UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENŢĂ

ReaderLiSh - Read, Like & Share

propusă de

Buzemurgă Mihaela

Sesiunea: *februarie, 2017*

Coordonator ştiinţific
Asist. Dr. Vasile Alaiba

UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

FACULTATEA DE INFORMATICĂ

ReaderLiSh - Read, Like & Share

Buzemurgă Mihaela

Sesiunea: *februarie, 2017*

Coordonator ştiinţific

Asist. Dr. Vasile Alaiba

DECLARAŢIE PRIVIND ORIGINALITATE ŞI RESPECTAREA
DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licenţă cu titlul „*ReaderLiSh - Read, Like & Share*” este scrisă de mine și nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;
- codul sursă, imaginile etc. preluate din proiecte *open-source* sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
- rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Iaşi,

Absolvent *Buzemurgă Mihaela*

_____

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că nu sunt de acord ca Lucrarea de licență cu titlul „*ReaderLiSh - Read, Like & Share*”, codul sursă al programelor și celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, nu sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” Iași să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi,

Absolvent *Buzemurgă Mihaela,*

_____

Cuprins

Introducere

„Cartea reflectă ca o oglindă lungul şir de secole al vieţii omenirii, istoria luptei sale pentru existenţă, pentru un viitor mai luminos, suferinţele, bucuriile, înfrângerile şi biruinţele sale toate. Iubiţi cartea, îngrijiţi-o şi citiţi cât mai mult. Cartea ne este prieten credincios, de nădejde."

(G. F. Morozov)

Motivație

Sunt o amatoare a cărților, în special a celor motivaționale și de dezvoltare personală. Consider că niciodată nu ar trebui să ne oprim din citit, acesta reprezentând principalul stimulent al creierului. Știind că nu sunt singura care gândește în acest fel, și că mai sunt și alte persoane care împărtășesc această idee, m-am gândit la o aplicație care să vină în ajutorul celor care doresc să citească și în momentele în care nu își pot lua o carte cu ei. După cum știm, tehnologia a avansat într-un ritm destul de alert, iar telefonul a devenit indispensabil în zilele noastre. Așadar, aplicația va fi destinată celor care dispun de un dispozitiv mobil cu Android.

ReaderLiSh - Read, Like & Share este un concept de aplicație care dispune de mai multe functionalitați. În primul rând aplicația permite citirea cărților, acesta fiind principalul scop. În al doilea rand, aplicația permite cu ajutorul GPS-ului găsirea de biblioteci și librării în zona în care te afli, sau o alta selectată de utilizator. Ceea ce aduce nou aplicația, față de cele deja existente, este opțiunea cu ajutorul căreia poți căuta o carte după o anumită stare de spirit. Dacă utilizatorul dorește într-un moment de supărare să citească o carte care să îi ridice moralul și să îi îmbunătățească starea de spirit, va putea foarte ușor să scrie sentimentul "fericire" sau "bucurie" sau orice alt sentiment, iar aplicația îi va returna din baza de date o serie de cărți care fac parte din categoria aleasă. Pe langă aceste functionalități, ReaderLiSh - Read, Like & Share va funcționa și ca un fel de rețea de socializare. Citind o carte, un utilizator îi poate acorda un

calificativ și o poate recomanda și celorlalți utilizatori care folosesc această aplicație. De asemenea va fi posibilă încărcarea uneia, sau a mai multor cărți în baza de date a aplicației.

Toate aceste facilități le-am îmbinat într-un singur concept, ReaderLiSh - Read, Like & Share, nume compus din principalele funcționalități ale aplicației: Read, Like & Share.

Context

Un prim argument în ceea ce privește rolul lecturii în dezvoltarea personală este efectul pe care-l exercită o carte asupra unui individ: prin lecturarea unei cărți, omul își îmbogățește lumea interioară prin intermediul imaginației. Desigur, există și multe filme bazate pe anumite cărți, însă avantajul citirii unei opere este că fiecare își poate imagina într-un anumit fel un personaj, un loc sau poate să judece în felul său un eveniment.

Un al doilea argument în favoarea lecturii este faptul că aceasta oferă subiecte de meditație și reprezintă o sursă imensă de idei, idei care pot fi împărtășite altor oameni sau chiar aplicate în viața de zi cu zi. Astfel, un anumit paragraf dintr-o carte de psihologie poate să reprezinte un răspuns la o întrebare care demult sălășuiește în mintea individului. Ideea principală a unui roman poate fi privită din mai multe puncte de vedere, la fel și acțiunile protagonistului sau chiar decorul, iar aceste aspecte ale unei cărți pot fi discutate la infinit cu un om ce a citit la rândul său opera dată.

În concluzie, consider că lectura este un aspect important în viața individului, indiferent de forma pe care o are: carte, revistă, ziar, carte audio. În dezvoltarea personală aceasta joacă rolul de sursă de informație, ajută la antrenarea imaginației și reprezintă o formă de socializare și descoperire a oamenilor ce împart aceeași pasiune.[1]

[1] http://www.eseuargumentativromana.com/2016/07/rolul-lecturii-in-dezvoltarea-personala.html

Obiectiv

Îmi propun realizarea unei aplicații care să vină în ajutorul persoanelor pasionate de citit, persoane care doresc să-și îmbunătățească cultura generală prin lecturarea unor cărti de pe un dispozitiv mobil. Aplicația este destinată celor care nu dispun de foarte mult timp liber, dar chiar și așa, acest timp vor să îl investească într-un mod cât mai inteligent.

Cerințe funcționale

În momentul în care un utilizator își va descărca și instala aplicația, își va crea un cont în baza de date. Contul va fi completat și personalizat în funcție de preferințele fiecărui utilizator.

Principalul scop al aplicației este citirea cărților. Așadar, utilizatorii vor putea citi cărțile puse la dispoziție de către aplicație în momentul instalării, sau le vor putea citi pe cele încărcate de ei.

Odată deschisă o carte, utilizatorii vor putea face și mici adnotări asupra textului. Ei vor putea sublinia un paragraf și desigur să selecteze și anumite fragmente din carte care li se par mai interesante, pentru ca mai apoi să le poată asocia un sentiment.

Prin activarea GPS-ul, aplicația va avea activă opțiunea de a căuta bibliotecile și librăriile din zona în care se află, sau o altă zonă setată de utilizator. Aplicația pune la dispoziție și această facilitate, deoarece încurajează și cititul cărților sub formă tipărită.

După citirea fiecărei cărti, utilizatorul poate aprecia sau nu cartea, printr-o notă.

Așa cum menționam la început, aplicația funcționează și ca o rețea de socializare, adică utilizatorii pot face anumite recomandări pe baza cărților citite.

CAPITOLUL I. Tehnologii folosite:

Pentru dezvoltarea aplicției “ReaderLiSh - Read, Like & Share” au fost folosite următoarele tehnologii:

- Sistemul de operare Android
- Android Google Maps API
- Server Java
- Fire de execuție
- SQLite
- Biblioteca MuPDF

Sistemul de operare Android

Android este un sistem de operare pentru dispozitive și telefoane mobile, bazat pe nucleul Linux, dezvoltat inițial de compania Google, iar mai târziu de consorțiul comercial Open Handset Alliance. Android permite dezvoltatorilor să scrie cod gestionat în limbajul Java, controlând dispozitivul prin intermediul bibliotecilor Java dezvoltate de Google. Aplicații scrise în C și în alte limbaje pot fi compilate în cod mașină ARM și executate, dar acest model de dezvoltare nu este sprijinit oficial de către Google. Lansarea platformei Android, la 5 noiembrie 2007, a fost anunțată prin fondarea Open Handset Alliance, un consorțiu de companii de hardware, software și de telecomunicații, consacrat dezvoltării de standarde deschise pentru dispozitive mobile. Google a lansat cea mai mare parte a codului Android sub licența Apache, o licență de tip free-software și open-source.

Android a fost disponibil ca *Open Source* începând din 21 octombrie 2008. Google a deschis întregul cod sursă, care anterior era disponibil sub licența Apache. Aceasta permite producătorilor să adauge extensii proprietare, fără a le face disponibile comunității open source. În timp ce contribuțiile Google la această platformă se așteaptă să rămână open source, numărul versiunilor derivate ar putea exploda, folosind o varietate de licențe.

SDK-ul Android conține un set de instrumente de dezvoltare, precum biblioteci, program de depanare, un emulator de dispozitiv, documentație, mostre de cod și tutoriale menite să simplifice dezvoltarea de aplicații mobile. Platformele de dezvoltare sprijinite în prezent înglobează calculatoare pe X86 care rulează Linux, Mac OS sau mai recent Windows 10 Îmbunătățirile aduse la SDK-ul Android facilitează dezvoltarea de aplicații destinate primelor versiunii ale platformei Android, astfel se pot realiza aplicații și pentru dispozitivele mai vechi. Instrumentele pentru dezvoltare pot fi descărcate în funcție de platforma și versiunea dorită pentru a se asigura compatibilitatea acestora cu dispozitivele pentru care sunt realizate.

