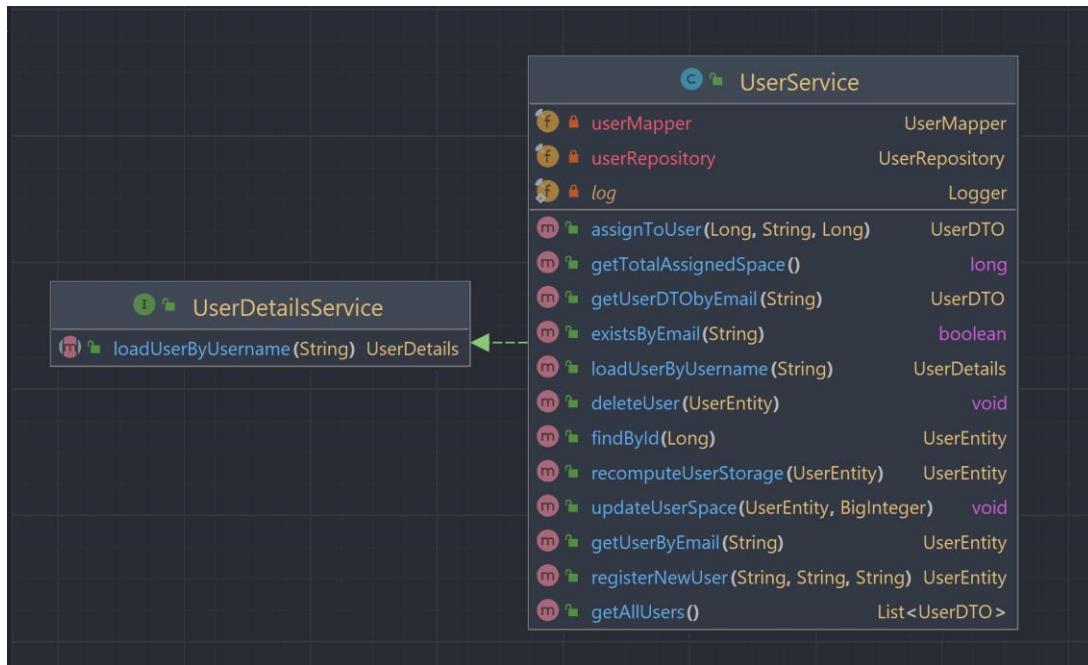


Figura 3.3.49 – Diagrama de clasă a serviciului UserService.

Acest serviciu are 2 dependințe : UserMapper – clasa folosită pentru a translata un obiect din entitate, într-un POJO (Plain Old Java Object) și UserRepository – interfață din Spring Data JPA pentru comunicare cu baza de date.

Printre metodele cele mai importante ale serviciului se enumeră : loadUserByUsername, getAllUsers, getUserDTObyEmail, registerNewUser și assignToUser.

```
@Override
public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
    UserEntity byEmail = userRepository.findByEmail(email)
        .orElseThrow();
    //check password
    log.info("User FOUND in the db" + email);
    Collection<SimpleGrantedAuthority> authorities = new ArrayList<>();
    authorities.add(new SimpleGrantedAuthority(byEmail.getRole()));
    return new User(byEmail.getEmail(), byEmail.getPassword(), authorities);
}
```

Figura 3.3.50 – Metoda loadUserByUsername din UserService.

Metoda loadByUsername este o suprascriere a metodei cu același nume, definită în interfața UserDetailsService, care este un serviciu specializat din Spring Security prin care putem marca această clasă ca și sursă pentru credentiale, în momentul în care un

utilizator se logeaza. Practica, aceasta metoda este apelata de către managerul de autentificare în momentul în care se verifică datele utilizatorului stocate în JWT.

Aceasta returnează un obiect de tip UserDetails care conține datele de autentificare ale utilizatorului în funcție de email-ul primit ca și parametru.

```
public List<UserDTO> getAllUsers() {
    List<UserEntity> all = userRepository.findAll();
    return userMapper.entitiesToDTOs(all);
}
```

Figura 3.3.51 - Metoda getAllUsers din UserService.

Această metodă are rolul de a returna o listă cu toate obiectele de tip UserEntity venite din userRepository pentru a le mapeaza la obiecte de tip UserDTO pe care le returnează mai apoi.

```
public UserDTO getUserDTObyEmail(String userEmail) {
    UserEntity userByEmail = getUserByEmail(userEmail);
    return userMapper.entityToDTO(userByEmail);
}
```

Figura 3.3.52 - Metoda getUserDTObyEmail din UserService.

Această metodă are rolul de a interoga repository-ul pentru datele referitoare la un utilizator specificat prin email, pe care le mapeaza mai apoi la obiecte de tip UserDTO și le returnează.

```
public UserEntity registerNewUser(String email, String password, String role) {
    Optional<UserEntity> byEmail = userRepository.findByEmail(email);
    if (byEmail.isPresent()) {
        throw new IllegalStateException("User already exists");
    }

    BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

    var newUser = new UserEntity(email, passwordEncoder.encode(password), role);

    return this.userRepository.save(newUser);
}
```

Figura 3.3.53 - Metoda registerNewUser din UserService.

Această metodă are rolul de a crea un nou utilizator în baza de date. Inițial se verifică dacă există deja utilizatorul, caz în care este aruncată o excepție, după care se cripteză parola și la final se salvează utilizatorul cu datele primite ca și parametru care sunt și returnate la final.

```
public UserDTO assignToUser(Long userId, String amount, Long usableSpace) {  
    long requestedAmount = FileStoreUtils.parseAmountString(amount);  
    if (requestedAmount > usableSpace)  
        throw new AbnormalAssignmentAmountException("Not enough sys space to assign!");  
  
    UserEntity userEntity = userRepository.findById(userId)  
        .orElseThrow(() -> new UsernameNotFoundException("User does not exist " + userId));  
  
    if (userEntity.getOccupiedSpace().longValueExact() > requestedAmount)  
        throw new AbnormalAssignmentAmountException("User has already occupied more space!");  
    userEntity.setAssignedSpace(BigInteger.valueOf(requestedAmount));  
  
    return userMapper.entityToDTO(userRepository.save(userEntity));  
}
```

Figura 3.3.54 - Metoda assignToUser din UserService.

Această metodă are rolul de a oferi utilizatorului referit prin parametrul userId, o anumită cantitate de spațiu de stocare definită în parametrul amount. Inițial se verifică dacă spațiul existent pe sistem este suficient iar în caz contrar este aruncată o excepție.

La final, se modifică cantitatea de stocare atribuită utilizatorului cu cea nouă și este returnat un obiect de tip UserDTO ce conține ultimele modificări.

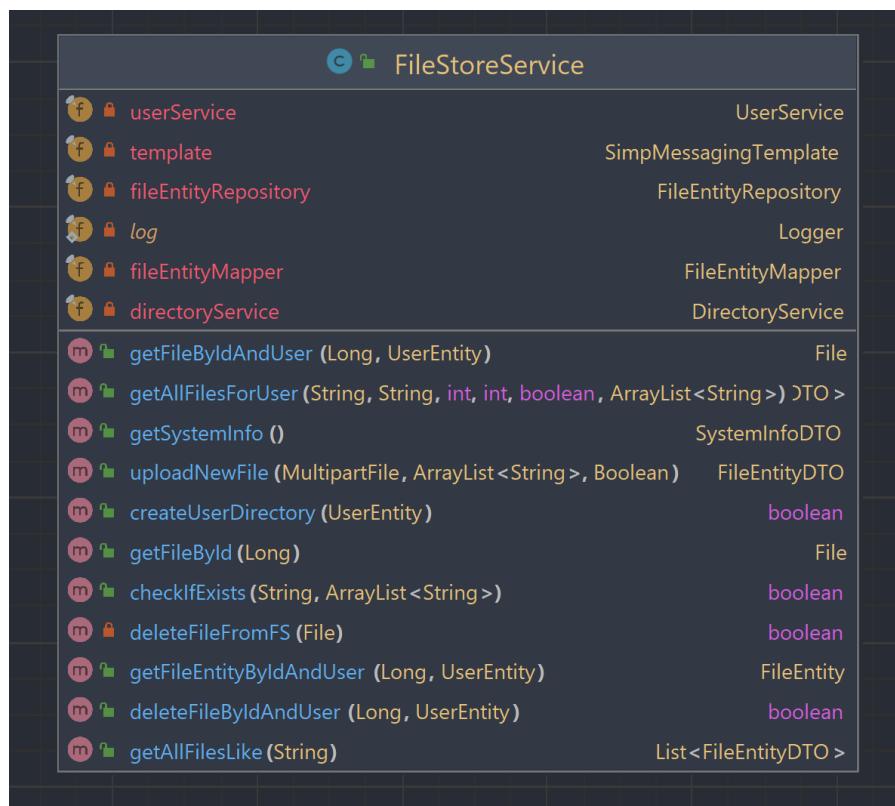


Figura 3.3.55 - Diagrama de clasă a serviciului FileStoreService.

Acest serviciu are 5 dependințe : UserService – serviciul ce se ocupă de operațiile cu utilizatorii, SimpMessagingTemplate – se ocupă de publicarea de mesaje pe topicurile deschise prin Websockets, FileEntityRepository – interfață din Spring Data JPA pentru comunicare cu baza de date, FileEntityMapper – bean ce are ca și rol maparea de la entitate la DTO și invers, DirectoryService – serviciul ce se ocupă cu operațiile cu directoare.

Printre metodele cele mai importante ale serviciului se enumera : uploadNewFile, checkIfExists, getAllFilesForUser, deleteFileByIdAndUser, getSystemInfo.

```
public FileEntityDTO uploadNewFile(MultipartFile file, ArrayList<String> pathFromRoot, Boolean shouldUpdate) throws IOException {
    BigInteger size = BigInteger.valueOf(file.getSize());

    String fileName = file.getOriginalFilename();

    String contentType = file.getContentType();
    FileType filetype = FileStoreUtils.getFileTypeFromContentType(contentType);

    String userEmail = SecurityContextHolder.getContext().getAuthentication().getPrincipal().toString();
    UserEntity userByEmail = this.userService.getUserByEmail(userEmail);

    BigInteger assignedSpace = userByEmail.getAssignedSpace();
    BigInteger occupiedSpace = userByEmail.getOccupiedSpace();

    if (occupiedSpace.add(assignedSpace).compareTo(size) == -1){
        throw new OutOfSpaceException("Not enough storage assigned to this user!");
    }

    String pathToDir = FileStoreUtils.computePathFromRoot(userByEmail.getEmail(), pathFromRoot.toString());
    boolean isDir = FileUtils.isDirectory(new File(pathToDir));
    if (!isDir){
        throw new DirectoryNotFoundException("Directory not found!");
    }
    Path finalPath = Paths.get(pathToDir, file.getOriginalFilename());
    File fileToSave = finalPath.toFile();
}
```

Figura 3.3.56 - Metoda uploadNewFile din FileStoreService (partea de verificare a spațiului).

