UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENŢĂ

Recunoașterea optică a caracterelor pe platforma Android

propusă de

Petre Mihail Pohrib

Sesiunea: *februarie, 2017*

Coordonator ştiinţific

Asist. Dr. Vasile Alaiba

UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

FACULTATEA DE INFORMATICĂ

Recunoașterea optică a caracterelor pe platforma Android

Petre Mihail Pohrib

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 „*Recunoașterea optică a caracterelor pe platforma Android*” este scrisă de mine şi 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 şi 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, *data*

Absolvent *Petre Mihail Pohrib*

_____

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Recunoașterea optică a caracterelor pe platforma Android*”, codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

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

Iaşi, *data*

Absolvent *Petre Mihail Pohrib*

_____

Introducere

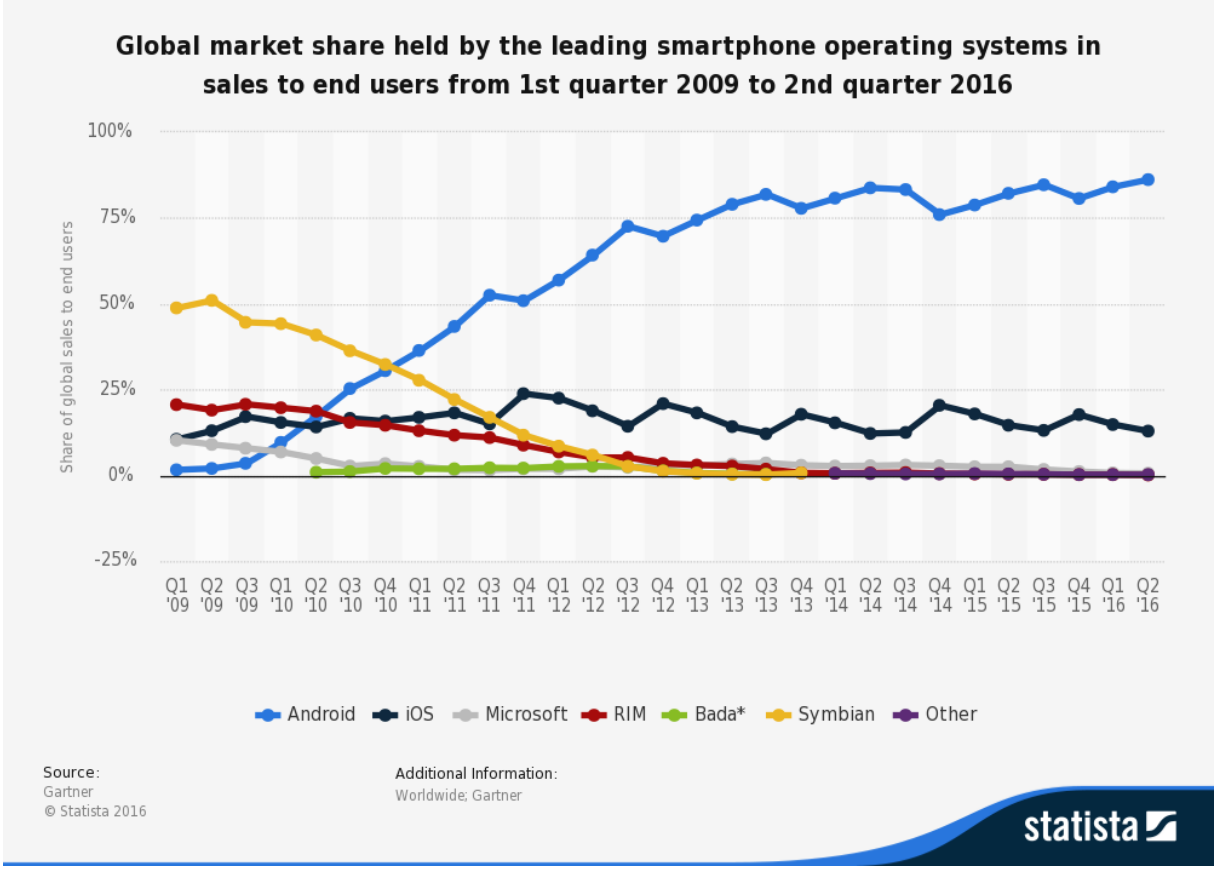
Android OCR (optical character recognition) este o aplicație care oferă utilizatorului posibilitatea de a obține text editabil dintr-o imagine care conține text (de exemplu o pagină scanată). Utilizatorul va putea salva rezultatul obținut într-un format care să permită stocarea, editarea și distribuirea acestuia.