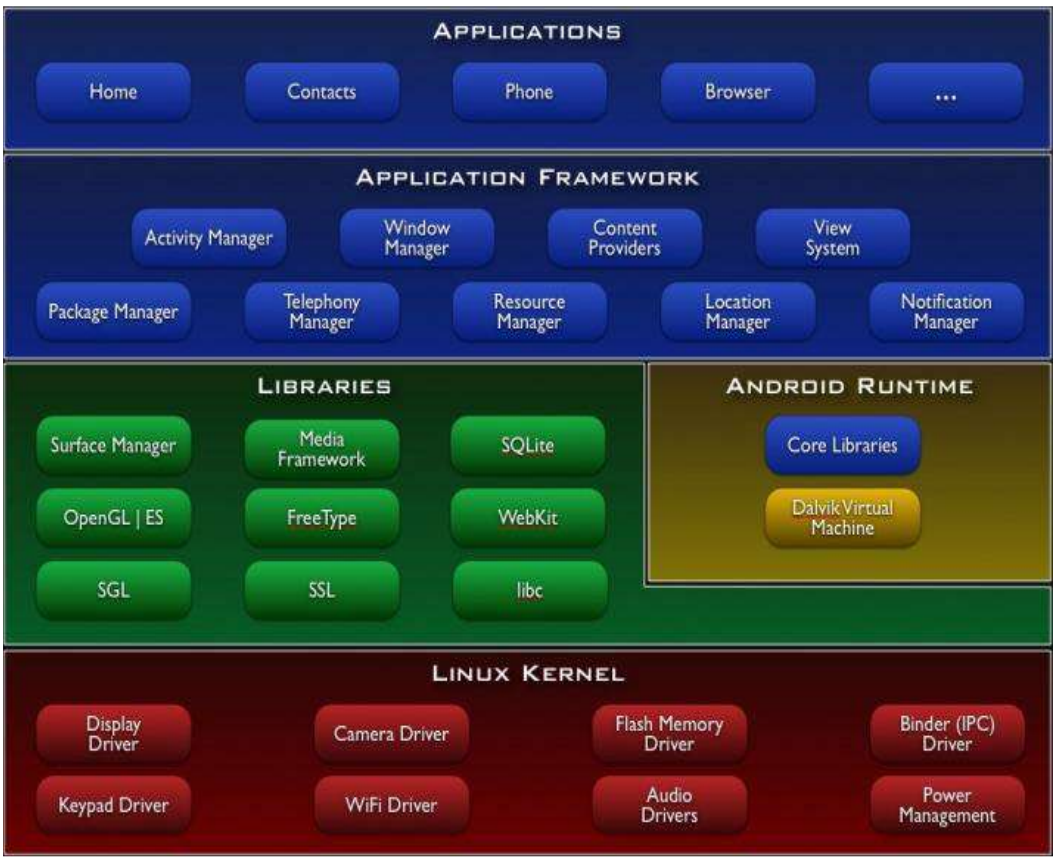
Versiuni Android

Versiune Android	Nivel API	Data Lansării	Nume de Cod	Cota de Piață
6.0 - 6.0.1	23	05.10.2015	Marshmellow	1.2%
5.1 - 5.1.1	22	09.03.2015	Lollipop	17.1%
5.0 - 5.0.2	21	12.11.2014	Lollipop	17.0%
4.4W - 4.4W.2	20	25.06.2014	KitKat with wearable extensions	
4.4 - 4.4.4	19	31.10.2013	KitKat	35.5%
4.3	18	24.07.2013	Jelly Bean	3.4%
4.2.x	17	13.11.2012	Jelly Bean	11.7%
4.1.x	16	09.07.2012	Jelly Bean	8.8%
4.0.3 - 4.0.4	15	16.12.2011	Ice Cream Sandwich	2.5%
4.0 - 4.0.2	14	19.10.2011	Ice Cream Sandwich	
3.2	13	15.07.2011	Honeycomb	
3.1	12	10.05.2011	Honeycomb	
3.0	11	22.02.2011	Honeycomb	
2.3.3 - 2.3.7	10	09.02.2011	Gingerbread	2.7%
2.3 - 2.3.2	9	06.12.2010	Gingerbread	
2.2 - 2.2.3	8	20.05.2010	Froyo	0.1%
2.1	7	12.01.2010	Eclair	
2.0.1	6	03.12.2009	Eclair	
2.0	5	26.10.2009	Eclair	
1.6	4	15.09.2009	Donut	
1.5	3	30.04.2009	Cupcake	
1.1	2	09.02.2009		
1.0	1	23.09.2008		

Figură 1 Versiuni Android

Arhitectura sistemului de operare Android

În continuare aș dori să prezint arhitectura sistemului de operare Android, care cuprinde cinci secțiuni pe patru nivele, acestea fiind: Kernelul Linux, biblioteci, motorul Android și cadrul pentru aplicații, după cum se poate observa și în Figură 2.



Figură 2 Arhitectura sistemului de operare

Kernelul Linux utilizează o serie de patch-uri pentru versiune oficială, și conține toate driverele pentru componentele hardware, precum camera, tastatura, WiFi și dispozitive audio. De asemenea, la Kernelul Linux au fost adăugate unele funcționalități specifice Android, acestea fiind: wakelocks, low-memory killer, binder, logger.

Wakelocks este o soluție pentru problema de power management folosită de Android, care are ca scop reducerea consumului prin intrarea în starea de ”sleep” pentru momentele în care nu este utilizat.

În ceea ce privește *low memory killer*, scopul său este de a opri componentele care nu au fost folosite o perioadă lungă de timp.

Următoarea funcționalitate este reprezentată de *binder*, un mecanism de RPC/IPC, care are ca scop capabilitate de invocare remote a obiectelor asemănătoare cu obiectele COM din Windows.

Și în cele din urmă *logger*, care are ca scop substituirea *sistemului clasic de logging* al kernelului, cu scopul diminuării numărului de task switch-uri și scrieri în fișier, printr-un buffer circular.

O altă componentă este *motorul Android*, alcătuit dintr-o serie de biblioteci de bază, care permit utilizatorilor să dezvolte aplicații mobile, folosind ca limbaj de programare Java; aceste biblioteci permit accesul la funcțiile unui dispozitiv și anume: telefonie, mesaje, gestiunea pachetelor. La baza arhitecturii stau regiștri, fiind echipată cu un compilator JIT, care permite modificarea executabilului obținut pe dispozitivul mobil.

Următoarea componentă este cadrul pentru aplicații: *Android framework*, care furnizează diverse funcționalități ale sistemului de operare pentru ca programatorii să le poată transpune în aplicațiile lor. Această componentă oferă dezvoltatorilor posibilitatea de a realiza aplicații complexe și inovative, întrucât aceștia sunt liberi să utilizeze hardware-ul echipamentelor, de informațiile despre locație, rularea de servicii în background, setarea de alarme, precum și adăugarea de notificări pe bara de stare. Programatorilor li se oferă acces la aceleași API-uri ca și aplicațiile distribuite cu Android, prin urmare arhitectura aplicațiilor este proiectată astfel încât să fie simplificată reutilizarea componentelor: o aplicație poate publica anumite funcționalități, și o altă aplicație să le poată utiliza.

Ultima componentă este reprezentată de *nivelul aplicații*, care oferă atât produsele încorporate în dispozitivele mobile, precum: Camera, Music player, Contacts, și Video player, cât și produsele disponibile pe Play Store.

Funcționalitățile sistemului de operare Android

Principalele funcționalități pe care sistemul de operare Android le oferă sunt:

- *stocare*, care folosește SQLite, bază de date relațională ce permite utilizarea eficientă a resurselor;
- *conectivitatea* prin diverse modalități, precum: 3G, WiFi, Bluetooth, WiMAX, GPRS, EDGE;
- *WiFi direct*, care permite interconectarea între diverse dispozitive având o lățime de bandă mare;
- *Android Beam*, prin care utilizatorii partajează conținut instant prin apropierea dispozitivelor respective.
- *navigarea pe Internet* bazată pe motorul open source pentru navigare WebKit împreună cu motorul JavaScript de la Chrome V8 suportând HTML5 și CSS3.
- *Multimedia* admite mai multe formate precum: H.263, M-peg-4, AMR-WEB, AAC, JPEG;
- *multi-touch*, care suportă posibilitatea de contact în mai multe puncte concomitent;
- *multi-tasking*; *GCM*(Google Cloud Messaging) permițând dezvoltatorilor expedierea de date de dimensiuni reduse, în lipsa unei soluții de sincronizare proprietară.

Google Maps API

Google a lansat în anul 2005 Google Maps API, care permite programatorilor integrarea de hărților de la Google în propriile aplicații; acest serviciu fiind gratuit. API-ul permite accesul la serverele Google Maps, descărcarea de date, afișarea unei hărți, și trimiterea unui răspuns la interacțiunea cu harta.

O altă facilitate pe care API-ul Google Maps o furnizează este inserarea de informații suplimentare despre obiectele de pe o porțiune a hărții, astfel încât să permită interacțiunea

utilizatorului cu harta. Totodată, acesta permite adăugarea de obiecte grafice pe hartă, precum: ancore pentru pozițiile specifice pe hartă, cunoscute sub denumirea de markeri, segmente de linii, segmente închise, imagini și elemente grafice de tip bitmap atașate la poziții specifice pe hartă.

Hărțile afișate de Google Maps API prezintă următoarele caracteristici: titlul acestora nu includ conținut personalizat, iar referitor la pictograme, nu toate permit acțiunea de click. În plus față de funcționalitea de cartografie, API sprijină o gamă completă de interacțiuni cu harta, în concordanță cu modelul Android UI, un exemplu ilustrativ este posibilitatea configurării interacțiunilor cu harta, prin definirea de *listeneri*, concept care are ca scop oferirea de răspunsuri la gesturile utilizatorilor. Principala clasă care se utilizează în cazul utilizării hărților este *GoogleMap*. Aceasta modelează harta în cadrul aplicației, iar în cadrul interfeței utilizator, harta va fi reprezentată printr-un MapFragment sau MapView. Clasa *GoogleMap* permite realizarea următoarelor acțiuni: conectarea la serviciul Google Maps, descărcarea componentelor hărții, afișarea de diverse controale, precum zoom și răspunderea la acțiunea de zoom.

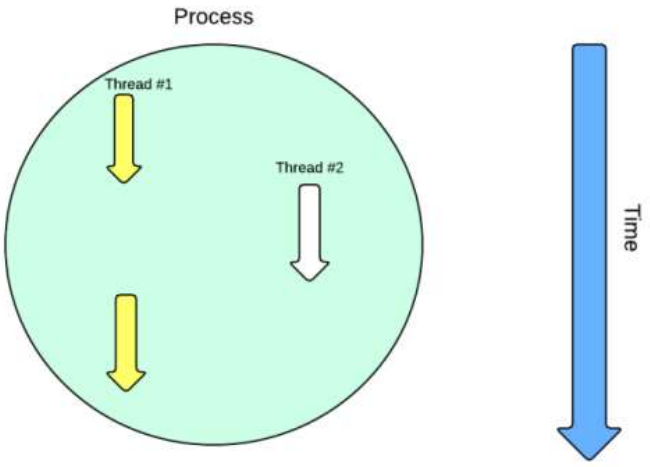
MapFragment este o subclasă Android, care permite plasarea unei hărți într-un fragment Android și oferă acces la obiecte de tip GoogleMap. Spre deosebire de o extensie de vizualizare (View), un fragment reprezintă un comportament sau o porțiune din interfața utilizator dintr-o activitate, astfel o activitate poate include mai multe fragmente. De asemenea, un fragment poate fi utilizat în mai multe activități, dar și utilizarea mai multor fragmente pentru realizarea unui interfețe.