Metoda uploadNewFile are poate cel mai important rol din aplicație. Aceasta inițial verifică toate condițiile necesare salvării unui fișier. Se verifică spațiul disponibil al utilizatorului care a inițiat cererea, după directorul în care urmează să fie salvat și mai apoi dacă respectivul fișier există deja la calea specificată în parametrul pathFromRoot.

```
FileEntity fileEntity = new FileEntity(fileName, finalPath.toString(), size, LocalDateTime.now(), filetype, userByEmail);

DirectoryEntity directoryEntity = directoryService.getDirectoryEntityFromNameAndUser(userByEmail, pathFromRoot)
    .orElseThrow(() -> new DirectoryNotFoundException("Directory was not found for user : " + userEmail
    + "and path : " + pathFromRoot));

fileEntity.setDirectory(directoryEntity);
FileEntity saved = null;
if (fileToSave.exists() && shouldUpdate.equals(Boolean.TRUE)) {
    log.info("File already exists so replace on FS and update on DB");
    if (fileToSave.delete()) {
        IOUtils.copyLarge(file.getInputStream(), Files.newOutputStream(fileToSave.toPath()));
        FileEntity existingFileEntity = fileEntityRepository.findByName(fileName)
            .orElseThrow(() -> new FileEntityNotFoundException("File was supposed to exist in the DB, SYNC ERROR."));

        FileEntity newFile = FileEntity.builder()
            .name(existingFileEntity.getName())
            .path(existingFileEntity.getPath())
            .size(size)
            .lastModified(LocalDateTime.now())
            .fileType(existingFileEntity.getFileType())
            .user(existingFileEntity.getUser())
            .directory(existingFileEntity.getDirectory())
            .build();

        fileEntityRepository.delete(existingFileEntity);
        saved = fileEntityRepository.save(newFile);
    }
}
```

Figura 3.3.57 - Metoda uploadNewFile din FileStoreService (partea de verificare a existenței fișierului).

În cazul în care, fișierul ce urmează să fie salvat, există deja la calea data, atunci, în cazul în care parametrul shouldUpdate este setat cu valoarea true, se vă șterge fișierul deja existent iar în locul acestuia se vă scrie fișierul nou. De asemenea se vor actualiza și datele intrării din baza de date.

În caz contrar, se realizează salvarea fișierului în sistemul de fișiere la calea dată și salvarea entității în baza de date cu coloana „path” setată cu valoarea caii din sistem către acel fișier.

```
} else {
    IOUtils.copyLarge(file.getInputStream(), Files.newOutputStream(fileToSave.toPath()));
    saved = fileEntityRepository.save(fileEntity);
}

userService.recomputeUserStorage(userByEmail);
FileEntityDTO fileEntityDTO = fileEntityMapper.fileEntityToFileEntityDTO(saved);
template.convertAndSendToUser(userEmail, destination: "/queue/newFile", fileEntityDTO);
return fileEntityDTO;
```

Figura 3.3.58 - Metoda uploadNewFile din FileStoreService (partea finală).

La final, se recalculează capacitatea de stocare disponibilă utilizatorului după ce a fost salvat fișierul și se trimită un mesaj pe topicul „/queue/newFile” către utilizatorul care a inițiat cererea, prin Websocket, ce conține ca și mesaj detaliile noului fișier adăugat. Ulterior, metoda returnează un obiect de tipul FileEntityDTO cu informațiile actualizate despre respectivul fișier.

Este important de menționat, că salvarea fișierelor efective în sistemul de fișiere (adică pe disc) este considerată o bună practică în ceea ce privește performanța

operațiilor cu fișiere [24] și este motivul principal pentru care în dezvoltarea aplicației, fișierele nu au fost salvate în baza de date , ci calea din sistemul de fișiere către acestea.

```
public boolean checkIfExists(String filename, ArrayList<String> pathFromRoot) {  
    String userEmail = SecurityContextHolder.getContext()  
        .getAuthentication()  
        .getPrincipal().toString();  
  
    pathFromRoot.add(filename);  
    Path path = FileStoreUtils.computePathFromRoot(userEmail, pathFromRoot);  
  
    File fileToSave = path.toFile();  
  
    if (fileToSave.exists()) {  
        log.warn("File {} already exists for user {}", fileToSave, userEmail);  
        return true;  
    }  
    return false;  
}
```

Figura 3.3.59 - Metoda checkIfExists din FileStoreService.

Metoda checkIfExists are rolul de a verifica dacă fișierul cu numele primit ca și parametru în variabila „filename”, aflat la calea definită prin variabila „pathFromRoot”, există deja.

Este returnată valoarea booleană „true” dacă fișierul există sau „false” în caz contrar.

```
public Page<FileEntityDTO> getAllFilesForUser(String userEmail, String sortBy, int page, int size, boolean asc, ArrayList<String> pathFromRoot) {  
    UserEntity userByEmail = userService.getUserByEmail(userEmail);  
  
    PageRequest pageable;  
  
    if (asc) {  
        pageable = PageRequest.of(page, size, Sort.by(sortBy).ascending());  
    } else {  
        pageable = PageRequest.of(page, size, Sort.by(sortBy).descending());  
    }  
  
    DirectoryEntity directoryEntity = directoryService.getDirectoryEntityFromNameAndUser(userByEmail, pathFromRoot)  
        .orElseThrow(); //todo handle  
        //todo there is no root dir saved in the db  
    Page<FileEntity> byUser = fileEntityRepository.findByDirectoryAndUser(directoryEntity, userEmail, pageable);  
  
    return byUser.map(FileEntityDTO::fromEntity);  
}
```

Figura 3.3.60 - Metoda getAllFilesForUser din FileStoreService.

Metoda getAllFilesForUser, are următorii parametrii : userEmail – adresa de email a utilizatorului pentru care să se returneze datele despre fișiere, sortBy – proprietatea după care să se realizeze sortarea datelor, page – numărul paginii, size – cantitatea de elemente de pe o pagină, asc – definește ordonarea elementelor și pathFromRoot – calea relativă la directorul de bază.

Această metodă inițial definește un obiect de tipul PageRequest. Acest obiect este pus la dispoziție de Spring Data JPA și este utilizat în definirea caracteristicilor elementelor dintr-o pagină.

O pagină reprezintă o listă de elemente dintr-un tabel care are o anumită ordonare, sortare și mărime. În momentul în care se trimit aceste PageRequest către un Repository, acesta va returna un obiect de tip Page, care conține elementele cu structura specificată în PageRequest.

Metoda această este folosită pentru afișarea inițială a datelor despre fișiere pe pagină, dar și în momentul în care sunt aplicate filtre.

```
public boolean deleteFileByIdAndUser(Long id, UserEntity user) {
    FileEntity fileByIdAndUser = this.getFileEntityByIdAndUser(id, user);
    File fileById = getFileById(id);
    fileEntityRepository.delete(fileByIdAndUser);

    return deleteFileFromFS(fileById);
}
```

Figura 3.3.62 - Metoda deleteFileByIdAndUser din FileStoreService.

Această metodă primește ca și parametru id-ul fișierului și entitatea corespunzătoare utilizatorului care deține acel fișier.

Rolul metodei este inițial, de a șterge entitatea referitoare la respectivul fișier, iar mai apoi să face ștergerea efectiva de pe sistemul de fișiere, returnând true dacă operația a reușit și false în caz contrar.

```
public SystemInfoDTO getSystemInfo() {

    File file = new File(FileStoreUtils.getBaseDir());
    long totalSpace = file.getTotalSpace();
    long usableSpace = file.getUsableSpace();
    long totalAssignedSpace = userService.getTotalAssignedSpace();
    usableSpace = usableSpace - totalAssignedSpace;

    return SystemInfoDTO.builder()
        .totalSpace(totalSpace)
        .usableSpace(usableSpace)
        .build();
}
```

Figura 3.3.63 - Metoda getSystemInfo din FileStoreService.

Această metodă are rolul de a returna un obiect de tipul SystemInfoDTO care conține capacitatea totală de spatiu de pe sistem și capacitatea totală utilizabilă de pe sistem.

 DirectoryService	
 directoryRepository	DirectoryRepository
 userManager	UserMapper
 userService	UserService
 directoryEntityMapper	DirectoryEntityMapper
 getAllDirectoriesInPath(ArrayList<String>)	DirectoriesWithParentDTO
 deleteDirById(Long)	DirectoryDTO
 getDirectoryEntityFromNameAndUser(UserEntity, ArrayList<String>)	Optional<DirectoryEntity>
 createInitialDirectoryEntity(Path, UserEntity)	void
 createDirectory(List<String>)	DirectoryDTO
 deleteUserById(Long)	UserDTO

Figura 3.3.64 – Serviciul DirectoryService cu toate dependințele și metodele acestuia.

Acest serviciu are următoarele dependințe : DirectoryRepository – Repository-ul asociat din Spring Data, UserMapper, DirectoryEntityMapper – beanuri care au rolul de a converti un obiect din entitate în DTO și invers.

Serviciul se ocupă cu crearea, citirea și ștergerea efectivă a directoarelor din cadrul aplicației.

Printre cele mai importante metode din acest serviciu sunt : createDirectory, getAllDirectoriesInPath și deleteDirById.

```
public DirectoryDTO createDirectory(List<String> pathsFromRoot) {
    String userEmail = SecurityContextHolder.getContext().getAuthentication().getPrincipal().toString();
    UserEntity userByEmail = userService.getUserByEmail(userEmail);

    Path directoryPath = FileStoreUtils.computePathFromRoot(userEmail, pathsFromRoot);

    DirectoryEntity createdDirectory = null;
    try {
        if (directoryPath.toFile().exists()){
            throw new IllegalStateException("Dir already exists!");
        }
        FileUtils.forceMkdir(directoryPath.toFile());
        if (!pathsFromRoot.get(pathsFromRoot.size() - 1).equals(userEmail)) {
            DirectoryEntity directoryEntity = new DirectoryEntity(directoryPath.toString(), pathsFromRoot.get(pathsFromRoot.size() - 1), userByEmail);
            createdDirectory = directoryRepository.save(directoryEntity);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return directoryEntityMapper.directoryEntityToDirectoryDTO(createdDirectory);
}
```

Figura 3.3.65 - Metoda createDirectory din FileStoreService.

Metoda createDirectory primește ca și parametru calea relativă la directorul de baza la care trebuie să se creeze directorul. Inițial se verifică dacă există deja un director cu același nume la aceeași cale, caz în care se aruncă o excepție.

Ulterior, se crează directorul pe sistem și se salvează entitatea în baza de date, după care se mapează la DTO și este returnată.

```
public DirectoriesWithParentDTO getAllDirectoriesInPath(ArrayList<String> pathsFromRoot) {
    String userEmail = SecurityContextHolder.getContext().getAuthentication().getPrincipal().toString();
    UserEntity userByEmail = userService.getUserByEmail(userEmail);

    Path directoryPath = FileStoreUtils.computePathFromRoot(userEmail, pathsFromRoot);
    List<DirectoryEntity> byFiles_path = directoryRepository.findByPathContainsAndUser(directoryPath.toString(), userByEmail);
    List<DirectoryEntity> correctDirs = new ArrayList<>();

    try (Stream<Path> stream = Files.list(directoryPath)) {
        Set<String> collect = stream
            .filter(Files::isDirectory)
            .map(Path::toString)
            .collect(Collectors.toSet());

        for (DirectoryEntity directoryEntity : byFiles_path) {
            for (String correctPath : collect) {
                if (directoryEntity.getPath().equals(correctPath)) {
                    correctDirs.add(directoryEntity);
                }
            }
        }
    }

    } catch (IOException e) {...}
    DirectoryEntity parentEntity = null;
    if (!CollectionUtils.isEmpty(pathsFromRoot)) {...}

    List<DirectoryDTO> directoryDTOS = directoryEntityMapper.entityListToDtoList(correctDirs);
    return DirectoriesWithParentDTO.builder()
        .directories(directoryDTOS)
        .parent(directoryEntityMapper.directoryEntityToDirectoryDTO(parentEntity))
        .build();
}
```

Figura 3.3.65 - Metoda getAllDirectoriesInPath din FileStoreService.

Această metodă primește ca și parametru calea către un director, relativă la directorul de bază. Inițial este interogată baza de date prin intermediul DirectoryRepository cu scopul de a primii o listă cu toate directoarele care conțin calea primită ca și parametru. Aceste directoare sunt mai apoi parcuse și verificate cu ce există pe sistemul de fișiere, după care se adaugă într-o listă.