Aplicația va fi disponibilă posesorilor de smartphone-uri cu sistemul de operare Android, versiunea Lollipop (5.0) sau mai recent în primă fază, dar portarea acesteia pe alte platforme va fi foarte facilă, întrucat aplicația folosește pentru partea de logică un API care poate fi apelat de pe orice platformă.

Motivație

Am ales tema „Recunoașterea optică a caracterelor pe platforma Android” pentru prezenta lucrare deoarece consider că tema este de actualitate și are o importantă utilitate în practică în contextul răspândirii telefoanelor inteligente și a disponibilității informației prin intermediul internetului.

Aplicația dorește să vină în ajutorul utilizatorului atunci când acesta are o imagine cu conținut scris și dorește să vină cu modificări la acesta sau să prelucreze textul respectiv. De exemplu, există o mulțime de articole pe internet cu conținut științific care se găsesc sub formă de pagini scanate. De obicei, scrisul este destul de mic și face parcurgerea acestui material destul de anevoioasă. Dar dacă am putea obține text editabil din aceste pagini scanate, am putea crește fontul, traduce anumite cuvinte și putem stoca mai ușor aceste materiale întrucât textul ocupă mai puțin spațiu decât o imagine. De aceea am decis să abordez această temă, a recunoașterii optice a caracterelor și am decis să creez o aplicație pentru cea mai populară platformă pentru dispozitivele de tip telefon inteligent (Statista, 2016).



Pentru a putea folosi această aplicație sunt necesare doar un telefon cu sistemul de operare Android și o conexiune la internet și eventualele programe necesare pentru vizualizarea fișierelor rezultate.

Context

Recunoașterea optică a caracterelor, adesea prescurtată OCR, este o ramură a informaticii care presupune recunoașterea textului scris din diverse tipuri de documente fizice, cum ar fi documente scanate, documente PDF sau imagini surprinse cu o cameră digitală, și convertirea acestuia într-un format care să permită unui sistem de calcul manipularea acestuia (Abbyy, 2016).

Recunoașterea optică a caracterelor este un domeniu intens cercetat deoarece această tehnologie are o aplicabilitate foarte mare în domenii cum ar fi domeniul poștal, cel economic și securitate. Printre alte aplicații se numără și tehnologiile de asistarea a persoanelor cu dizabilități ale aparatului vizual, conservarea documentelor tipărite în format digital,

interacțiunea cu calculatorul în timp real prin recunoașterea comenzilor scrise, catalogarea cărților din bibliotecă și aplicații în procesarea limbajului natural. Există o multitudine de articole care prezintă și încearcă să dezvolte această temă.

Marile companii din domeniul I.T. și dezvoltatorii de aplicații s-au implicat în crearea de aplicații care oferă facilități de recunoaștere a caracterelor. La o simplă căutare a cuvântului „OCR” pe motorul de căutare Google putem observa o multitudine de aplicații care au implementat această funcționalitate. Companii precum Microsoft și Google s-au implicat activ în acest domeniu și pun la dispoziție API-uri pentru realizarea procesului de recunoaștere optică a caracterelor.

Exemple de aplicații care profită de aceste API-uri sunt *Office Lens* de la Microsoft și *Google Keep*. Ambele aplicații oferă capabilități de recunoaștere a caracterelor.

În concluzie, tema este de actualitate, foarte utilă în viața de zi cu zi și de asemenea foarte interesantă.