În ceea ce privește *MapView* este o subclasă a clasei *Android View*, care permite plasarea hărții într-o extensie de vizualizare. Un View reprezintă o regiune a ecranului, element esențial în dezvoltarea de aplicații Android și widget-uri. Mai mult decât atât, prin intermediul unui MapFragment, MapView-ul acționează asemănător unui container pentru hartă.

Fire de execuție

Firele de execuție, ca și procesele, sunt un mecanism care permit unui program să facă mai multe lucruri în același timp. Kernel-ul Linux le împarte asincron, întrerupând la un interval de timp fiecare fir de execuție pentru a da și alora posibilitataea de a se executa. Astefel, asemenea proceselor, firele de execuție par să ruleze concurențial pentru a crește eficiența

programelor. În sistemele cu procesoare multiple sau cu nuclee multiple, firele de execuție rulează în același timp pe procesoare sau nuclee diferite. Pentru procesoarele cu un singur nucleu, sistemul împarte timpul de execuție între firele de execuție, aspect prezentat în Figură 3.



Figură 3 Fire de executie

La nivel de implementare se poate utiliza standardul *Pthreads (POSIX Threads)*, ce definește un API pentru crearea și manipularea firelor de execuție. Pentru implementarea unui server TCP care deservește clienții multiplii se utilizează acest standard deoarece aplicațiile ce folosesc fire de execuție sunt în general mai rapide decât dacă utilizează procese.

SQLite

SQLite este o bibliotecă care implementează un motor de baze de date SQL încapsulat, care nu are nevoie de o configurație și nici de un server. Codul pentru această librărie este public și poate fi folosit pentru orice scop.

Spre deosebire de alte baze de date SQL, SQLite nu are un server separat pentru procesare. Acesta citește și scrie direct în memoria sistemului. Oferă o bază de date complexă cu posibilitatea de a crea tabele multiple, indici, *trigger*-e și *view*-uri iar tranzacțiile sunt *ACID (Consistency, Isolation, Durability*) chiar dacă sunt întrerupte de erori. De asemenea, formatul

fișierelor scrise de aceasta bibliotecă este *cross-platform* oferind posibilitatea de a fi utilizat atât pe sisteme de 32 de biți cât și pe cele de 64 sau pe arhitecturi de tip *big-endian* sau *little-endian*. Datorită acestor facilitați, SQLite este opțiunea ideală în implementarea unei aplicații performante.

Biblioteca MuPDF

MuPDF este o librărie dezvoltată de Artifex Software și Inc, care ajută dezvoltatorii de aplicații să redea conținutul unui fișier cu format pdf, dar și sa poată prelucra acesta. Exemplu în acest sens sunt: afișarea conținutului, căutarea după un cuvant cheie, subliniere, adnotare. Această librărie este scrisă modular, astfel încât creatorii permit celorlalți dezvoltatori să adauge funcționalități noi, pentru a se mula pe nevoile lor.

CAPITOLUL II. Arhitectura Aplicației

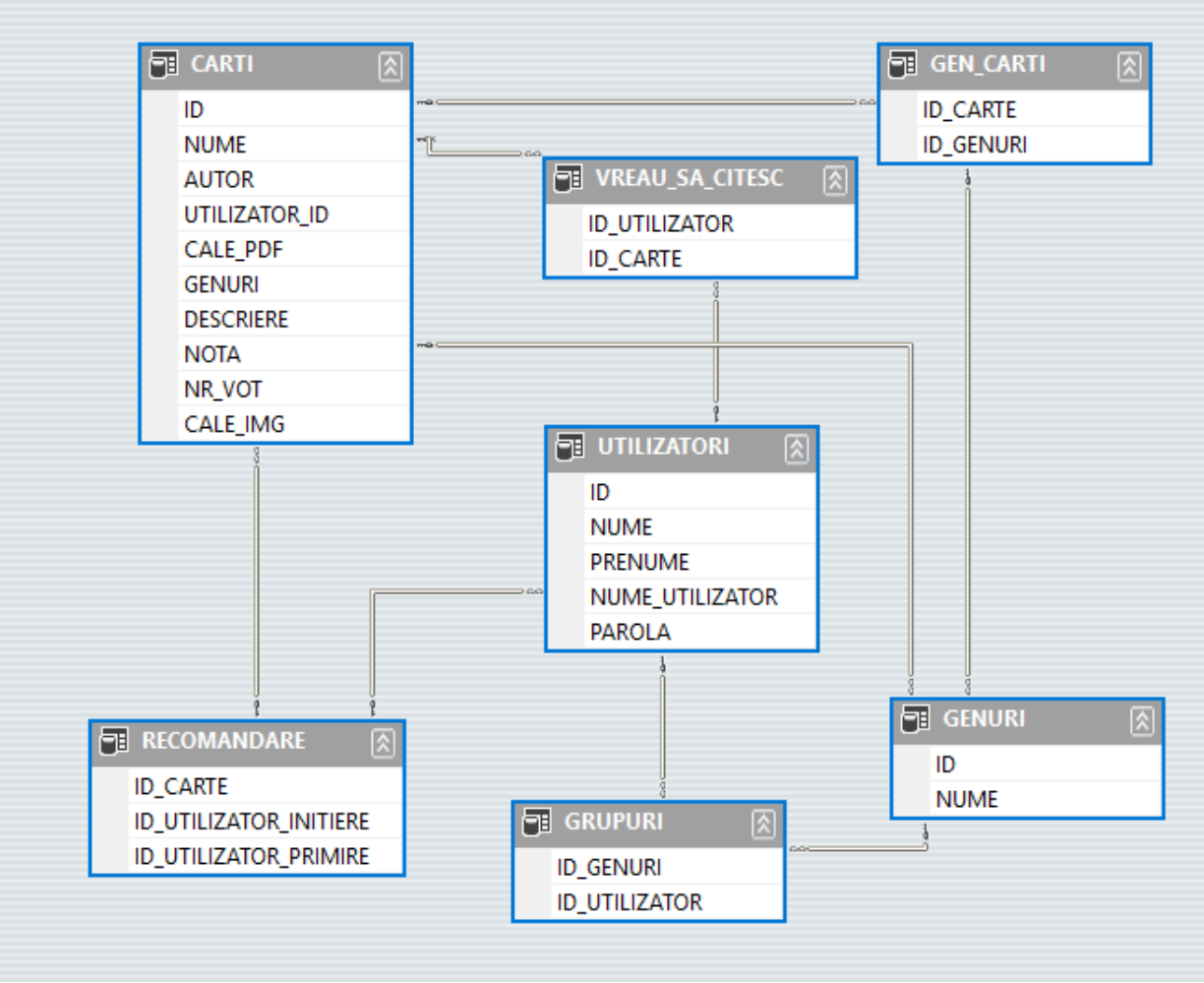
În acest capitol voi evidenția arhitectura aplicației **ReaderLiSh - Read, Like & Share**, principalele module ale aplicației, precum și modul de comunicare al acestora.

În ceea ce privește arhitectura aplicației, aceasta este de tip client-server, după modelul pe două nivele, în care primul este reprezentat de aplicația client și anume aplicația Android, iar cel de-al doilea este serverul Java. Pentru stocarea datelor am utilizat baza de date SQLite, care oferă eficiență.

Baza de date

Baza de date este formată dintr-un număr de șapte tabele, după cum se poate observa și în figura Figură 4.

- Tabela *Utilizatori* reține datele utilizatorilor care sunt înregistrați;
- Tabela *Carti* este tabela în care se reținn informații despre cărțile de pe server;
- Tabela *Genuri* stochează genurile cărților disponibile;
- Tabela *Gen_Carti* este o tabelă intermediară care face legătura dintre tabelele *Cărți* și *Genuri*; am folosit această tehnică pentru a evita relația "many to many";
- Tabela *Grupuri* este cea care reține grupurile din care face parte un utilizator;
- Tabela *Recomandare* păstreaza toate informațiile legate de recomandările dintre useri;
- Tabela *Vreau_Sa_Citesc* va fi folosita la a reține cărțile pe care un utilizator va dori să le citească în viitor;