La final, se caută și directorul părinte, și se adaugă prin intermediul unui builder la un obiect de tip DirectoriesWithParentDTO, care constituie rezultatul final.

```
public DirectoryDTO deleteDirById(Long id) throws IOException {
    DirectoryEntity directoryEntity = directoryRepository.findById(id)
        .orElseThrow(() -> new DirectoryNotFoundException("Dir not found for deletion"));

    UserEntity user = directoryEntity.getUser();
    File directoryToDelete = Path.of(directoryEntity.getPath()).toFile();
    FileUtils.deleteDirectory(directoryToDelete);
    DirectoryDTO directoryDTO = directoryEntityMapper.directoryEntityToDirectoryDTO(directoryEntity);

    directoryRepository.delete(directoryEntity);

    userService.recomputeUserStorage(user);
    return directoryDTO;
}
```

Figura 3.3.66 - Metoda deleteDirById din FileStoreService.

Această metodă primește ca și parametru id-ul directorului ce urmează să fie șters. Inițial se verifică dacă respectivul director există, după care este șters de pe sistem iar mai apoi este șters și din baza de date. Atât ștergerea din sistem, cât și ștergerea din baza de date produc că și efect ștergerea tuturor fișierelor din interiorul respectivului director.

La final se recalculează spațiul utilizat de către utilizator și se returnează un DTO care conține informațiile entității șterse.

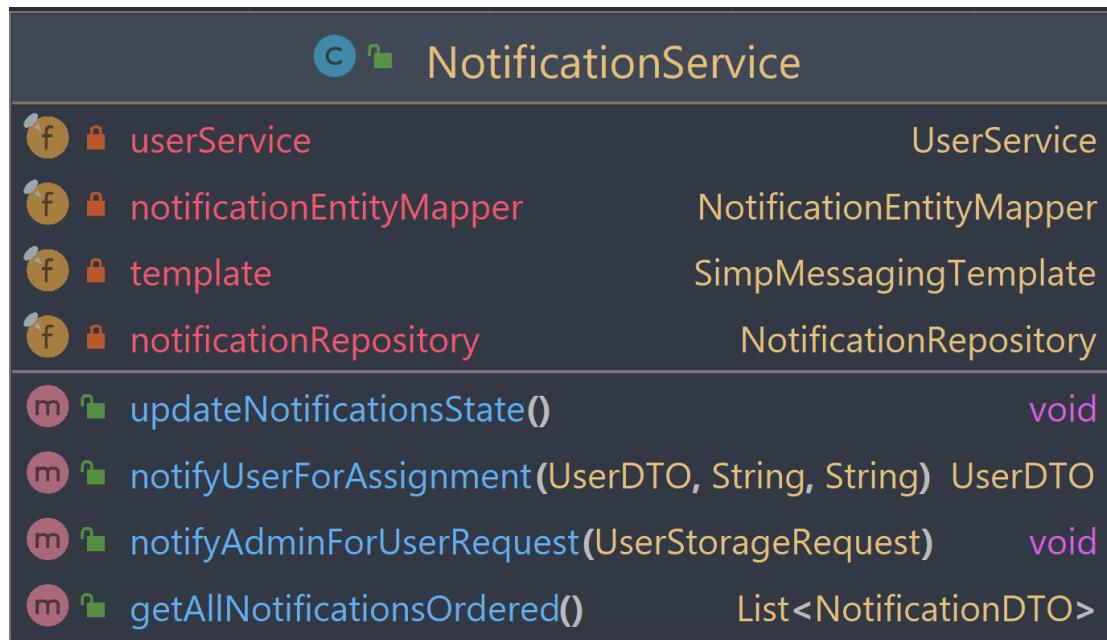


Figura 3.3.67 – Dependințele și metodele prezente în serviciul Notification Service.

Acest serviciu are ca și scop crearea, citirea și modificarea stării notificărilor unui anumit utilizator, dar și transmiterea de mesaje prin intermediul Websockets.

Metodele cele mai importante sunt : `notifyUserForAssignment` și `notifyAdminForUserRequest`.

```
public UserDTO notifyUserForAssignment(UserDTO userDTO, String amount, String description) {  
  
    UserEntity user = userService.getUserByEmail(userDTO.getEmail());  
  
    NotificationEntity notificationEntity = new NotificationEntity( description: ADMIN_EMAIL+":"+description + ":" +  
        amount, user);  
    NotificationEntity save = notificationRepository.save(notificationEntity);  
  
    NotificationDTO notificationDTO = notificationEntityMapper.entityToDTO(save);  
  
    template.convertAndSendToUser(userDTO.getEmail(), destination: "/queue/notify", notificationDTO);  
  
    return userDTO;  
}
```

Figura 3.3.68 - Metoda `notifyUserForAssignment` din `NotificationService`.

Această metodă primește ca și parametrii detaliile ce au fost trimise de către administrator către utilizator referitoare la atribuirea de spațiu de stocare, pe care mai apoi le salvează în baza de date.

La final se mapează entitatea rezultată într-un DTO și este trimisă utilizatorului prin Websocket.

```
public void notifyAdminForUserRequest(UserStorageRequest userStorageRequest) {
    String userEmail = SecurityContextHolder.getContext().getAuthentication().getPrincipal().toString();
    UserEntity sender = userService.getUserByEmail(userEmail);
    UserEntity admin = userService.getUserByEmail(ADMIN_EMAIL);

    NotificationEntity notificationEntity = new NotificationEntity( description: sender.getEmail() + ":" + userStorageRequest.description() + ":" + userStorageRequest.preferredAmount(),
        admin);
    NotificationEntity save = notificationRepository.save(notificationEntity);

    NotificationDTO notificationDTO = notificationEntityMapper.entityToDTO(save);

    template.convertAndSendToUser(ADMIN_EMAIL, destination: "/queue/notify", notificationDTO);
}
```

Figura 3.3.69 - Metoda notifyAdminForUserRequest din NotificatonService.

Această metodă are rolul de a salva o notificare trimisă de către un utilizator către un administrator, referitoare la cererea de atribuire de spatiu, în baza de date.

Ulterior, această notificare este trimisă administratorului prin Websocket.

3.3.5 Repositories

Repository-urile prezente în aplicația de backend sunt :UserRepository, FileEntityRepository, DirectoryRepository și Notification repository.

Fiecare serviciu își folosește repository-ul asociat pentru a prelua date de la baza de date PostgreSQL.

```
@Repository
public interface UserRepository extends JpaRepository<UserEntity, Long> {
    3 usages  ↗ mariannpm
    Optional<UserEntity> findByEmail(String email);
    1 usage  ↗ Petrica Marian
    boolean existsByEmail(String email);
}
```

Figura 3.3.70 – Captura de ecran cu UserRepository și metodele sale.

```
public interface FileEntityRepository extends JpaRepository<FileEntity, Long> {
    1 usage  ↳ marianpmd
    Page<FileEntity> findByUser(UserEntity user, Pageable pageable);
    1 usage  ↳ Petrica Marian
    Page<FileEntity> findByDirectoryAndUser(DirectoryEntity directory, UserEntity user, Pageable pageable);
    2 usages  ↳ marianpmd
    Optional<FileEntity> findByIdAndUser(Long id, UserEntity user);
    1 usage  ↳ marianpmd
    Optional<FileEntity> findByName(String name);
    1 usage  ↳ marianpmd
    List<FileEntity> findByUserAndNameContaining(UserEntity user, String name);
}
```

Figura 3.3.71 – Captura de ecran cu FileEntityRepository și metodele sale.

```
@Repository
public interface DirectoryRepository extends JpaRepository<DirectoryEntity, Long> {
    1 usage  ↳ Petrica Marian
    List<DirectoryEntity> findByUser(UserEntity user);
    2 usages  ↳ Petrica Marian
    Optional<DirectoryEntity> findByPath(String path);
    1 usage  ↳ marianpmd
    List<DirectoryEntity> findByPathContainsAndUser(String path, UserEntity user);
```

Figura 3.3.72 – Captura de ecran cu DirectoryRepository și metodele sale.

```
@Repository
public interface NotificationRepository extends JpaRepository<NotificationEntity, Long> {
    2 usages  ↳ Petrica Marian
    List<NotificationEntity> findByUserEntityOrderByNotificationStateDescDateDesc(UserEntity userEntity);
```

Figura 3.3.73 – Captura de ecran cu NotificationRepository și metodele sale.

3.4 Arhitectura UI-ului

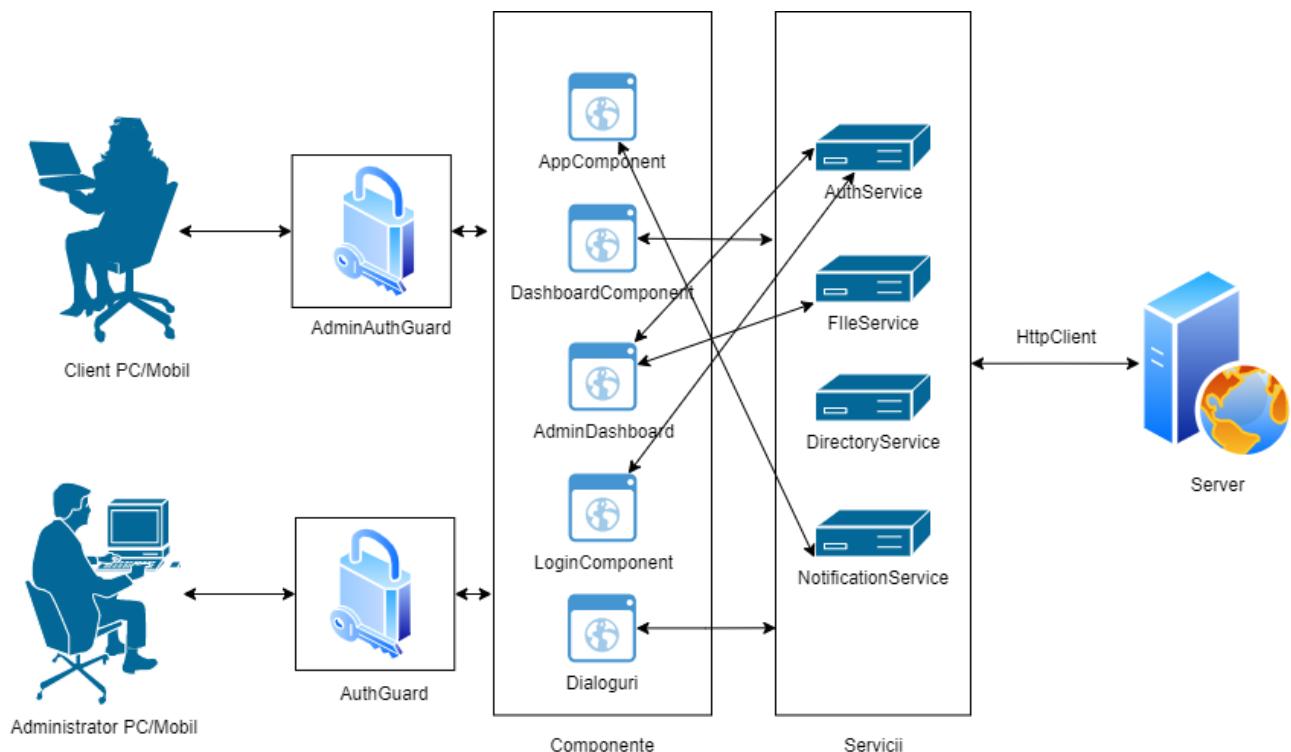


Figura 3.4.1 – Arhitectura de baza a UI-ului (Angular).

Angular este renumit pentru faptul că impune o structură rigidă și niște metodologii de dezvoltare bine puse la punct. Aplicația FileStorm se folosește la rândul ei de această structură pentru dezvoltarea interfeței de utilizator. Această structură are la bază componente, care reprezintă unitățile de bază din care este compusă interfața. Acestea pot fi imbricate, folosite că și view întreg (adică o pagină de sine stătătoare), refolosite și multe alte atrbute.