Cerințe funcționale

1. Înregistrarea și autentificarea în aplicație – Utilizatorul se va înregistra în aplicație la prima utilizare a aplicației, sau dacă acesta are deja cont va trece direct la partea de logare, altfel acesta nu va putea folosi aplicația.
2. Managementul fotografiilor încărcate în aplicație – Fotografiile încărcate de utilizator vor fi salvate în aplicație, iar acesta va putea face managementul acestora.
3. Realizarea unei fotografii noi – Aplicația va permite realizarea fotografiilor în cadrul acesteia, nefiind nevoie de alte aplicații adiționale.
4. Explorarea memoriei telefonului pentru găsirea fotografiilor – Utilizatorul va putea explora memoria telefonului pentru găsirea fotografiilor și încărcarea acestora în aplicație.
5. Detectarea de text dintr-o fotografie – rezultatul procesului de recunoaștere optică a caracterelor va fi afișat direct pe ecranul dispozitivului, utilizatorul având posibilitatea să salveze rezultatul într-un fișier;
6. Administrarea utilizatorilor – utilizatorii cu rol de administrator vor putea face managementul utilizatorilor direct din aplicație;

Specificații tehnice

Aplicația de Android va fi dezvoltată în limbajul de programare Java, iar mediul de dezvoltarea va fi Android Studio.

Java este un limbaj de programare de nivel înalt, dezvoltat de JavaSoft, companie în cadrul firmei Sun Microsystems, complet orientat pe obiecte, concurent, robust și foarte sigur. Limbajul Java este foarte portabil deoarece codul este compilat într-un cod intermediar care este apoi interpretat de o mașină virtuală Java, de aceea nu trebuie recompilat pentru a funcționa pe alte platforme, fiind suficientă existența mediului de dezvoltare Java pe acea mașină (Frăsinaru, 2016).

Limbajul Java este o componentă vitală pentru sistemul de operare Android întrucât kitul de dezvoltare a aplicațiilor Android (Android SDK) folosește limbajul Java ca bază a aplicațiilor (Wikipedia, 2016).

Pentru comunicarea cu API-ul voi folosi biblioteca *Retrofit*, o bibliotecă open-source dezvoltată de „Square”. Retrofit transformă API-ul HTTP într-o interfață Java pentru a ușura comunicarea cu acesta. De asemenea, Retrofit face automat deserializarea obiectelor JSON venite de la API în obiecte Java (Retrofit, 2016).

Aplicația necesită versiunea de Android 5.0 (nivelul API 21) deoarece pentru realizarea fotografiilor am utilizat *camera2* API, introdus în versiunea de Android specificată mai sus.

Pentru afișarea interfeței am folosit fragmente întrucât acestea sunt mai rapide decât activitățile și sunt mai ușor de interschimbat, contribuind astfel la o experiență vizuală mai bună.

API-ul va fi scris în limbajul C# și folosește framework-ul ASP.NET Web API. Ca mediu de dezvoltare voi folosi Microsoft Visual Studio.

C# este un limbaj de programare de nivel înalt dezvoltat de Microsoft complet orientat pe obiecte, concurent și robust. Acesta permite dezvoltarea unei game largi de aplicație folosind framework-ul .NET (Microsoft, 2016). În curând limbajul va putea fi folosit pentru dezvoltarea de aplicații multi-platformă odată cu introducerea framework-ului .NET Core (Microsoft .NET Core, 2016).

Framework-ul ASP .NET Web Api oferă facilitatea de a dezvolta servicii web de tip „RESTful” folosind toate avantajele framework-ului .NET (Blog Web API, 2016).

API-ul va avea în spate o bază de date de tipul Microsoft SQL pentru salvarea datelor, iar managementul bazei de date va fi făcut folosind „Entity Framework” 6 Code First.

EntityFramework este după cum îi sugerează și numele un framework „Object/Relational Mapping”(ORM) care permite dezvoltatorilor să lucreze cu date relaționale ca obiecte specifice domeniului, eliminând nevoia de a scrie codul de acces la baza de date. Interogările se fac utilizând LINQ, obținând obiecte puternic tipizate. Folosind „Code First”, baza de date este creată pe baza obiectelor create în C# (Entity Framework, 2016).