Figură 4 Tabelele aplicației

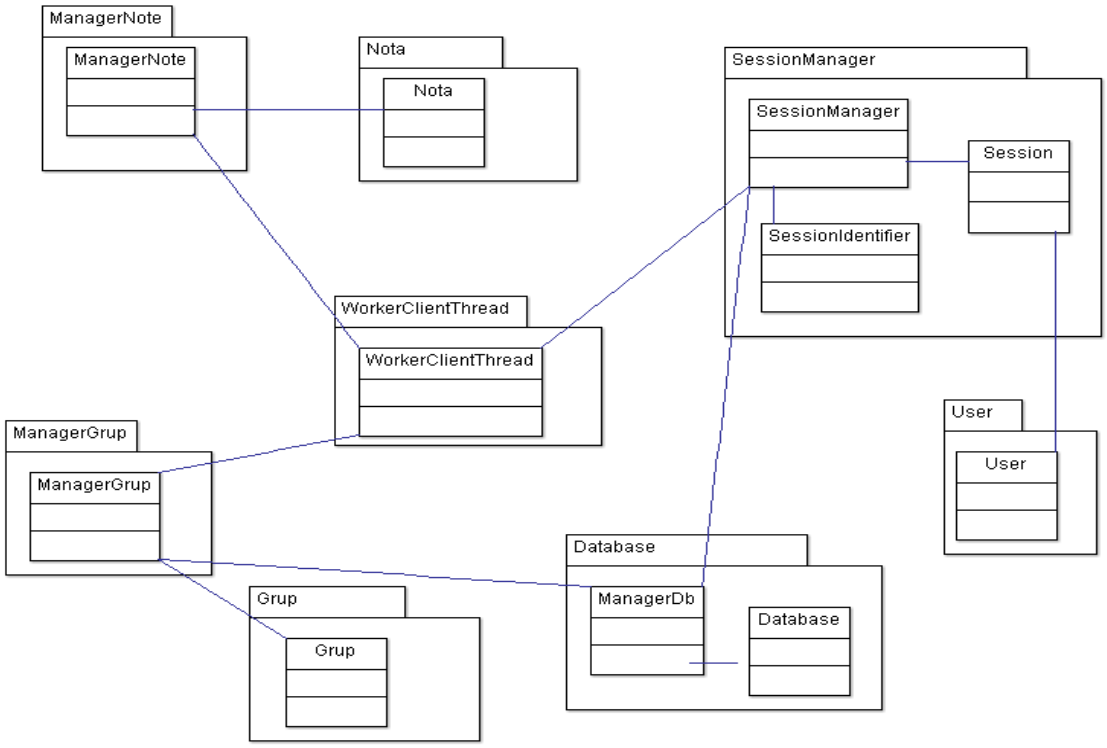
Server

Pentru implementarea server-ului Java, am urmărit cât mai multe principii de programare orientată pe obiecte pentru a realiza o aplicație ușor de înțeles, modificat, întreținut și testat.

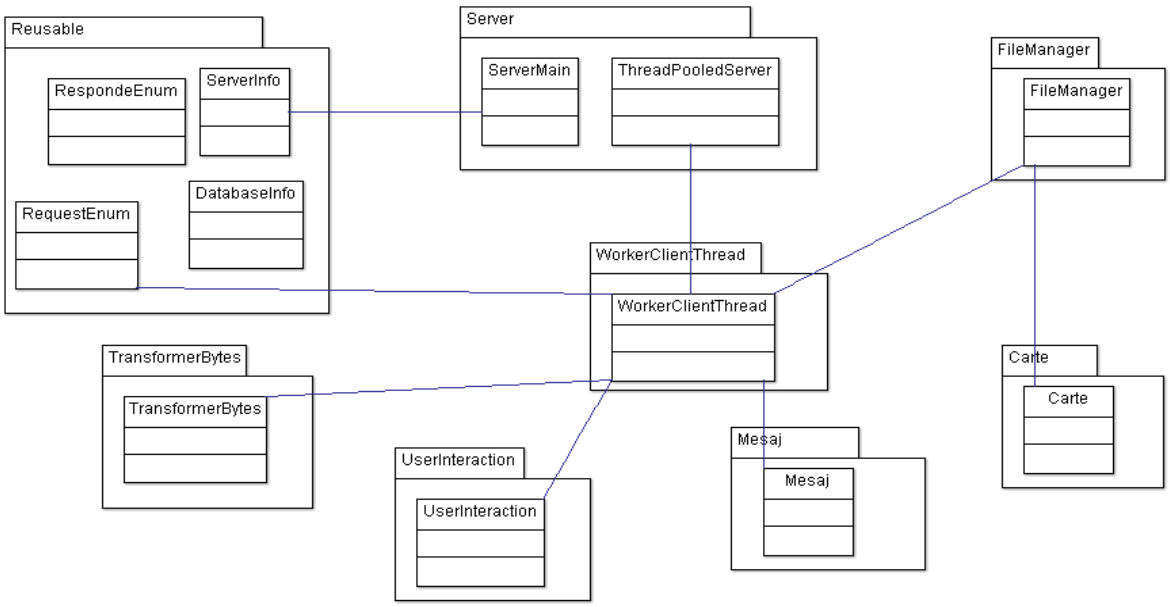
Astfel, clasele încapsulează datele, ascund reprezentarea lor și sunt ușor de refolosit. De asemenea, responsabilitățile sunt alocate astfel încât coeziunea în sistem să rămână ridicată. O coeziune ridicată înseamnă că responsabilitățile pentru un element din sistem sunt înrudite și concentrate în jurul aceluiași concept. Totodată, sarcinile sunt împărțite în așa fel încât cuplarea

rămâne slabă. O cuplare slabă presupune dependențe puține între clase, impact scăzut în sistem la schimbarea unei clase și potențial ridicat de refolosire.

În cele două figuri Figură 5 si Figură 6, este reprezentată structura serverului care este formată din 14 pachete.



Figură 5 Structură server



Figură 6 Structură server

Pentru o mai bună înțelegere a acestora, voi detalia fiecare pachet în parte, după cum urmează:

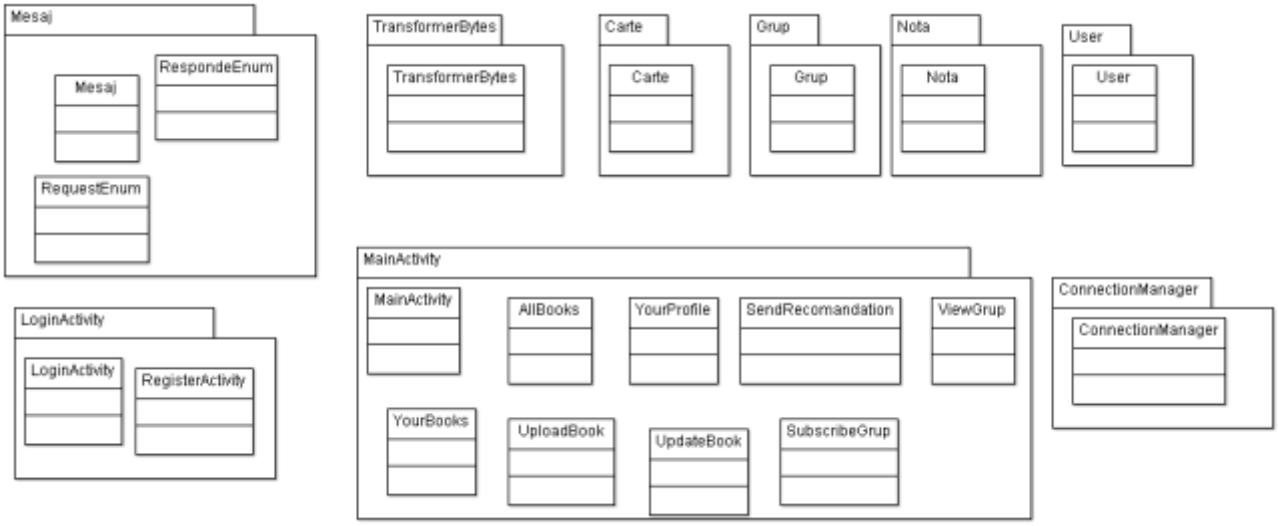
- Pachetul *Server* conține următoarele clase:
  - Clasa *ServerMain* este clasa principală care instanțiază obiectul *ThreadPooledSever*: acesta pornește toate firele de execuție destinate clienților. În momentul în care un client se conectează, acestuia i se atribuie un fir de execuție prin care îi vor fi procesate cererile.
  - Clasa *ThreadPooledServer*
- Pachetul *WorkerClientThread* conține clasa cu același nume.
  - *WorkerClientThread*: această clasă se ocupă exclusiv de cititrea mesajelor din partea clienților. În funcție de tipul mesajului, acesta va fi trimis unei clase specifice. În urma procesării, va rezulta un obiect de tip *Mesaj* care va fi trimis înapoi clientului;
- Pachetul *User*:
  - Clasa *User* conține datele despre un client;
- Pachetul *TransformerBytes*:
  - Clasa *TransformerBytes* va fi folosită pentru a serializa obiectul de tip *Mesaj*;

- Pachetul *SessionManager*:
    - Clasa *SessionManager* se ocupă de sesiunea utilizatorului, dar și de înregistrarea, logare și deconectarea acestuia. Aceasta mai conține două clase cu următoarele funcționalități:
        - *SessionIdentifier*, folosită pentru a returna un identificator unic pentru sesiunea utilizatorului;
        - *Session*, care stochează datele sesiunii.
- Pachetul *Reusable* conține patru enumerații pentru a facilita lizibilitatea codului, dar și pentru optimizarea acestuia (modificarea unei valori dintr-o enumerație va duce la actualizarea automată a acesteia în codul serverului):
        - *ServerInfo*
        - *RespondeEnum*
        - *DatabaseInfo*
        - *RequestEnum*
- Pachetul *Nota*:
    - Clasa *Nota* conține informațiile despre o notă.
- Pachetul *Mesaj*:
    - Clasa *Mesaj* este folosită pentru a stoca mesajele dintre client și server. Acest proces se va realiza prin serializare.
- Pachetul *ManagerGrup*:
    - Clasa *ManagerGrup* se ocupă de organizarea grupurilor.
- Pachetul *Grup*:
    - Clasa *Grup* stochează informațiile unui grup.
- Pachetul *FileManager*:
    - Clasa *FileManager* se ocupă de prelucrarea cărtilor și anume salvarea și clasificarea lor pe server;
- Pachetul *Database* conține doua clase:
    - Clasa *ManagerDb* se ocupă de pornirea bazei de date, dar și de toate cererile celorlalte clase;
    - Clasa *Database* se ocupă strict de conexiunea cu SQLite.

- Pachetul *UserInteraction*:
    - Clasa *UserInteraction* se ocupă de notificările utilizatorului, dar și de recomandările dintre utilizatori;
- Pachetul *ManagerNote*:
    - Clasa *ManagerNote* se ocupă de interacțiunea utilizatorului cu notele;
- Pachet *Carte*:
    - Clasa *Carte* se ocupă de stocarea informațiilor despre o carte.

Client

În ceea ce priveste clientul, acesta este structurat în mai multe module, după cum se poate observa și în figura Figură 7



Figură 7 Structură Client

1. Modulul principal care contine următoarele pachete:
    - *MainActivity*- acest pachet conține clasa *MainAcitvity* în care se realizează întreaga logică a aplicației. De asemenea acesta încarcă și „fragmentul” necesar acțiunii dorite de catre utilzator.