Componentele din care este compusă aplicația FileStorm sunt :

AppComponent – componenta de bază pe care sunt randate restul componentelor

DashboardComponent – view-ul principal pe care apar iconițele cu fișierele, directoarele etc.

LoginComponent – view-ul care conține formularul de logare

AdminDashboard – componenta care conține informațiile de sistem, accesibilă doar de către administrator

PublicResourceComponent – această componentă poate fi accesată resursele publice, pentru a le descărca fară să fie nevoie ca utilizatorul să fie logat. Nu este crucială existența acestei componente, ci ea reprezintă mai mult o adiție pentru a ușura trimiterea fișierelor către alte persoane.

Dialoguri – acestea sunt niște ferestre modale utilizate în comunicarea cu utilizatorul și diferite prompt-uri. Printre cele mai importante se enumerează: RegisterDialogComponent, FileItemDialogComponent, UploadLodingDialogComponent, FileUploadDialogComponent, etc.

3.4.1 Routes

Angular Routing este o dependință a framework-ului Angular care permite asocierea unei rute din bara de adresă a browserului cu un component care să acopere întreg view-ul sau cu mai multe componente care reprezintă child-urile pentru componentul principal.

```
const routes: Routes = [
  {
    path : "login",
    component : LoginComponent
  },
  {
    path : "dashboard",
    component : DashboardComponent,
    canActivate: [AuthGuardService],
    children : [
      {
        path:"admin",
        component : AdminDashboardComponent,
        canActivate:[AdminGuard]
      }
    ],
    {
      path : "",
      redirectTo : "dashboard",
      pathMatch : "full"
    }
  ],
  @NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
})
export class AppRoutingModule { }
```

Figura 3.4.2 – Configurarea rutelor din fișierul app-routing.module.ts.

Astfel, în urma înregistrării rezultă următoarele asocieri adresă-componentă :

„mariancr.go.ro/” -> „mariancr.go.ro.dashboard” -> DashboardComponent

„mariancr.go.ro/login” -> LoginComponent

„mariancr.go.ro/dashboard/admin” -> DashboardComponent este ascuns și se afisează AdminDashboardComponent.

3.4.2 Guards

Angular Guards reprezintă niște servicii speciale, care au rolul de a restricționa accesul asupra unor view-uri după anumite criterii definite anterior. În cadrul FileStorm există 2 astfel de guards, care se disting de restul claselor din typescript prin interfață CanActivate.

```

@Injectable({
  providedIn: 'root'
})
export class AuthGuardService implements CanActivate {
  constructor(private cookieService:CookieService,
    private router : Router) {
  }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot)
  : Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    let token = this.cookieService.get("app-jwt")

    if (!token){
      this.router.navigate( commands: ['login']);
      return false;
    }

    return true;
  }
}

```

Figura 3.4.3 – Definiția guardului AuthGuardService.

AuthGuardService este aplicat view-ului principal (Dashboard) și are rolul de a verifica dacă utilizatorul este logat. Inițial, se preia din Cookie-uri JWT-ul generat de aplicația de backend după cererea către /login. Dacă cookie-ul există, este permisă trecerea către view-ul principal iar în caz contrar se face rutarea către view-ul de logare.

```
export class AdminGuard implements CanActivate {

    constructor(private cookieService: CookieService,
                private router: Router) {...}

    canActivate(
        route: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean>

        let token = this.cookieService.get("app-jwt")

        if (!token) {
            this.router.navigate( commands: ['login']);
            return false;
        }

        let decodedToken = jwt_decode(token);
        console.log("DECODED TOKEN", decodedToken);
        // @ts-ignore
        console.log("DECODED TOKEN ROLE", decodedToken.roles);
        // @ts-ignore
        return decodedToken.roles[0] === 'admin';
    }
}
```

Figura 3.4.4 – Definiția guardului AdminGuard.

AdminGuard este aplicat componentei asociate căii /dashboard/admin, și are rolul de a verifica dacă utilizatorul logat este administrator. Similar cu AuthGuardService se preia JWT-ul, după care se decodează și se verifică ca rolurile să conțină token-ul „admin”.

3.4.3 Componete

Componentele notabile (datorită faptului că sunt și view-uri dar nu numai) ale acestei aplicații sunt : LoginComponent, DashboardComponent și AdminDashboardComponent.

```
<form [formGroup]="formGroup" class="uploadForm">

    <mat-form-field appearance="fill">
        <mat-label>Email</mat-label>
        <input matInput [formControl]="email" required>
        <mat-error *ngIf="email.invalid">Email is invalid!</mat-error>
    </mat-form-field>

    <mat-form-field appearance="fill">
        <mat-label>Password</mat-label>
        <input type="password" matInput [formControl]="password" required>
        <mat-error *ngIf="password.invalid">Password is required!</mat-error>
    </mat-form-field>

    <button [disabled]="this.formGroup.invalid" mat-raised-button...>

</form>
```

Figura 3.4.5 – Secvența de cod din login.component.html.

Acest component are rolul de a genera interfața de logare ce conține un formular cu două câmpuri : email și parola. Acest formular este referit în fișierul de typescript, iar la apăsarea butonului „Login”, este apelat serviciul de logare. În cazul în care răspunsul primit nu este are un status de eroare, se face rutarea la view-ul principal iar în caz contrar se afișează o notificare cu un mesaj de eroare.

```

onLogin() {
  this.isLoading = true;
  if (this.email.value && this.password.value) {
    let encodedPassword = btoa(this.password.value);

    this.auth.onLogin(this.email.value, encodedPassword)
      .subscribe( observer: {
        next: result => {
          this.isLoading = false;
          this.router.navigate( commands: [ '/dashboard' ]);
        },
        error: err => {
          console.log("Error , showing snackbar")
          this.snackBar.open( message: "Login has failed!", action: "Close",
            duration: 2000,
            panelClass: [ 'mat-toolbar', 'mat-warn' ]
          )
          this.isLoading = false;
        }
      })
  }
}

```

Figura 3.4.6 – Secvența de cod din login.component.ts.

View-ul Dashboard este cel principal, și compus din multiple secțiuni care de regula constituie alte componente din diverse biblioteci.

De menționat este faptul că bara de sus ce conține logo-ul aplicației, butonul pentru sidenav, butonul de notificări și linkul către github sunt din AppComponent.

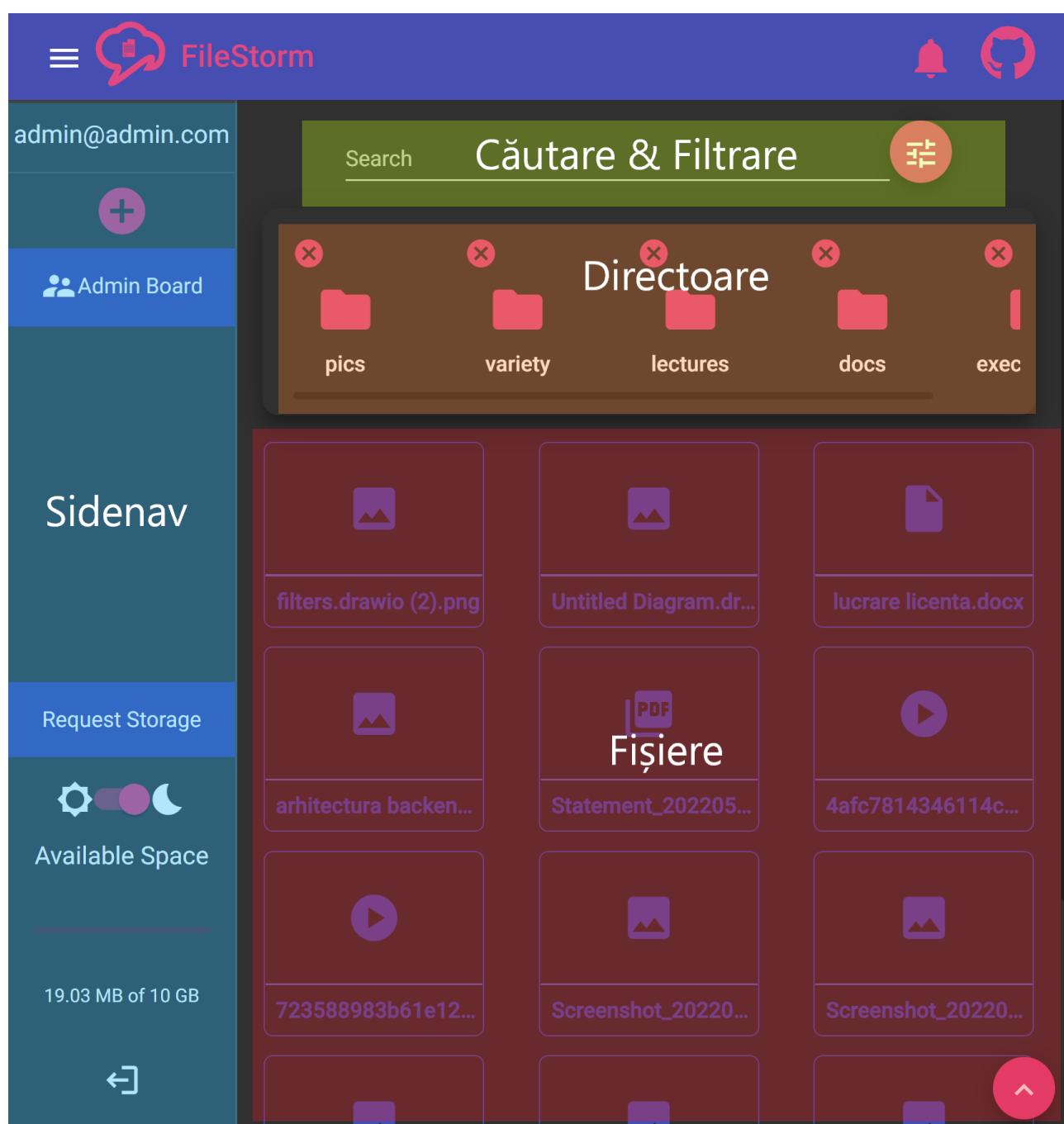


Figura 3.4.7 – Împărțirea pe secvențe a interfeței principale.

Secvențele în care se împarte interfața principală sunt urmatoarele :

- Sidenav - Aceasta este un component din Angular Material și este un meniu dispus pe partea stângă care poate fi închis sau deschis. Pe acesta se află :
 - Butonul de adăugare a fișierelor/directoarelor
 - Butonul de trecere la view-ul de administrator (doar dacă utilizatorul este logat cu un cont de administrator)
 - Butonul pentru dialogul de cerere pentru spațiu de stocare
 - Toggle button pentru mod întunecat
 - Bară care arată spațiul ocupat relativ la cel atribuit utilizatorului și textul aferent
 - Buton de logout.

```
<mat-sidenav class="sidenav" #snav [mode]="mobileQuery.matches ? 'over' : 'side'>
  <mat-nav-list class="nav-list">
    <mat-list-item *ngIf="loggedUser" [disableRipple]=true>{{loggedUser.email}}
      <mat-divider></mat-divider>
    <mat-list-item [disableRipple]=true matRipple class="add-button" [mat-menu-trigger]>
      <mat-menu #menu xPosition="before">
        <button (click)="uploaderAction(uploader)" mat-menu-item>
          <mat-icon>upload_file</mat-icon>
          File
        </button>
        <button mat-menu-item (click)="addDirectory()">
          <mat-icon>create_new_folder</mat-icon>
          Directory
        </button>
      </mat-menu>
      <mat-icon color="accent">add_circle</mat-icon>
    <input
      hidden
      type="file"
      multiple
      #uploader
      (change)="onUploadClick($event)"
    />
  </mat-list-item>
```

Figura 3.4.8 – Secvența de cod pentru sidenav din dashboard.component.html.

- Căutare si Filtrare - această zona cuprinde bara de căutare și butonul pentru selectare a filtrelor.

```
<div [class.show]="isLoading" class="filtering-section">

  <mat-form-field>
    <input #auto class="search-bar" matInput placeholder="Search" aria-label="State"
      [FormControl]="searchFileControl">
  </mat-form-field>

  <button class="tune-button" mat-mini-fab (click)="openFilters()">
    <mat-icon>tune</mat-icon>
  </button>

</div>
```

Figura 3.4.9 – Secvența de cod din zona de căutare și filtrare.

```
ngAfterViewInit() {
  this.searchFileControl.valueChanges
    .pipe(
      debounceTime( dueTime: 500),
      tap( next: () => {
        this.isLoading = true;
      }),
      switchMap(...)
    )
    .subscribe( next: data => {
      if (data.length == 0) {
        this.files = [];
      } else {
        data.forEach(fdata => {...})
        this.files = data;
      }
    });
}
```

Figura 3.4.10 – Secvența cu codul ce realizează apelul către serviciul cu keyword-ul necesar pentru căutare.

Input-ul care realizează căutarea, are definit un eveniment care este declanșat de fiecare dată când își schimbă valoarea, după o perioadă de 500 de milisecunde. Acest eveniment face apelul către serviciu și afișează în liste de fișiere, doar cele care se potrivesc cu keyword-ul dat.

```
openFilters() {
  let matDialogRef = this.dialog.open(FiltersDialogComponent, { config: {
    data: {
      sortBy: this.sortBy,
      asc: this.asc
    },
    autoFocus: false
  });

  matDialogRef.afterClosed()
    .subscribe(next: Result => {
      console.log("Chosen result : ", result);
      if (!result) return;
      this.sortBy = result.sortBy;
      this.asc = result.asc;

      this.currentPage = 0; //reset
      this.files = [];

      this.loadAllInitialFilesPaginated(this.sortBy, this.currentPage, { size: 100, this.asc, this.currentPaths })
    })
}
}
```

Figura 3.4.11 – Captura de ecran cu funcția openFilters.

Funcția openFilters deschide un dialog în care se află configurările pentru filtrare. După ce utilizatorul alege o opțiune de filtreare, este apelat serviciul care va reîncărca toate fișierele din pagină după opțiunile de filtrare date.

- Zona de directoare : Această zonă afisează o iconiță cu un director și este modificată dinamic de fiecare dată când se crează sau se șterge un director.

```
<section class="directory-section mat-elevation-z10" *ngIf="!isLoading">
  <div class="directory-div">
    <p...>
    <div matRipple class="dir-item"...>

      <div class="dir-item" *ngFor="let dir of directories">
        <button mat-button (click)="onDirDeleteClick(dir)" class="dir-delete-button"...>
        <button mat-button...>
      </div>
    </div>
  </section>
```

Figura 3.4.12 – Secvența de cod cu zona de directoare din fișierul html.

```
onDirDeleteClick(dir: DirectoryInfo) {
  let matDialogRef = this.dialog.open(DirectoryDeleteDialogComponent, {data: dir...});

  matDialogRef.afterClosed()
    .subscribe(next: response => {
      if (response) {
        this.directoryService.deleteDirectory(dir.id)
          .subscribe(next: deleted => {
            this.directories = this.directories.filter(dir => dir.id !== deleted.id);
            this.initUserInfo(this.userEmail);
          });
      }
    });
}
```

Figura 3.4.13 – Funcția onDirDeleteClick.

Funcția de ștergere a unui director este apelată după apăsarea butonului cu simbolul „x” din stângă directorului. Aceasta deschide întăi un dialog în care utilizatorul este întrebat dacă dorește să continuie și în cazul în care se dorește, este apelat serviciul care vă șterge acel director.

- Zona de fișiere : În cadrul acestei zone sunt dispuse chenarele care arată informații despre fiecare fișier în parte. De menționat este faptul că această zonă este și un container de drag & drop, adică se pot lua fișiere dintr-un folder de pe sistem, după ce poate face hover pe această zonă iar la lăsarea click-ului se va realiza procesul de adăugare a fișierelor. Aceste chenare conțin o iconiță reprezentativă pentru tipul de fișier, și numele fișierului, iar zona aceasta este încărcată în momentul în care este deschisă pagina principală, dar și în momentul în care se fac schimbări la nivelul listei de fișiere.

```
private loadAllInitialFilesPaginated(sortBy: string, page: number, size: number, asc: boolean, currentPaths: string[])
  this.fileService.loadAllFiles(sortBy, page, size, asc, currentPaths)
    .subscribe(next: fileData => {
      fileData.content.forEach(fdata => {...})
      this.files = fileData.content;

      if (!fileData.last)
        this.currentPage = fileData.pageable.pageNumber + 1;
      this.isRequestMade = true;
      this.isLoading = false;
    })
}
```

Figura 3.4.14 – Funcția loadAllInitialFilesPaginated.

Funcția loadAllInitialFilesPaginated este cea care realizează apelul către serviciu pentru popularea inițială a listei de fișiere ordonate după ultima modificare.

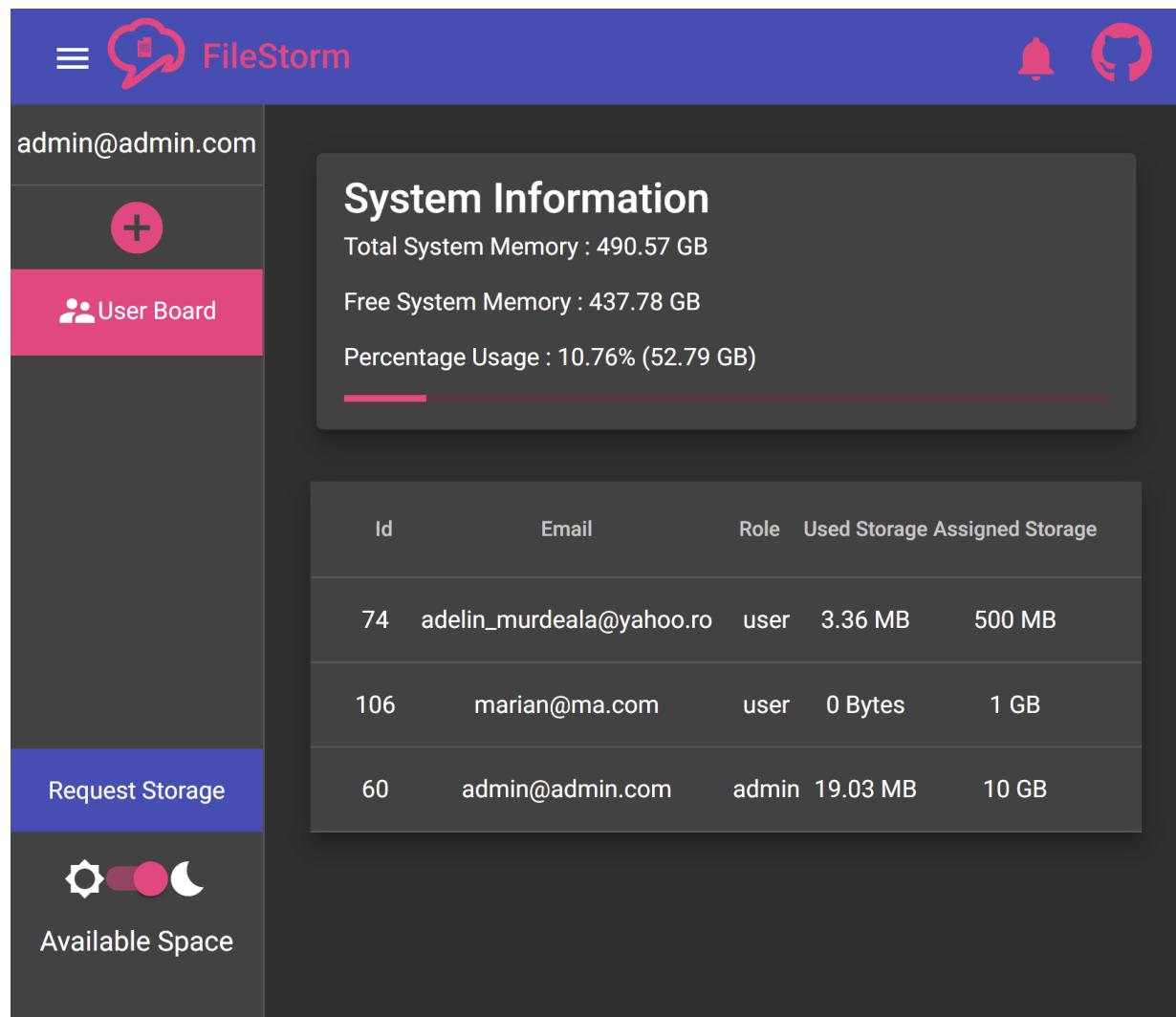


Figura 3.4.15 – View-ul componentei AdminDashboard.

Ultimul view, acela de dashboard administrativ este accesibil doar de către utilizatorii logați cu un cont de administrator și conține 2 secțiuni : informațiile de sistem și informațiile despre utilizatori. De asemenea se poate observă că un administrator se poate întoarce la dashboard-ul normal apăsând butonul User Board care acum are altă culoare în sidenav.

```
<mat-card class="mat-elevation-z10 system-mem-div">
  <mat-card-title>System Information</mat-card-title>
  <p>Total System Memory : {{computeSize(sysTotalSpace)}}</p>
  <p>Free System Memory : {{computeSize(sysUsableSpace)}}</p>
  <p>Percentage Usage : {{computeSysPercentage(sysTotalSpace,sysUsableSpace).toFixed( fractionDigits: 2) + '%' }}{{ ' (' +computeSize(sysTotalSpace) + ' ) / ' + computeSize(sysUsableSpace) + ' * 100' }}%</p>
  <mat-progress-bar color="accent" [value]="computeSysPercentage(sysTotalSpace,sysUsableSpace)"></mat-progress-bar>
</mat-card>
```

Figura 3.4.16 – Secvența de cod din prima secțiune.

În cadrul primei secțiuni este apelat serviciul pentru date referitoare la statusul sistemului și mai apoi afișate formatat în pagină.

```
<table mat-table  
      [dataSource]="tableDataSource" multiTemplateDataRows  
      class="mat-elevation-z10">  
  <ng-container ...>  
  
    ● <ng-container matColumnDef="expandedDetail" ...>  
  
      <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>  
      <tr mat-row *matRowDef="let element; columns: displayedColumns;"  
          class="example-element-row"  
          [class.example-expanded-row]="expandedElement === element"  
          (click)="expandedElement = expandedElement === element ? null : element">  
        </tr>  
      <tr mat-row *matRowDef="let row; columns: ['expandedDetail']" class="example-detail-row"></tr>  
    </table>
```

Figura 3.4.17 – Secvența de cod din cea de-a doua secțiune.

Cea de-a doua secțiune este un tabel în care apar datele despre toți utilizatorii platformei și de unde se pot efectua operațiile de atribuire de spațiu și ștergere.

3.4.4 Servicii

Serviciile, asemănător cu cele din Spring, se ocupă strict de logică și, în cazul Angular, de utilizarea unui client http pentru a putea accesa Web API-ul.

Aceste servicii sunt : AuthService, FileService, DirectoryService și NotificationService.

- AuthService – se ocupă cu cererile de logare și înregistrare.

```
onLogin(email: string, password: string) {  
  let params = new FormData();  
  params.append('name: "email", email);  
  params.append('name: "password", password);  
  
  return this.http.post<LoginData>(LOGIN_URL, params, {  
    observe: 'response'  
  });  
}
```

Figura 3.4.18 – Funcția onLogin care apelează endpoint-ul „user/login”.