- *AllBooks*
- *RegisterAccount*
- UploadBook
- *YourBooks*
- *UpdateBook*
- *YourProfile*
- *SendRecomandation*
- *ViewGrup*
- *SubscribeGrup*

2. Modulul de comunicare, cu ajutorul acestuia se realizează comunicarea dintre aplicație și server. Acesta conține pachetele:
   - *Mesaj*, de care la rândul său aparțin clasele:
     - *Mesaj*
     - *RequestEnum*
     - *RespondeEnum*
   - *Carte*
   - *Grup*
   - *Nota*
   - *User*
   - *ConnectionManager*
   - *TransformemBytes*
3. Modulul de logare, care conține doar clasele:
   - *LoginActivity*
   - *RegisterActivity*

CAPITOLUL III. Detalii de implementare

În continuare voi prezenta detaliile de implementare ale proiectului, și anume clientul și serverul.

Server

Serverul este un proces ce va rula în permanență pentru a facilita utilizatorilor toate funționalitățile aplicației. Astfel, la deschiderea serverului, acesta creează un *socket*[2] de conectare căruia îi atribuie o adresă și un port la care va asculta. Totodată se instanțiază clasa *ThreadPoolServer* care va creea un număr predefinit de fire de execuție. Am folosit această metodă deoarece micșorează șansa ca serverul să fie suprapopulat și să fie oprită funcționarea lui. Astfel, în momentul conectării unui utilizator, acestuia i se va atribui un fir de execuție deja creat (prin clasa *ThereadPoolServer*). Folosind această modalitate, serverul folosește mai puține resurse, evitând scenariul în care trebuie să aloce și să dealoce memorie pentru crearea și ștergerea unui fir. Totodată este îmbunatățită și viteza de rurale a serverului. Mai jos (Listare 1) se poate vedea modul în care acestea sunt create. S-a folosit clasa *java.util.concurrent.Executors* care furnizează metoda *newFixedThreadPool* pentru a crea aceste fire de execuție, primind ca parametru numărul de fire dorit.

```
protected ExecutorService threadPool =
        Executors.newFixedThreadPool(ServerInfo.SERVER_MAX_USERS.getValue());
public void run(){
        synchronized(this){
            this.runningThread = Thread.currentThread();
        }
        openServerSocket();
        while(! isStopped()){
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(isStopped()) {
                    System.out.println("Server Stopped.") ;
                    break;
```

[2] Un socket reprezintă un punct de conexiune într-o rețea TCP\IP. Când două programe aflate pe două calculatoare în rețea doresc să comunice, fiecare dintre ele utilizează un socket. Unul dintre programe (serverul) va deschide un socket si va aștepta conexiuni, iar celălalt program (clientul), se va conecta la server și astfel schimbul de informații poate începe. Pentru a stabili o conexiune, clientul va trebui să cunoască adresa destinației (a pc-ului pe care este deschis socket-ul) și portul pe care socketul este deschis.

```
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            this.threadPool.execute(
                new WorkerClientThread(clientSocket));
        }
        this.threadPool.shutdown();
        System.out.println("Server Stopped.") ;
    }}
```

Listare 1 Crearea firelor de execuție și administrarea clienților

În momentul în care firele de execuție au fost create, serverul va intra într-o stare de așteptare, va asculta la adresa setată pană când un client se va conecta. Odată conectat, se extrage un fir de execuție și i se atribuie un client prin crearea unui *WorkerClientThread*. Această clasă va fi punctul de comunicare cu, clientul. Va prelua toate mesajele de la client care vor fi prelucrate, și va întoarce răspunsurile aferente.

Fiecare mesaj transmis între client și server, și viceversa constă într-o clasa Mesaj care conține:

- Obiect, ce poate lua orice formă din lista Carte, Grup, Nota, User, dar și tipuri predefinite;
- RespondeEnum, care va lua o valoare în funcție de răspunsul furnizat (un exemplu în acest caz poate fi cel din momentul logării, dacă logarea s-a realizat cu success sau nu);
- RequestEnum, ce va primi o valoare care indică funcția care trebuie să se apeleze în interiorul serverului.

Mesajul va fi serializat și trimis către server sau client, sub următoarea formă: primul bit va avea rolul de început al mesajului, după care urmează lungimea mesajului. La sfârșit va conține obiectul mesaj serializat. Între toate acestea există un separator.

Pentru serializare s-a folosit clasa *SerializationUtils.serialize* care transformă obiectul într-un vector de biți, iar pentru crearea mesajului propriu zis s-a folosit funcția *getDataPacket*, funcție care se poate vedea în Listare 2.

```
public static byte[] getDataPacket(Object obiect) {
		byte[] data = SerializationUtils.serialize((Serializable) obiect);
		byte[] packet = null;
		try {
			byte[] separator = new byte[1];
			separator[0] = 4;
			byte[] initialize = new byte[1];
			initialize[0] = 2;
			byte[] data_length =
String.valueOf(data.length).getBytes("UTF8");
			packet = new byte[initialize.length+separator.length +
data_length.length + data.length];

			System.arraycopy(initialize, 0, packet, 0, initialize.length);

			System.arraycopy(data_length, 0, packet, initialize.length,
data_length.length);
			System.arraycopy(separator, 0, packet,
initialize.length+data_length.length, separator.length);
			System.arraycopy(data, 0, packet,
initialize.length+data_length.length + separator.length, data.length);
		} catch (UnsupportedEncodingException ex) {
		}
		return packet;
}
```

Listare 2 Serializarea obiectelor

Fiecare mesaj primit se va citi în funcția run() a clasei WorkerClientThread. Astfel, folosind descriptorul aferent, se va citi bitul de inițializare. În cazul în care acesta este corect se va citi și restul mesajului. Acesta este deserializat tot cu ajutorul clasei *SerializationUtils* transformând vectorul de biți într-un obiect, care la rândul lui va fi convertit la tipul *Mesaj*.

În urmatoarea listare (Listare 3) se poate vedea o parte din modul acesta de lucru.

```
public void run() {
	while (true) {
		TransformerBytes mesaj;
		byte[] initilize = new byte[1];
		try {
			din.read(initilize, 0, initilize.length);
			if (initilize[0] == 2) {
				byte[] recv_data = ReadStream();
				if (recv_data != null) {
					Mesaj newMesaj=null;
		Mesaj m_mesaj = (Mesaj) SerializationUtils.deserialize(recv_data);
```

Listare 3 Citirea mesajelor

În funcție de RequestEnum-ul primit, serverul va apela funcția necesară, astfel încat să se realizeze cu succes operația dorită.

```
switch (m_mesaj.getCmd()) {
	case REQUEST_LOGIN:

	newMesaj=SessionManager.getSession().Login((User)m_mesaj.getObiect());

	dout.write(TransformerBytes.getDataPacket(newMesaj));
	dout.flush();
	break;
	case REQUEST_REGISTER:

	newMesaj=SessionManager.getSession().Register((User)m_mesaj.getObiect());

	dout.write(TransformerBytes.getDataPacket(newMesaj));
	dout.flush();
	break;
```

Listare 4 Apelarea unei funcții

Pentru citirea mesajelor s-a folosit funcția *ReadStream()*. Aceasta citește prima dată biții pentru lungimea mesajului, îi transformă în tipul *int*, după care îi asignează variabilei *data_lenght* noua valoare. În următorul pas se inițializează două variabile *byte_read* (câți biți au fost citiți) și *byte_offset* (până unde s-a citit din *buffer*[3]). În final, se citește mesajul propriu zis din *buffer*, cât timp *byte_offset* nu depășește *data_lenght* (lungimea mesajului).

```
private byte[] ReadStream() {
		byte[] data_buff = null;
		try {
			int b = 0;
			String buff_length = "";
			while ((b = din.read()) != 4) {
				buff_length += (char) b;
			}
			int data_length = Integer.parseInt(buff_length);
			data_buff = new byte[Integer.parseInt(buff_length)];
			int byte_read = 0;
			int byte_offset = 0;
			while (byte_offset < data_length) {
				byte_read = din.read(data_buff, byte_offset, data_length -
byte_offset);
				byte_offset += byte_read;
			}
		} catch (IOException ex) {

		}
		return data_buff;
	}
```

Listare 5 Citirea mesajelor

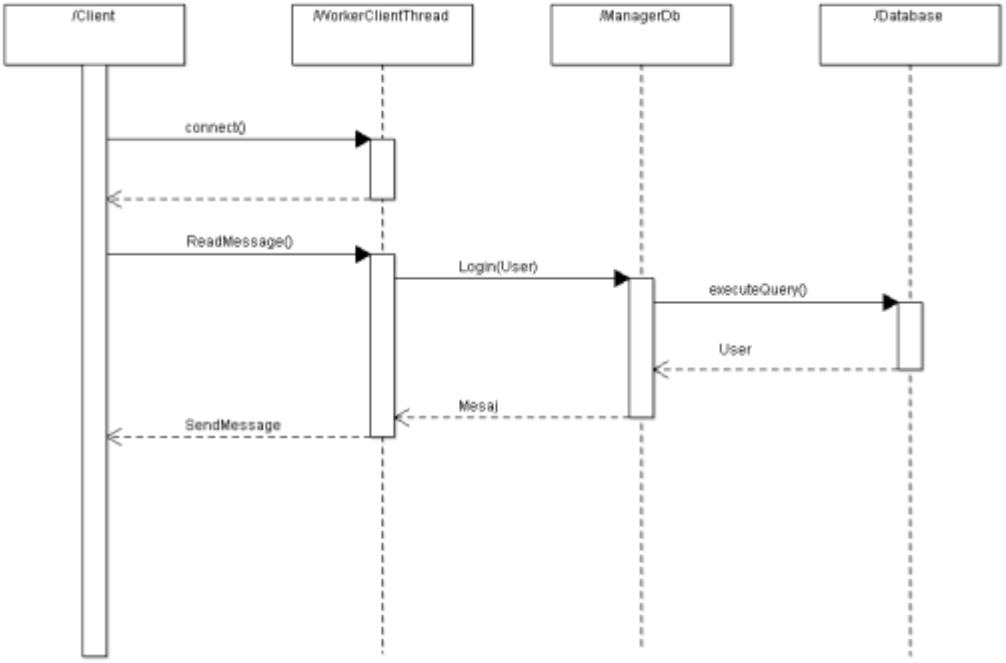
[3] Este o memorie necesară pentru stocarea temporară a unor informații. Dacă tipărești foarte repede și procesorul de cuvinte nu poate să afișeze atât de repede caracterele tipărite, caracterele culese sunt stocate intr-un buffer.