- FileService – se ocupă de cererile de creare, ștergere, modificare și descărcare a fișierelor.

```
    return this.http.post(UPLOAD_URL, formData, {  
      reportProgress: true,  
      responseType: 'json',  
      observe: 'events',  
      headers: headers,  
      params : httpParams  
    });  
  }
```

Figura 3.4.19 – Secvența de cod din funcția uploadFile cu cererea catre endpoint-ul „file/upload”

- DirectoryService – se ocupă de operațiile cu directoare.

```
createDirectory(dirName:string,pathsFromRoot:string[]):  
  let fullPath = [...pathsFromRoot , dirName];  
  return this.http.post<DirectoryInfo>(CREATE_DIRECTORY, fullPath);  
}
```

Figura 3.4.20 – Funcția care apelează endpoint-ul „directory/create”.

- NotificationService – se ocupă de cererile referitoare la notificări.

```
getUserNotifications(){  
  return this.http.get<NotificationInfo[]>(ALL_NOTIFICATIONS);  
}
```

Figura 3.4.21 – Funcția care apelează enpoint-ul „/notification/all”.

3.5 Testarea performanței

Testarea performantei, în cazul platformei a fost realizată luând în considerare, timpul pe care utilizatorul trebuie să îl aștepte până când se încarcă un fișier de 1GB pe o platformă. Alte criterii ce pot fi luate în considerare sunt viteza de încărcare a paginilor sau timpul de răspuns pentru diferite acțiuni. Pentru acestea se poate afirma faptul că toate aplicațiile testate sunt aproximativ la fel în ceea ce privește acest aspect, inclusiv FileStorm. Din această cauză, am luat în considerare doar viteza de încărcare/descărcare deoarece aici este zona în care FileStorm se distinge de restul aplicațiilor testate.

Aplicațiile testate au fost : Google Drive, DropBox și OneDrive.

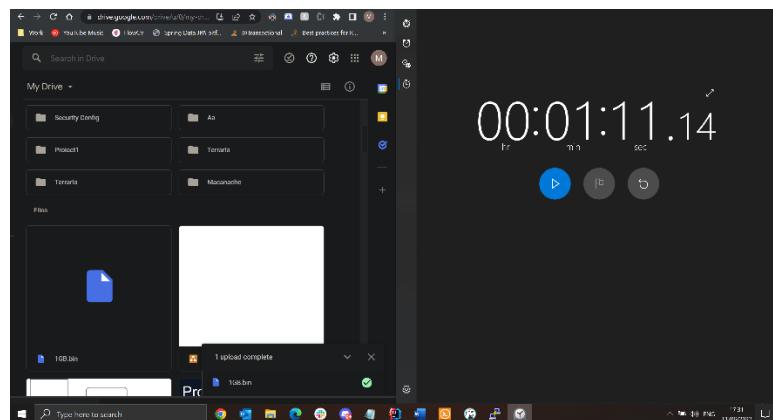


Figura 3.5.1 – Durata de încărcare a fișierului în GoogleDrive.

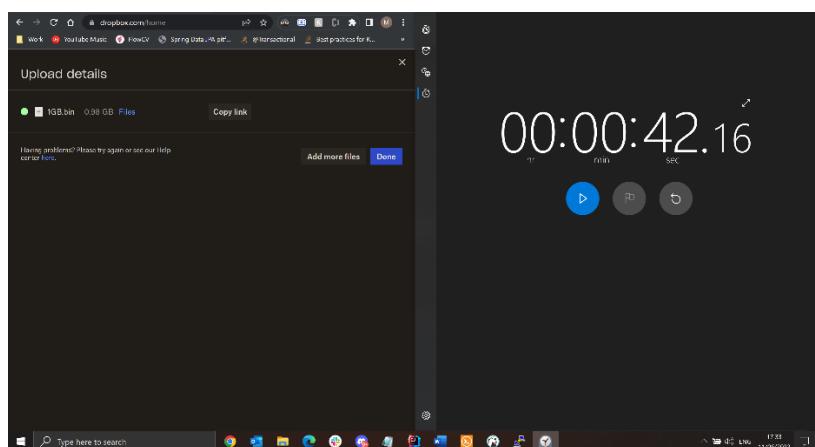


Figura 3.5.2 – Durata de încărcare a fișierului în Dropbox.

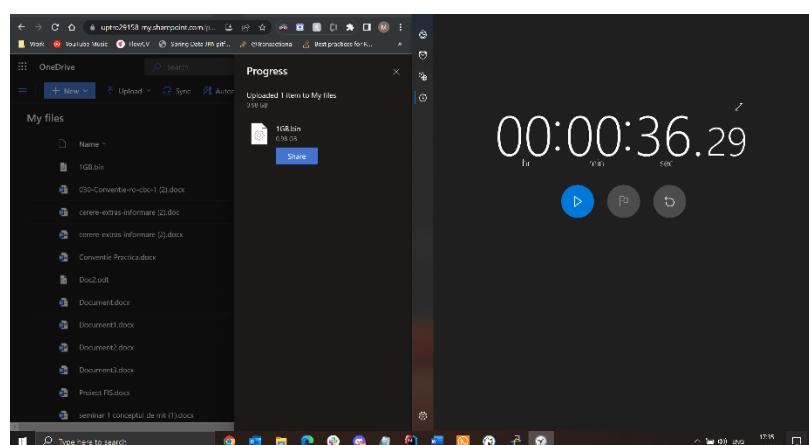


Figura 3.5.1 – Durata de încărcare a fișierului in OneDrive.

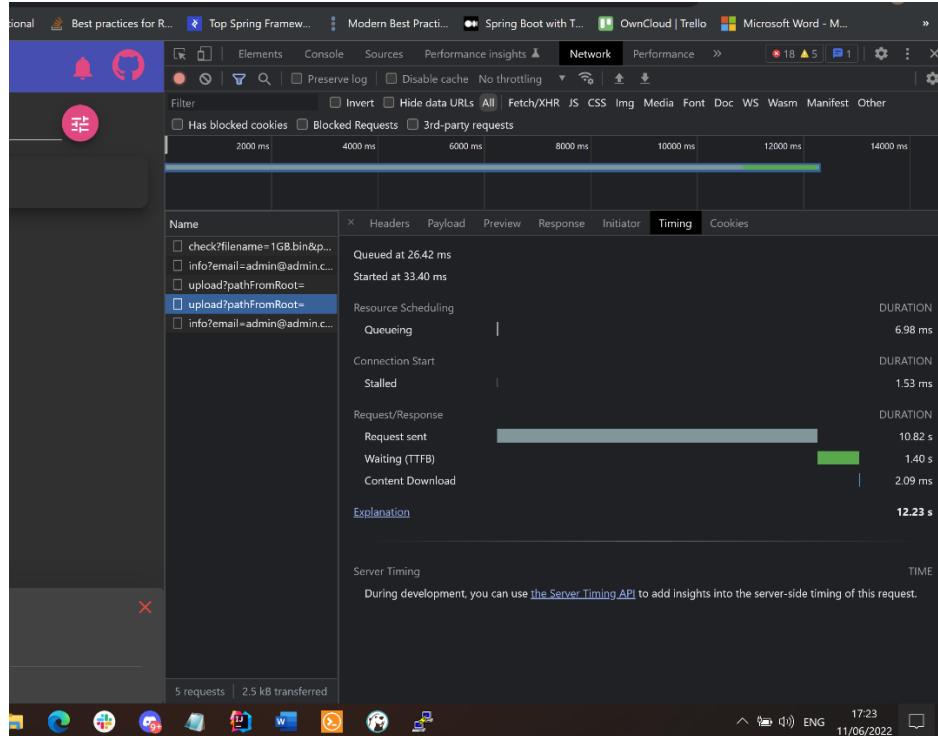


Figura 3.5.1 – Durata de încărcare a fișierului în FileStorm(12.23 secunde).

Se observă o diferență majoră între timpii de încărcare a fișierului de 1GB pe FileStorm și pe celelalte aplicații, fiind de 5 sau 4 ori mai rapid. Acest lucru se datorează faptului că FileStorm nu limitează viteza de download sau upload asupra resurselor la fel cum fac celelalte aplicații. Bineînțeles, aceste viteze depind de lățime de banda a utilizatorului (în acest exemplu am fost conectat la o rețea wi-fi ce fluctuează între 220 și 300 mbps), iar în urma testelor se observă faptul că la viteze de sub 30-45 mbps, vitezele nu diferă foarte mult.

Același efect se aplică și pentru vitezele de descărcare, pentru acest fișier de 1GB descărcarea din FileStorm a durat doar 4 – 5 secunde, iar restul aplicațiilor, în jur de 20-30 de secunde.

4. UTILIZAREA APLICAȚIEI

Acest capitol prezintă felul în care un potențial utilizator poate folosi această aplicație, indiferent dacă acesta dorește să fie un simplu client, sau o gazdă-administrator.

4.1 Necesități hardware și software

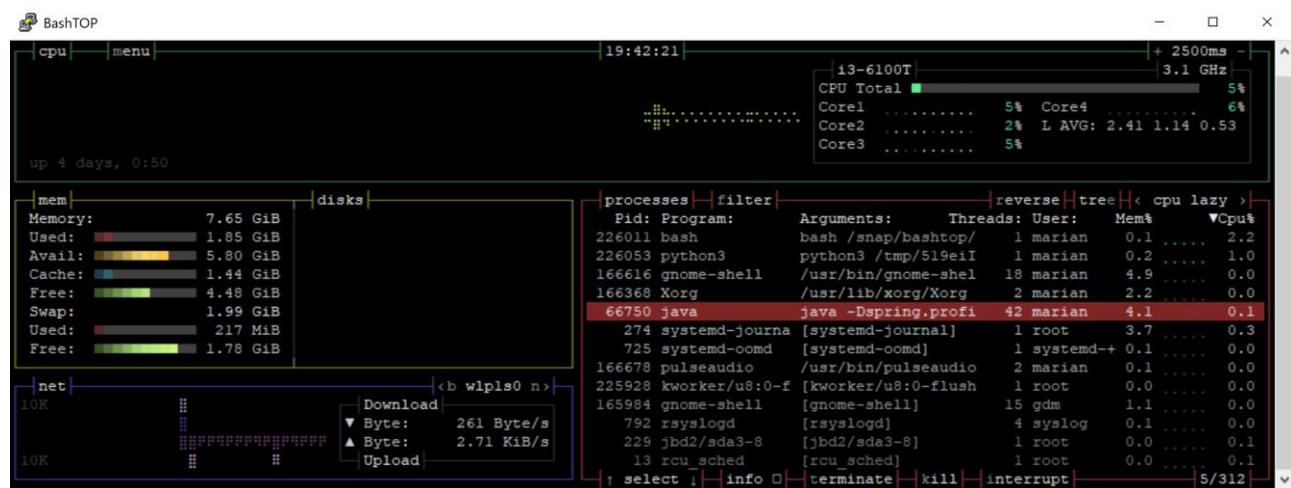


Figura 4.1 – Starea sistemului, afișată cu ajutorul aplicației bashtop.

FileStorm nu necesită un dispozitiv foarte performant pentru a rula, după cum se observă și în captura de mai sus, aplicația în stand-by(fără să realizeze vreo operație), consumă 4% din memoria RÂM și mai puțin de 1% din procesor.

Specificațiile dispozitivului pe care este găzduită aplicația sunt : Intel Core i3-6100T 3.20GHz, 8GB DDR4, 500GB SATA.

Astfel se poate observa că necesitățile hardware sunt minime, și aplicația poate funcționa pe orice dispozitiv mediocru.

Cât despre necesitățile software, aplicația a fost dezvoltată folosind java OpenJdk 17, iar în cazul în care utilizatorul dorește să găzduiască și aplicația de frontend, este necesar serverul nginx sau orice alt server de fisiere.

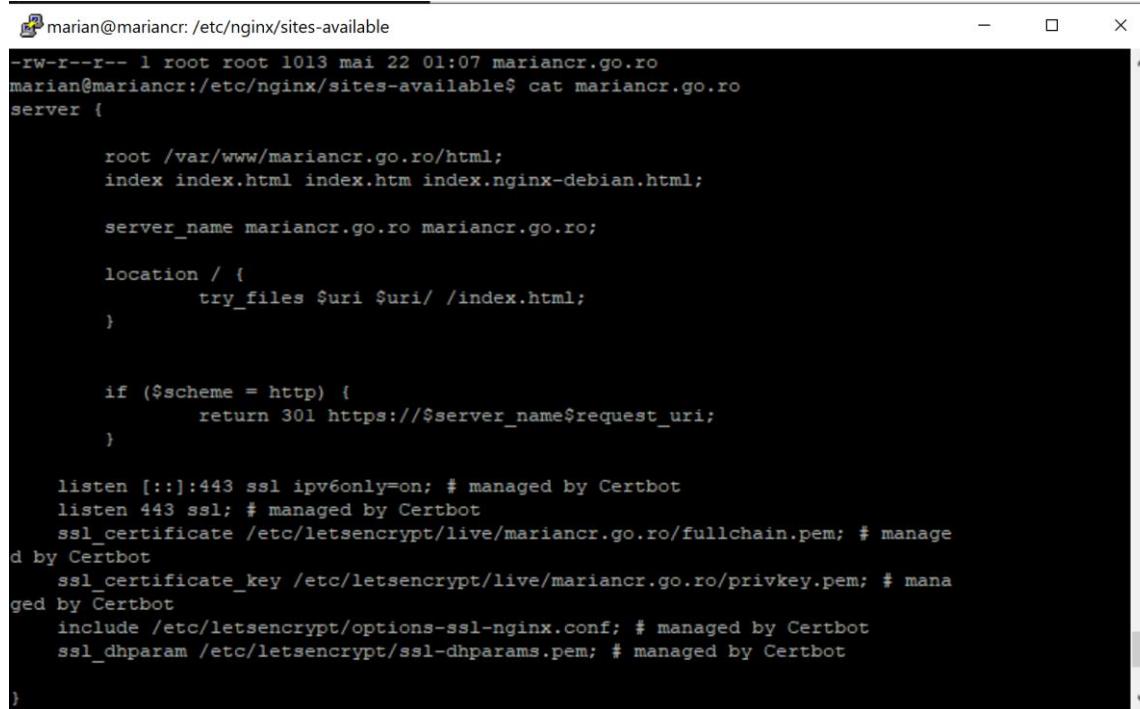
4.2 Instalare și găzduire

Pentru instalare este necesară construirea unei arhive jar pentru proiectul de backend, și de rularea procesului de build în proiectul de angular.

Astfel, arhiva jar se vă rula folosind linia de comandă în felul următor :
"nohup java -Dspring.profiles.active=prod -jar FileStorm-backend-0.0.1-SNAPSHOT.jar > log.txt 2>&1 &".

Această comandă vă porni un process imun la semnalele de hangup, pe care va rula aplicația de Spring, care își vă scrie log-urile într-un fișier log.txt.

În ceea ce privește aplicația de frontend, este necesar un server de fișiere precum nginx.



The screenshot shows a terminal window with the command `cat mariancr.go.ro` running. The output displays the Nginx configuration file for the domain mariancr.go.ro. The configuration includes a server block for the domain, setting the root directory to /var/www/mariancr.go.ro/html and specifying index files. It also includes SSL configuration for port 443, using certificates managed by Certbot. The configuration is well-structured with proper indentation and comments.

```
-rw-r--r-- 1 root root 1013 mai 22 01:07 mariancr.go.ro
mariancr@mariancr:/etc/nginx/sites-available$ cat mariancr.go.ro
server {
    root /var/www/mariancr.go.ro/html;
    index index.html index.htm index.nginx-debian.html;

    server_name mariancr.go.ro mariancr.go.ro;

    location / {
        try_files $uri $uri/ /index.html;
    }

    if ($scheme = http) {
        return 301 https://$server_name$request_uri;
    }
}

listen [::]:443 ssl ipv6only=on; # managed by Certbot
listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/mariancr.go.ro/fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/mariancr.go.ro/privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}
```

Figura 4.2 – Configurările pentru nginx.

FileStorm poate fi pornit fie local, folosind un server http mai simplu precum http-server, dar în acest caz este setat cu ajutorul nginx. Domeniul mariancr.go.ro este setat în hostfile, și pointează către adresa de ip privată a dispozitivului gazdă. Această configurare face în aşa fel în cât să înregistreze și niște certificate ssl pentru a face disponibilă găzduirea site-ului cu https.

4.3 Autentificarea & logarea

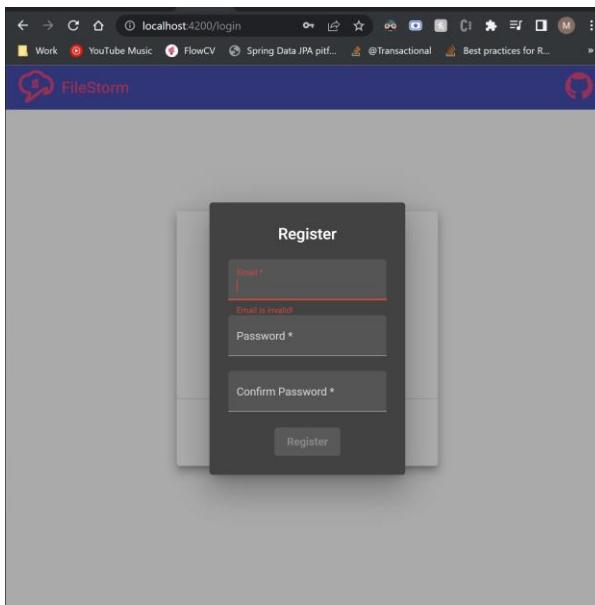


Figura 4.3 – Formularul de înregistrare.

Inițial, când utilizatorul accesează URL-ul : <https://mariancr.go.ro> , este direcționat către pagina de logare (daca nu este deja logat). Aici se regăsește și butonul de Register care deschide un formular.

După completarea formularului, dacă datele sunt valide și nu mai există deja, vă fi redirectionat direct către pagina principală.

Din pagina principală, apăsând butonul de logout amplasat în josul sidenav-ulu, se poate reveni la pagina de logare.

4.4 Cererea și atribuirea spațiului de stocare

Un utilizator proaspăt-inregistrat, nu are atribuit spatiu de stocare. Acesta masura a fost luata pentru a pastra managementul spatiului in mainile administratorului.

Pentru a face o cerere de atribuire, utilizatorul poate apăsa butonul de Request Storage aflat pe sidenav.

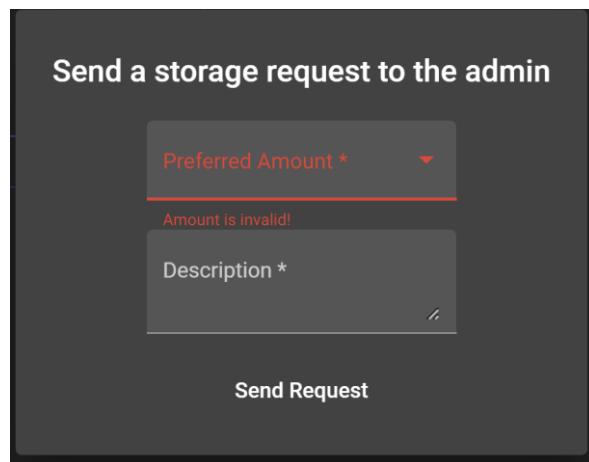


Figura 4.5 – Dialogul pentru cererea de stocare.

Se va deschide un dialog ce conține 2 câmpuri : Preferred Amount – cantitatea de stocare dorită de către utilizator și Description – un camp în care utilizatorul își poate motiva cererea.

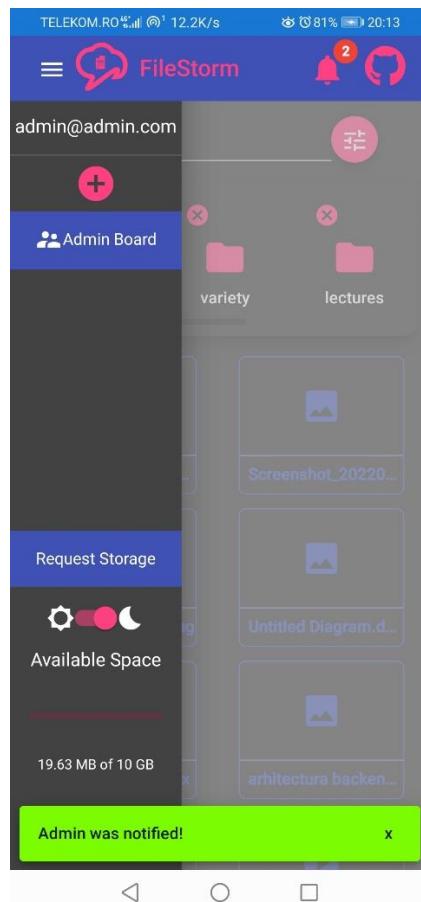


Figura 4.6 – Exemplu de trimitere de notificare.

Pe partea administratorului, acesta va primi o notificare cu aceste informații, și va putea să atribuie utilizatorului cantitatea dorită, apăsând pe butonul dedicat după ce selectează un utilizator din tabel.

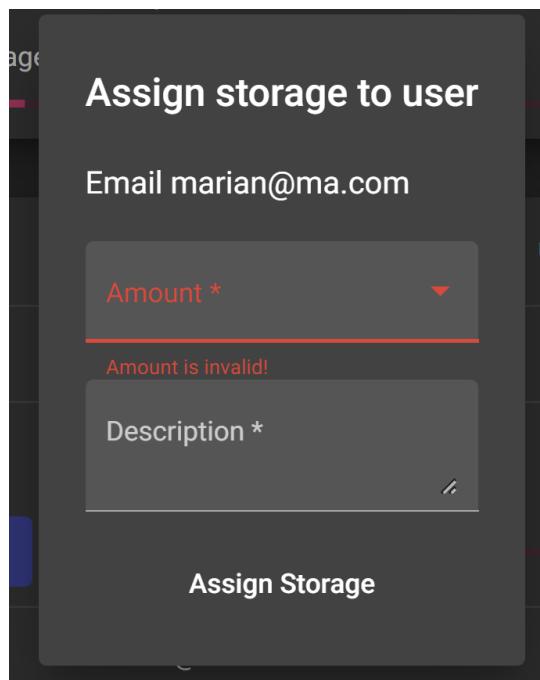


Figura 4.7 – Dialogul de pe partea administratorului.

Similar cu administratorul, utilizatorul va fi notificat despre faptul ca l-a fost atribuit spațiu și poate continua să folosească aplicația.

4.5 Operații CRUD cu fișiere

Adăugarea fișierelor se poate face apăsând butonul dedicat de pe sidenav, sau facandu-se drag & drop în zona de fișiere. După realizarea oricărei dintre aceste acțiuni, este deschis un dialog în care este afișată mărimea fișierului, ce are scop de confirmare. După confirmare, este deschis un alt dialog, în colțul din dreapta-jos al ecranului unde este arătat progresul.

După ce fișierul a fost adăugat cu succes, se poate face click pe acesta, iar în funcție de tipul de fișier, poate fi vizualizat sau nu. Tipurile de fișiere ce se pot vizualiza sunt : imagini, pdf-uri, videoclipuri.