În următoarele subcapitole vom atinge unele dintre funcțiile importante ale serverului:

Prelucrarea bazei de date

Pentru a lucra cu baza de date s-a creat ManagerDb, care se ocupă de toate cererile serverului. Tot această clasă conține o instanță a clasei Database care face conexiunea cu SQLite. *ManagerDb* este o clasă *Singleton,* pentru a avea un singur punct din care se fac prelucrări asupra bazei de date. Pentru a sincroniza funcțiile s-a folosit cuvântul cheie *synchronized,* care se pune în fața funcției.

Logarea și Sesiunea



Figură 8 Logare

Pentru logare și pentru menținerea sesiunii s-a creat *SessionManager*. S-a mai creat și o clasă *SessionIdentifier*, rolul acesteia fiind generarea unui identificator unic pentru fiecare sesiune creată pe server. Cele trei funcții, de autentificare, înregistrare și deconectare, pot fi vizualizate în următoarele listări.

```
public void Logout(User user)
	{
		user_session.remove(user.getId());
	}
```

Listare 6 Funcția Logout()

În momentul în care utilizatorul se deconectează, acesta va fi șters și din sesiune.

```
public Mesaj Login(User user) {
		Mesaj mesaj=new Mesaj();

		if(!user_session.containsKey(user.getId()))
		{
			System.out.println("pregatim");
			user=ManagerDb.getSession().Logare(user);
			System.out.println(user.getNume());
			if (user.getLogat()) {

	mesaj.setM_raspunsServer(RespondeEnum.LOGIN_SUCCES);
				Session newSession=new Session(user);
				String
sessionId=SessionIdentifier.nextSessionId();
				user_session.put(sessionId, newSession);
				mesaj.setObiect(user);
			}
			else
			{
				mesaj.setM_raspunsServer(RespondeEnum.LOGIN_FAIL);
			}
		}
		else
		{
			System.out.println("nu exista");
			mesaj.setRaspuns("Nume/parola incorecta");
		}
		return mesaj;
	}
```

Listare 7 Funcția Login()

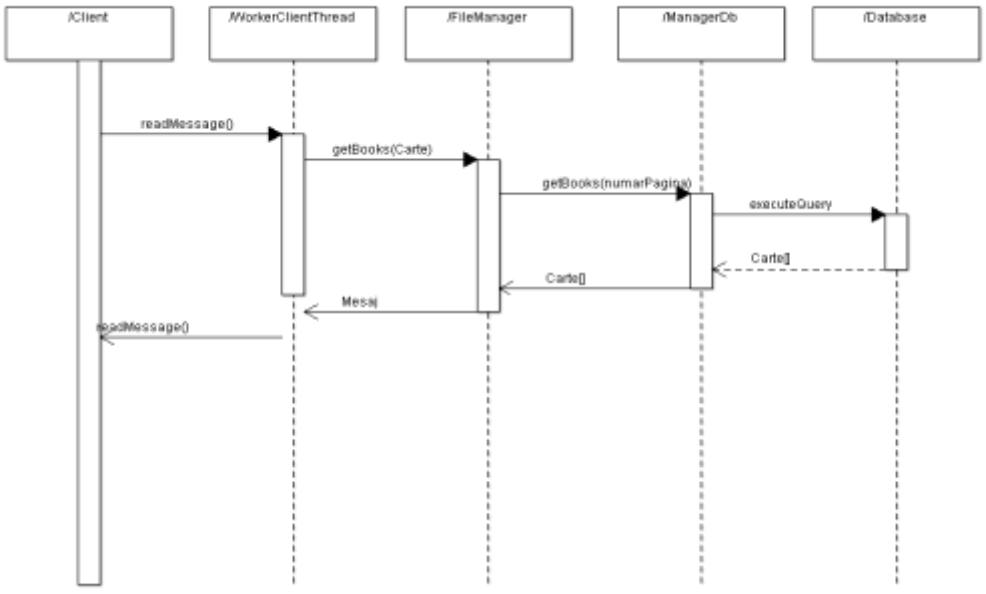
Când utilizatorul se autentifică, funcția *Login* va primi ca parametru un obiect de tip *User* care va conține doar numele și parola acestuia. Acest obiect este trimis la ManagerDb, care în funcție de corectitudinea datelor va returna tot un obiect *User* cu date valide sau nu. În cazul în care *getLogat()* are valoarea ‘adevărat’ se va crea o sesiune, i se va atribui un identificator unic și se va introduce obiectul *User* în interiorul mesajului ce va fi trimis la client. În funcție de rezultat se va întoarce ca răspuns în interiorul mesajului, LOGIN_SUCCES sau LOGIN_FAIL .

```
public Mesaj Register(User user)
	{
		Mesaj mesaj=new Mesaj();
		user=ManagerDb.getSession().Register(user);
		if(user.getId()!=-1)
		{
			Session newSession=new Session(user);
			String sessionId=SessionIdentifier.nextSessionId();
			user_session.put(sessionId, newSession);
			mesaj.setObiect(user);
			mesaj.setM_raspunsServer(RespondeEnum.REGISTER_SUCCES);
		}
		else
		{
			mesaj.setM_raspunsServer(RespondeEnum.REGISTER_FAIL);
		}
		return mesaj;}
```

Listare 8 Funcția Register()

Pentru a se înregistra din nou, se va primi ca parametru un obiect de tip *User* ce va fi populat cu datele necesare. Se va apela funcția *Register* din *ManagerDb*, după care se creează sesiunea aferentă și se va trimite la client mesajul cu obiectul returnat. Totodată, în funcție de rezultat, răspunsul poate avea două valori: REGISTER_SUCCES sau REGISTER_FAIL.

Lucrul cu fișiere



Figură 9 Lucrul cu fișiere

Acesta este un punct important din server, fiind modulul cel mai accesat. Astfel, pentru administrarea fișierelor s-a creat clasa *FileManager*. Aceasta este de tip *Singleton*, astfel accesul la fișierele stocate pe server se va face dintr-un singur punct. Tot această clasă face legătura cu ManagerDb pentru a prelua informații din baza de date. În continuare voi prezenta unele dintre cele mai importante funcții din această clasă.

Prealuarea cărților de pe server se poate vedea în urmatoare listare:

```
	public Mesaj getBooks(int pagina) {
		Carte []carte=ManagerDb.getSession().getBooks(pagina);
		for(int i=0;i<carte.length;i++)
		{
			if(carte[i]!=null)
			{
	carte[i].setImagine(getFileFromServer(carte[i].getCale_img().replaceFirst("^/
(.:/)", "$1")));
			}
		}
		Mesaj mesaj=new Mesaj();
		mesaj.setObiect(carte);
		return mesaj;
	}
```

Listare 9 Funcția getBooks()

Inițial se vor prelua din baza de date cărțile pentru pagina dorită, după care, dacă toate cărțile sunt disponibile, se preiau din fișierele salvate pe server, imaginile aferente cărților prin funcția *getFileFromServer*, care se poate vedea în imaginea de mai jos. După aceste operații, se creează un *Mesaj* în care se vor introduce cărțile.

```
  private byte[] getFileFromServer(String path_file)
  {
	byte[] data = null;
	Path path = Paths.get("Fisiere\\"+path_file);
	try {
		data = Files.readAllBytes(path);
    } catch (IOException ex) {

    }
	return data;
   }
```

Listare 10 Funcția getFileFromServer()

Pentru a încarca o carte pe server se folosește funcția uploadFile() (Listare 11) care primește ca parametru un obiect de tip Carte, populat în prealabil cu datele necesare. Se apelează funcția *uploadCarte* din *ManagerDb*, după care se salvează efectiv pe server cartea, plus imaginea aferentă cărții.

```
public void uploadFile(Object obiect) {
              Carte carte=(Carte)obiect;
              ManagerDb.getSession().uploadCarte(carte);
              System.out.println("Upload carte"+carte.getNume()+" cu id-
ul="+String.valueOf(carte.getID()));
              uploadOnServer(carte);
       }
```

Listare 11 Funcția uploadFile()

```
private void uploadOnServer(Carte carte)
{
       createDirector(String.valueOf(carte.getID()));
       uploadFileOnFolder(String.valueOf(carte.getID()),carte.getNume()+".pdf",
carte.getContinut());

       carte.setCale(String.valueOf(carte.getID())+"\\"+carte.getNume()+".pdf");

       uploadFileOnFolder(String.valueOf(carte.getID()),carte.getNume()+".jpg",
carte.getImagine());

       carte.setCale_img(String.valueOf(carte.getID())+"\\"+carte.getNume()+".jpg");
              ManagerDb.getSession().updateCarteCale(carte);
}
```

Listare 12 Funcția uploadOnServer()

Pentru a crea directoare și fișiere pe server, s-au folosit două funcții și anume: *createDirector* și *uploadFileOnFolder*. Prima funcție crează un folder. În cazul în care acesta nu exista, se va creea unul. A doua funcție, cea de încărcare a fișierelor, primește ca parametru numele directorului dorit, numele fișierului, dar și conținutul acestuia. Se instanțiază obiectul *File*. În cazul în care acesta nu există, se încearcă crearea lui, după care se începe scrierea conținutului.

```
private void createDirector(String nume)
{
        File file = new File("Fisiere\\"+nume);
        if (!file.exists()) {
            if (file.mkdir()) {
                System.out.println("Directory is created!");
```

```
            } else {
                System.out.println("Failed to create directory!");
            }
}
```

Listare 13 Funcția CreateDirector()

```
    private void uploadFileOnFolder(String director,String nume,byte[] continut) {
            File file = new File("Fisiere\\"+director+"\\"+nume);
            try {
                    if(!file.exists())
                    {
                            file.createNewFile();
                    }
                    FileOutputStream fisier = new FileOutputStream(file.getPath());
                    fisier.write(continut);
                    fisier.close();
            } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }
    }
```

Listare 14 Funcția uploadFileOnFolder()

Atribuirea unui sentiment

Pentru a putea atribui un sentiment unui cuvânt sau unei grupări de cuvinte, se va folosi clasa *ManagerSentiment* care va calcula pentru fiecare frază selectată un scor. Acesta va fi asociat cu un anumit sentiment dintr-o listă predefinită. Ca și bază de date pentru algoritm, se va folosi *SentiWordNet_3.0_20130122.txt*. Pentru a crea dicționarul de cuvinte cu scorurile aferente, se va folosi clasa *SentiWordNet*.