Totodată, aceste fișiere pot fi descărcate sau șterse folosindu-se butoanele dedicate din colțul din dreapta-sus al ecranului.

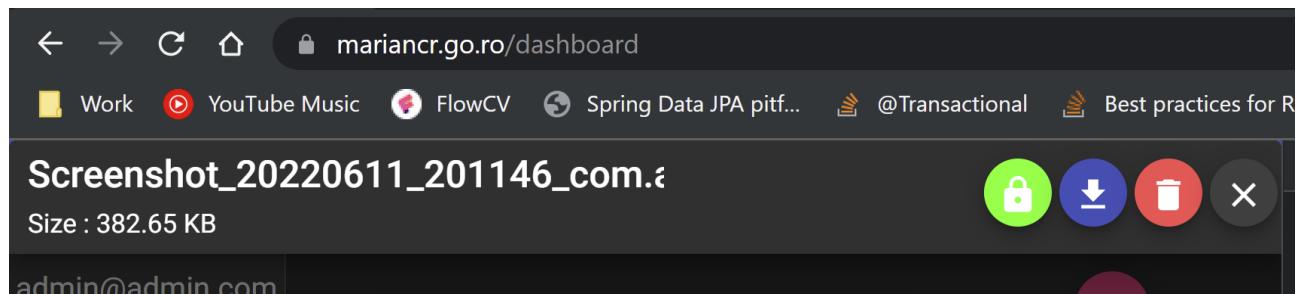


Figura 4.8 – Captura de ecran cu butoanele dedicate pentru descărcare, ștergere și vizibilitate.

De asemenea, zonei de fișiere i se pot aplica diverse filtre, utilizând dialogul deschis de butonul din dreapta barei de căutare. Bara de căutare modifică și ea zona de fișiere, în funcție de literele scrise și se updatează automat la 500 de milisecunde după apăsarea unei taste.

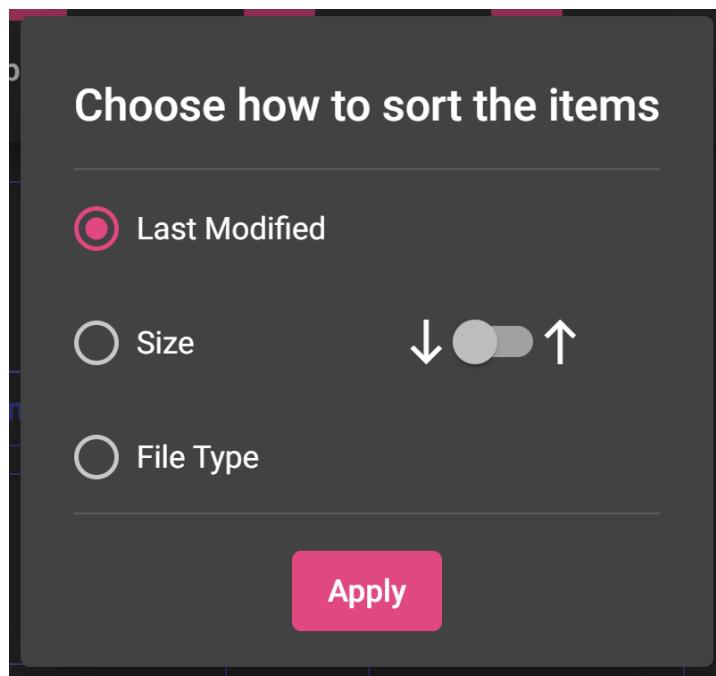


Figura 4.9 – Dialogul cu opțiunile de filtrare.

4.6 Operații cu directoare

Directoarele pot fi și ele create cu butonul „+” din sidenav, și sunt afișate într-o zonă dedicată sub bara de căutare și filtrare.

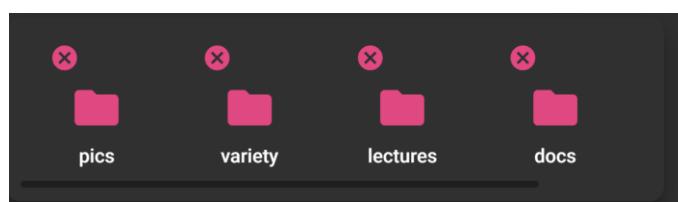


Figura 4.10 – Zona de directoare.

Fiecare iconiță de director are în colțul din stângă-sus un buton cu simbolul “x”, care va realiza ștergerea acelui director și a tuturor fișierelor din interiorul acestuia, după confirmarea unui alt dialog.

4.7 Operații cu utilizatori

Zona de dashboard administrativ este accesibilă doar de către utilizatorii conectați cu un cont de administrator, prin intermediul unui buton dedicat de pe sidenav denumit

Admin Board. În cadrul acestui view, administratorul poate observa starea de ocupare sistemului și poate atribui sau șterge utilizatorii aflați în tabel, bineînțeles după confirmarea unui dialog.

Id	Email	Role	Used Storage	Assigned Storage
74	adelin_murdeala@yahoo.ro	user	3.36 MB	500 MB
		1% Used		
106	mariann@ma.com	user	0 Bytes	1 GB

Figura 4.10 – Exemplu de utilizator aflat în tabel.

5. CONCLUZII

În concluzie, consider că aplicația dezvoltată, FileStorm reprezintă o platformă complexă, dificil de implementat, ce utilizează cele mai noi tehnologii din domeniu și cu toate acestea, ușor accesibilă.

În ceea ce privește restul serviciilor de acest fel, de obicei legate de o platformă Cloud, sunt toate foarte similare prin faptul că există o constrângere a spațiului și a lățimii de banda atribuite fiecărui utilizator. Adesea aceste constrângeri pot fi ridicate câteodată în urma plătirii unui cost sau a unui abonament la respectivele servicii.

FileStorm se prezintă ca fiind o opțiune pentru cei ce nu doresc să își împărtășească fișierele cu diverse corporații, doresc să își utilizeze propriul home-server pentru a beneficia de o viteză maximă de transfer sau doresc să readucă la viață dispozitive trecute de vreme, pentru a funcționa asemenea unui server dedicat acestei aplicații, ea având posibilitatea de a fi instalată pe aproape orice dispozitiv PC.

Totodată, datorită utilizării standardului PWA, care de abea în anul 2019 a început să fie luat în considerare ca și opțiune intermediară între aplicațiile native și cele web, FileStorm devine instalabilă pe toate platformele, nu doar în browser și are pe lângă o bună parte din funcționalitățile unei aplicații native precum utilizarea cache-ului pentru disponibilitate offline, și beneficiul adăugat prin independentă față de diverse magazine pentru aplicații cum sunt PlayStore sau AppStore.

Cu toate acestea, datorită faptului că standardul este nou, nu beneficiază încă de un suport suficient pentru a depăși aplicatiile native, însă este într-o continuă dezvoltare și devine din ce în ce mai dezirabil pentru clienți, fie ei simplii utilizatori dar și diverse companii.

În final, consider că proiectul propus își îndeplinește scopul, de a fi o platformă destinată stocării și distribuirii de fișiere, modernă, performantă, dezvoltată cu cele mai noi tehnologii, independentă de alte servicii, disponibilă pe diverse dispozitive ce poate fi utilizată de către oricine vrea să își păstreze confidențialitatea datelor, pe un sistem propriu și administrabil.

BIBLIOGRAFIE

- [1] – “Java (limbaj de programare)” -
[https://ro.wikipedia.org/wiki/Java_\(limbaj_de_programare\)](https://ro.wikipedia.org/wiki/Java_(limbaj_de_programare))
- [2] – “Lesson 2 — Behind The Scenes” – <https://medium.com/@PrayagBhakar/lesson-2-behind-the-scenes-4df6a461f31f>
- [3] – “1. Introduction to Spring Framework” – <https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html>
- [4] – “Spring Boot Architecture and Workflow” – <https://dzone.com/articles/spring-boot-architecture-and-workflow>
- [5] – “TypeScript” – <https://ro.wikipedia.org/wiki/TypeScript>
- [6] – “2021 Developer Survey” – <https://insights.stackoverflow.com/survey/2021>
- [7] – “Angular” – <https://ro.wikipedia.org/wiki/Angular>
- [8] – “What is Angular?” – <https://angular.io/guide/what-is-angular>
- [9] – “Two-way binding” – <https://angular.io/guide/two-way-binding>
- [10] – “The RxJS library” – <https://angular.io/guide/rx-library>
- [11] – “How to Understand RxJS Operators by Eating a Pizza: zip, forkJoin, & combineLatest Explained with Examples” –
<https://www.freecodecamp.org/news/understand-rxjs-operators-by-eating-a-pizza/>
- [12] – “Angular Components” – <https://material.angular.io/components/categories>
- [13] – “Sass (stylesheet language)” –
[https://en.wikipedia.org/wiki/Sass_\(stylesheet_language\)](https://en.wikipedia.org/wiki/Sass_(stylesheet_language))
- [14] – “Progressive web application” –
https://en.wikipedia.org/wiki/Progressive_web_application
- [15] – “Hypertext Transfer Protocol” –
https://ro.wikipedia.org/wiki/Hypertext_Transfer_Protocol
- [16] – “HTTP (Hyper Text Transfer Protocol)” – <https://www.javatpoint.com/http>
- [17] – “WebSocket” – <https://en.wikipedia.org/wiki/WebSocket>
- [18] – “An Introduction to WebSockets” – An Introduction to WebSockets
- [19] – “IoT Hub: What Use Case for WebSockets?” – <https://blog.scaleway.com/iot-hub-what-use-case-for-websockets/>
- [20] – “Git: Reference Sheet” – <https://support.nesi.org.nz/hc/en-gb/articles/360001508515-Git-Reference-Sheet>
- [21] – “SSH” – <https://ro.wikipedia.org/wiki/SSH>
- [22] – “Debugger” – <https://jwt.io/>
- [23] – “To BLOB or not to BLOB” – Russell Sears, Catharine van Ingen, Jim Gray:
Microsoft Research : University of California at Berkeley – 2006

ANEXA 3

DECLARAȚIE DE AUTENTICITATE A LUCRĂRII DE FINALIZARE A STUDIILOR *

Subsemnatul PETRICĂ MARIAN - DAIAN,

legitimat cu Ci seria TZ nr. 574506,

CNP 5010823715201

autorul lucrării APLICAȚIE WEB PROGRESIVĂ PENTRU
STOCAREA FIȘIERELOR

elaborată în vederea susținerii examenului de finalizare a studiilor de LICENȚĂ organizat de către Facultatea DE AUTOMATICA ȘI CALCULATOARE din cadrul Universității Politehnica Timișoara, sesiunea IUNIE a anului universitar 2021 - 2022, coordonator SL.DR.ING. RAUL ROBU, luând în considerare conținutul art. 34 din Regulamentul privind organizarea și desfășurarea examenelor de licență/diplomă și disertație, aprobat prin HS nr. 109/14.05.2020 și cunoscând faptul că în cazul constatării ulterioare a unor declarații false, voi suporta sancțiunea administrativă prevăzută de art. 146 din Legea nr. 1/2011 – legea educației naționale și anume anularea diplomei de studii, declar pe proprie răspundere, că:

- această lucrare este rezultatul propriei activități intelectuale,
- lucrarea nu conține texte, date sau elemente de grafică din alte lucrări sau din alte surse fără ca acestea să nu fie citate, inclusiv situația în care sursa o reprezintă o altă lucrare/alte lucrări ale subsemnatului.
- sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.
- această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de licență/diplomă/disertație.

Timișoara,

Data

17.06.2022

Semnătura



* Declarația se completează „de mână” și se inserează în lucrarea de finalizare a studiilor, la sfârșitul acesteia, ca parte integrantă.