Prelucarea Notelor

Pentru a prelucra notele, s-a creat clasa *ManagerNote*, care are ca unic scop calcularea notelor în funcție de numărul de votanți, dar și de nota actuală.

Pentru a menține pe server unele date, dar și pentru schimbul de informații dintre server și client, s-au creat următoarele clase, cu următorul rol comun: în momentul creării unui mesaj spre a fi trimis către server, respectiv către client, acesta va conține un obiect din clasele *Carte, Grup, Nota, User*, în funcție de operațiunea dorită.

Client

Clientul este o aplicație ce rulează pe platforma Android. Modul de implementare a acestuia se poate vedea în paginile ce urmează.

Pentru a realiza conexiunea la server, dar și pentru a primi și trimite mesaje, s-a creat clasa *ConnectionManager* de tip *Singleton*. S-a folosit această metodă pentru a avea acces din orice punct al programului la conexiunea cu serverul. Modalitatea de a trimite și primi mesaje se poate vedea în Listare 15 și Listare 16. S-a folosit *InputStream*, respectiv *OutputStream* puse la dispoziție de *socket*-ul creat.

```
public Object sendMessage(Object obiect)
{
    Mesaj mesaj=null;
    if(isNetworkAvailable() && isConnected) {

        boolean test = false;
        try {
            out.write(TransformerBytes.getDataPacket(obiect));
            mesaj = readMesage();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return mesaj;
}
```

Listare 15 Funcția sendMessage()

Trimiterea unui mesaj este similară cu cea implementată în server și anume: se serializează obiectul creat și se transformă într-un vector de biți, după care se apelează funcția *readMesage()*. În funcția *getDataPacket()* se serializează obiectul și se inserează într-un vector de biți, în urmatorul mod: primul bit va fi cel de inițializare, după care se scrie lungimea mesajului și obiectul propriu zis. Între ultimele două este inserat un separator.

```
private Mesaj readMesage()
{
    Mesaj mesaj=null;
    byte[] initilize = new byte[1];
    try {
        din.read(initilize, 0, initilize.length);
        if (initilize[0] == 2) {
            byte[] recv_data = ReadStream();
            mesaj = (Mesaj) SerializationUtils.deserialize(recv_data);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return mesaj;
}
```

Listare 16 Read Message()

Autentificare

În momentul în care utilizatorul deschide aplicația se va încărca activitatea *acitivty_login*. Aceasta conține două câmpuri de tip *EditText,* pentru nume și parole, dar și două butoane, pentru autentificare, respectiv înregistrare. La inițializarea activității, în funcția *onCreate()* se face conexiunea la server. În cazul succesului aplicația va continua să ruleze normal. În caz contrar utilizatorul va fi restricționat și nu i se va permite accesul la operațiunea de autentificare.

După ce utilizatorul a introdus datele de autentificare se crează un mesaj de tip *Mesaj*, având setat un obiect de tip *User*. Acesta se trimite serverului prin intermediul clasei *ConnectionManager* și se asteaptă răspunsul. Dacă răspunsul este de „LOGIN_SUCCES”, atunci se crează o intenție nouă care va porni activitatea princială a aplicației, *MainActivity*. În caz contrar, utilizatorului i se va cere să reintroducă datele corecte sau să își facă un cont. Tot acest proces se realizează într-o funcție asincronă, pentru a nu bloca interfața.

Odată pornită activitatea principală a aplicației, aceasta va implementa un *NavigationView.OnNavigationItemSelectedListener,* care va crea meniul din partea stânga a ecranului. Acesta va fi accesibil utilizatorului prin intermediul butonului din partea stângă a ecranului, sau prin glisarea acestuia din partea stânga a ecranului. În această clasă se suprascrie

funcția *onNavigationItemSelectedListener*, astfel încât la selectarea unei opțiuni din meniu, *layoutul* activității curente va fi înlocuit cu cel al opțiunii selectate.

```
@SuppressWarnings("StatementWithEmptyBody")
@Override
public boolean onNavigationItemSelected(MenuItem item) {
    // Handle navigation view item clicks here.
    int id = item.getItemId();
    FragmentManager fragmentManager=getFragmentManager();

    if (id == R.id.nav_allbooks) {
        fragmentManager.beginTransaction().replace(R.id.content_main,new
AllBooks()).commit();
        // Handle the camera action
    } else if (id == R.id.nav_uploadbook) {
        fragmentManager.beginTransaction().replace(R.id.content_main,new
UploadBook()).commit();

    } else if (id == R.id.nav_yourprofile) {
        fragmentManager.beginTransaction().replace(R.id.content_main,new
YourProfile()).commit();
      }
    DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
    drawer.closeDrawer(GravityCompat.START);
    return true;
}
```

Listare 17 Funcția onNavigationItemSelected()

În continuare voi detalia unele dintre cele mai importante activități ale aplicației, care vor fi puse la dispoziția utilizatorului prin intermediul activității principale, adică *MainActivity*:

1) Vizualizare cărți - ajuns la acest ecran, se va porni activitatea *AllBooksActivity*, care va crea un *TabHost*. Acesta va fi împărțit în doua file (tab-uri): “Toate cărțile” și “Cărțile mele”.

```
        private void createTabHost()
        {
            TabHost tabhost =(TabHost)
        myView.findViewById(R.id.tabhost);
            tabhost.setup();
            TabHost.TabSpec allBooks=tabhost.newTabSpec("Toate
        cartile");
            allBooks.setContent(R.id.tab1);
            allBooks.setIndicator("Toate cartile");

            TabHost.TabSpec yourBooks=tabhost.newTabSpec("Cartile
        mele");
            yourBooks.setContent(R.id.tab2);
            yourBooks.setIndicator("Cartile tale");

            tabhost.addTab(allBooks);
            tabhost.addTab(yourBooks);
        }
```

Listare 18 Funcția createTabHost()

Pentru a popula cele două tab-uri se vor crea două variabile *m_adapterAllBooks*, de tipul *MyListAdaptorAllBooks* și *m_adapterMyBooks*, de tipul *MyListArrayAdapterMyBooks*.

```
private class MyListAdapterAllBooks extends ArrayAdapter<CartiRepository>
{
    public MyListAdapterAllBooks() {
        super(myView.getContext(), R.layout.item_view,
myCartiRepositories_allBooks);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent)
{
        // Make sure we have a view to work with (may have been given
null)
        View itemView = convertView;
        if (itemView == null) {
            itemView = myInflater.inflate(R.layout.item_view, parent,
false);
        }
        CartiRepository currentCartiRepository =
myCartiRepositories_allBooks.get(position);

        ImageView imageView =
(ImageView)itemView.findViewById(R.id.imagine_carte);
        imageView.setImageBitmap(currentCartiRepository.getImagine());

        TextView gen_carte = (TextView)
itemView.findViewById(R.id.gen_carte);
        gen_carte.setText("" + currentCartiRepository.getM_gen());
```

```
        TextView nota_carte = (TextView)
itemView.findViewById(R.id.nota_carte);
        nota_carte.setText("" + currentCartiRepository.getNota());

        TextView autor = (TextView)
itemView.findViewById(R.id.autor_carte);
        autor.setText("" + currentCartiRepository.getM_autor());

        TextView titlu_carte = (TextView)
itemView.findViewById(R.id.titlu_carte);
        titlu_carte.setText(currentCartiRepository.getM_nume());

        return itemView;
    }
}
```

Listare 19 Implementarea clasei MyListAdapterAllBooks()

Pentru a popula tabelul cu cărți, se apelează metoda *startGetAllBooks*, care crează un *task* asincron astfel încât sa nu se blocheze interfața. Când vizualizarea cărților de pe o pagină a ajuns la final, se va apela funcția *onScrollStateChanged*, care va extrage din baza de date următoarele cărți.

```
@Override
public void onScrollStateChanged(AbsListView view, int scrollState) {
    if (scrollState == AbsListView.OnScrollListener.SCROLL_STATE_IDLE
            && (m_listAllBooks.getLastVisiblePosition() -
m_listAllBooks.getHeaderViewsCount() -
            m_listAllBooks.getFooterViewsCount()) >=
(m_listAllBooks.getAdapter().getCount() - 1)) {
        if (!m_finishAllBooks) {
            m_paginaAllBooks++;
            startGetAllBooks();

        }
    }
}
```

Listare 20 Extragerea cărților din baza de date

2) Citește carte - în momentul în care utilizatorul vrea să citească o carte, se va încărca activitatea *ReadBookActivity*, care va pune la dispoziție o serie de butoane, cu ajutorul cărora se vor putea face următoarele operațiuni: căutare dupa un cuvânt, sublinierea

unui cuvânt, mărirea și micșorarea dimensiunii paginii. Tot aici utilizatorul poate face selecții pe text, astfel încât în urma apăsării butonului “Sentiment”, i se vor afișa o serie de sentimente specifice textului. Pentru încărcarea și afișarea conținutului cărții, se va folosi clasa *MuPDFReaderView*.

```
reader = new MuPDFReaderView(this)
{
    @Override
    protected void onMoveToChild(int i) {
        if (core == null)
            return;

        mPageNumberView.setText(String.format("%d / %d", i + 1,
                core.countPages()));
        mPageSlider.setMax((core.countPages() - 1) *
mPageSliderRes);
        mPageSlider.setProgress(i * mPageSliderRes);
        super.onMoveToChild(i);
    }
}
```

Listare 21 Afișarea conținutului cărții

Scenarii de utilizare

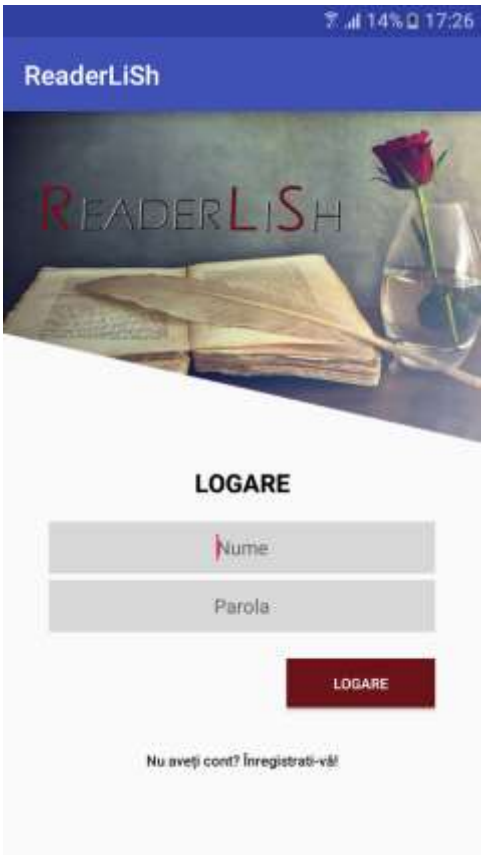
În acest subcapitol vor fi prezentate o parte dintre posibilele scenarii de utilizare ale aplicației cu imaginile aferente, pentru o mai bună înțelegere .

1. **Autentificarea**

Pentru a se realiza conexiunea la server, înainte de a intra în aplicație, utilizatorul va trebui să stabilească o conexiune la internet. În momentul în care se va intra în aplicație, acesta va fi întâmpinat de un ecran de autentificare. În partea de jos a ecranului va fi un link cu trimitere la pagina de înregistrare, în cazul în care nu aveți un cont deja făcut.

Pentru a putea intra în cont, utilizatorul va avea nevoie de numele de utilizator și de parola cu ajutorul cărora și-a creat contul. Atât în cazul în care se încearcă autentificarea fără unul, sau fără amble câmpuri, utilizatorul va primi un

mesaj specific. De asemenea acesta va fi atenționat și în cazul în care parola sau numele de utilizator sunt greșite.



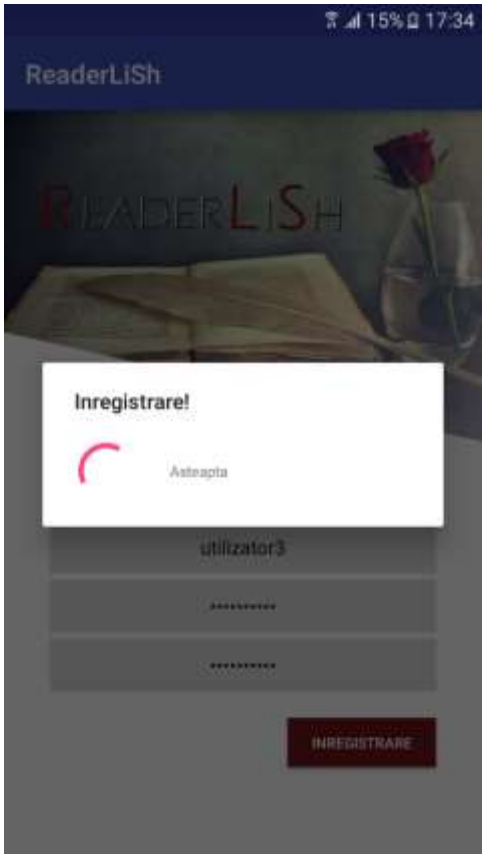
Figură 10 Autentificare

Figură 11 Autentificare

2. Înregistrarea

În cazul în care abia luați contact cu această aplicație, veți avea la dispoziție ecranul pentru înregistrare. Va trebui să se completeze câmpurile *Nume*, *Prenume*, *NumeUtilizator* și *Parolă*. În cazul în care unul, sau mai multe câmpuri nu sunt completate, utilizatorul nu va putea finaliza acțiunea de înregistrare, fiind atenționat asupra acestui aspect. O altă atenționare pe care utilizatorul o va primi, va fi în cazul în care parolele nu vor coincide.

Figură 12 Înregistrare



Figură 13 Înregistrare

3. Modificare profil

În momentul în care utilizatorul a reușit să treacă peste etapele anterioare, va avea posibilitatea de a-și complete integral profilul, alegând să își seteze o poză de profil. Tot în cadrul acestui ecran, utilizatorul va putea vedea grupurile din care face parte, precum și semnul care îl atenționează atunci când primește o notificare. În figurile de mai jos se pot vedea aceste lucruri:

Figură 14 Modificare profil

Figură 15 Modificare profil

4. Afișarea cărților mele

În această fereastră, utilizatorul va putea vedea cărțile pe care le-a încărcat. De asemenea va avea și posibilitatea de a căuta o carte prin intermediul opțiunii "Căutare".

Figură 16 Cărțile mele

5. Afișarea tuturor cărților

În această fereastră, utilizatorul va putea vizualiza toate cărțile disponibile atât din intermediul aplicației, cât și cele încărcate de utilizator. De asemenea va avea și posibilitatea de a căuta o carte prin intermediul opțiunii “Căutare”.

Figură 17 Toate cărțile

6. **Încărcarea unei cărți**

   În această fereastră utilizatorul va putea încarca o carte. Această opțiune devine activă doar în momentul în care utilizatorul se va afla în meniu. Acesta va trebui să completeze câmpurile cu *Nume, Autor, Descriere* și să selecteze genul cărții dintr-o listă cu mai multe opțiuni. De asemenea vor fi active două butoane care vor permite încărcarea unei imagini de copertă și încărcarea unui pdf. Abia după ce toate câmpurile au fost completate, utilizatorul va putea finaliza operațiunea de a încarca o carte.

Figură 18 Încarcă o carte

7. Vizualizare carte

În "Vizualizare Carte" utilizatorul va putea vizualiza informațiile despre o carte: nume, autor, notă și descriere.

În partea de jos a ecranului, utilizatorul va dispune de trei butoane cu următoarele funcționalități:

- "Citește cartea" - odată aleasă această funcționalitate, se va deschide modul vizualizare;
- "Vreau sa o citesc" – această funcționalitate îți permite să îți creezi o listă cu acele cărți pe care dorești să le citești pe viitor;
- "Recomandă" – această opțiune îți oferă posibilitatea de a recomanda cartea respectivă unui utilizator;

În Figură 19 este surprins acest moment:

Figură 19 Vizualizare carte

8. Citește carte

În această secțiune utilizatorul va putea vedea și citi conținutul pdf-ul. De asemenea vor fi active și anumite funcționalitați care pot fi făcute asupra textului: asocierea unui cuvant, sau a unor grupuri de cuvinte, cu un anume sentiment, căutare în text a unui cuvânt sau a unei fraze și de asemenea se poate face copierea unei selecții făcute în text. Acest lucru este vizibil în imaginea de mai jos, Figură 20.

Figură 20 Citește cartea

Configurare și cerințe

- Dispozitiv mobil cu sistem de operare Android incepand cu versiunea 5.0 (Lollipop);
- Buna dispoziție a utilizatorului pentru lectură;

Concluzii finale

Consider că aplicația ReaderLiSh - Read, Like & Share este o aplicație foarte utilă datorită facilităților pe care le oferă: căutarea unei cărți, recomandarea unei cărți unei persone, apartenența la un grup de lectură alături de persoane care împărtășesc aceleași preferințe în materie de lectură, posibilitatea de a localiza pe hartă, tot prin intermediul aplicației, librăriile și bibliotecile din zona în care te afli, sau o altă zonă setată de tine.

Direcții de viitor

Pe viitor, aplicația ar putea fi extinsă la nivel de funcționalități și de asemenea poate fi portată pe Web. În privința funcționalităților, dintre cele care pot fi adăugate ar fi:

- trimiterea mesajelor în timp real;
- un utilizator să poată primi recomandări pe baza cărților pe care le-a citit;

Bibliografie

[1] Simplenet, „Androidu,” Androidu.ro - Stiri, Device-uri, Review-uri si Aplicatii Android, 2017. [Interactiv]. Disponibil la: http://www.androidu.ro/despre-android-ce-este-android-totul-despre-android/.

[2] „Wikipedia,” Creative Commons cu atribuire și distribuire în condiții identice, [Interactiv]. Disponibil la: https://ro.wikipedia.org/wiki/SQLite.

[3] „Universitatea Tehnica Cluj Napoca,” [Interactiv]. Disponibil la: http://control.aut.utcluj.ro/scd/lab2/laborator2_1.htm.

[4] „MuPDF,” Artifex Software, Inc, 2006-2015. [Interactiv]. Disponibil la: http://mupdf.com/.

[5] „Facultatea de Informatica,” FII, 2006-2010. [Interactiv]. Disponibil la: http://profs.info.uaic.ro/~busaco/teach/courses/net/docs/threads/thread_creation.htm.

[6] „Eseu argumentativ tip SII Română,” Blogger, [Interactiv]. Disponibil la: http://www.eseuargumentativromana.com/2016/07/rolul-lecturii-in-dezvoltarea-personala.html.

[7] „Computer Science & Engineering Departament,” [Interactiv]. Disponibil la: http://ocw.cs.pub.ro/courses/eim/laboratoare/laborator01.

[8] „SentiWordNet,” 2013. [Interactiv]. Disponibil la: http://sentiwordnet.isti.cnr.it/code/SentiWordNetDemoCode.java.