Autentificarea utilizatorilor se va face folosind framework-ul *ASP.NET Identity 2.1* folosind middleware-ul *Owin,* implementarea Microsoft, *Katana*. Acesta oferă facilități de securitate foarte bune și este ușor de integrat în framework-ul ASP.NET Web API. Autentificarea va fi pe bază de jetoane. Clientul va face o cerere la API, acesta îi va valida identitatea și va emite un jeton care va însoți fiecare cerere la API și îi va confirma identitatea.

Pentru validarea datelor venite de la utilizator pe partea de server voi folosi biblioteca *FluentValidation*. *FluentValidation* este o bibliotecă mică pentru framework-ul .NET care folosește interfața fluent și expresiile lambda pentru validarea obiectelor venite de la clienți (Fluent Validation, 2016).

Pe partea de recunoaștere de caractere, pentru obținerea unor rezultate cât mai bune imaginile vor trece printr-o etapă de preprocesare care cuprinde: binarizarea (metoda Otsu), detectarea unghiului la care este înclinată imaginea și corectarea acestuia (transformarea Hough) și eliminarea zgomotului.

Mă voi folosi și de *Emgu CV* pentru a procesa imaginea și obținerea zonelor în care se află textul. *Emgu CV* este un decorator peste *OpenCV* care permite apelarea funcțiilor OpenCV în limbaje compatibile .NET.

Detectarea propriu-zisă a caracterelor se va face utilizând biblioteca *Tesseract*. *Tesseract* este sponsorizat și dezvoltat de cei de la Google încă din anul 2006 și este considerat unul dintre cele mai bune motoare de recunoaștere optică a caracterelor disponibil gratis.

Rezultatul obținut după procesarea imaginii folosind *Tesseract* va fi trecut printr-o fază de corectarea a textului utilizând Bing Spell Check API de la Microsoft.

Alte tehnologii pe care le voi folosi vor fi *Autofac* pentru „dependecy injection”, *Automapper* pentru maparea obiectelor de transport(DTO) la entitățile necesare pentru baza de date.

Autofac este un container pentru inversarea controlului folosit la injectarea dependințelor, iar Automapper este un mapper pentru obiecte bazat pe convenție.

Ca sabloane de proiectare, voi folosi *CQRS* (command and query responsability segregation) și *Repository*.

Contribuții

Prezenta lucrare va fi împărțită în două capitole în care voi prezenta, pe rând, procesul de dezvoltare a celor două părți ale aplicației, respectiv partea de client și partea de server, oferind detalii de implementare la soluțiile alese:

1. Capitolul 1: Dezvoltarea aplicației de server (API)
2. Capitolul 2: Dezvoltarea aplicației de Android

Dezvoltarea aplicației de server a avut ca scop obținerea unei aplicații ușor de utilizat și de întreținut, eficientă și prin intermediul căreia, ca și dezvoltator, să am ce învăța. De aceea în cadrul aplicației de server am utilizat cele mai noi tehnologii și paradigme de programare. Astfel am ajuns la o implementare ce utilizează:

1. Un framework modern și matur, ASP .NET Web API;
2. O arhitectură pe 3 straturi;
3. Utilizarea șablonelor de proiectare;
4. Utilizarea principiului de programare „dependency injection”;
5. Utilizarea unui ORM pentru accesul și crearea bazei de date;

Astfel, aplicația de server (API), este o aplicație ușor de folosit, foarte modernă prin intermediul tehnologiilor și paradigmelor de programare utilizate și foarte eficientă.

Dezvoltarea aplicației client pe platforma Android a avut ca scop obținerea unei aplicații eficiente și rapide. Astfel am utilizat metode pe care le consider benefice:

1. Utilizarea fragmentelor pentru interfața grafică;
2. Utilizarea librăriei „Retrofit” pentru apelarea serverului;
3. Implementarea unui interceptor pentru autentificare;
4. Implementarea unei camere de fotografiat pentru un spor de viteză;
5. Implementarea unui vizualizator de fotografii;

Cuprins

Capitolul 1: Dezvoltarea aplicației de server (API)

Am decis ca aplicația de server să fie foarte ușor de folosit pentru o multitudine de dispozitive și platforme, de aceea am scris aplicația de server ca un API. Pentru a realiza acest lucru am decis să folosesc soluția propusă de Microsoft și anume framework-ul .NET Web API. Operațiile suportate de framework sunt operațiile standard din cadrul protocolului HTTP și anume:

- GET – operațiune de returnare a datelor;
- POST – operațiune de creare a datelor;
- PUT – operațiune de actualizare a datelor;
- DELETE – operațiune de ștergere a datelor;

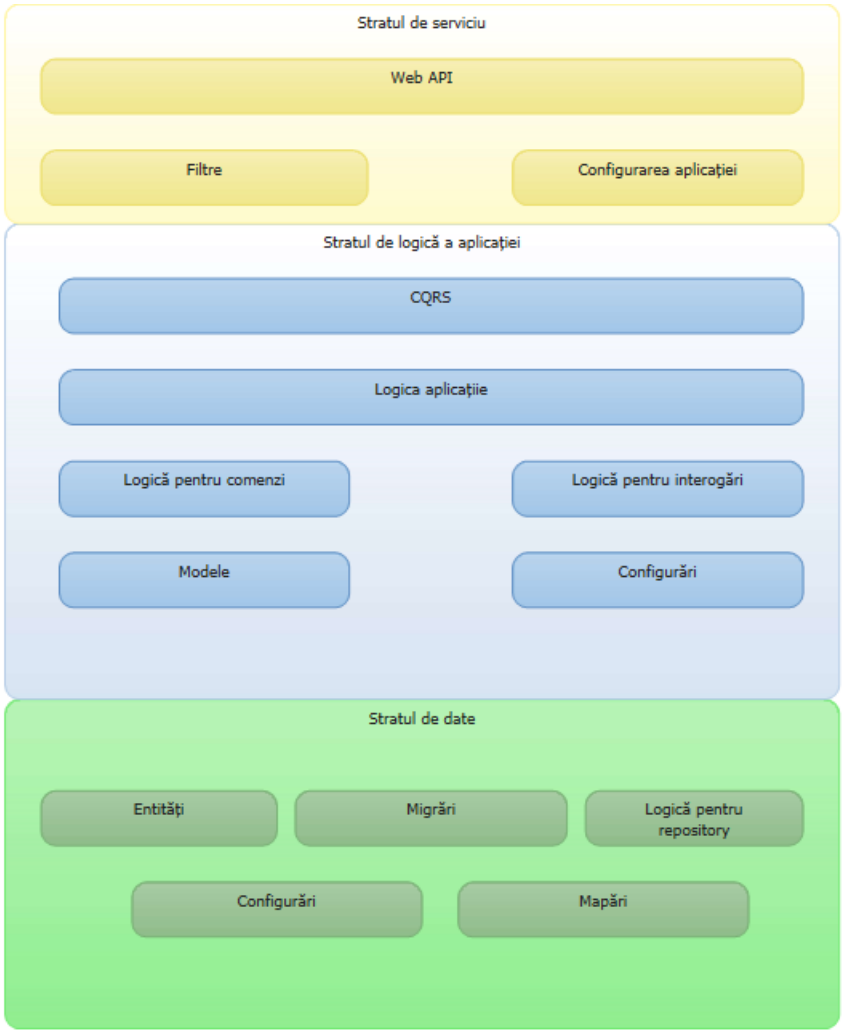
Utilizând framework-ul .NET Web API, limbajul de programare utilizat trebuie să fie un limbaj compatibil cu mediul de programare .NET. De aceea am decis să folosesc limbajul de programare C#. Dar înainte să vorbim despre limbajul de operare C#, trebuie să dăm câteva detalii despre framework-ul .NET.

Un framework este o colecție de intefețe de programare de aplicații (API) și o bibliotecă de cod partajat pe care dezvoltatorii o pot apela atunci când dezvoltă aplicații astfel încât să nu fie nevoiți să scrie codul de la zero. Dar framework-ul .NET este mai mult decât cele spuse mai sus încât .NET oferă și un mudiu de rulare pentru aplicații, adică aplicațiile rulează într-un fel de de mașină virtuală contribuind astfel la securitatea sistemului.

Limbajul de programare C# este un libaj de programare oferit de Microsoft, fiind un limbaj conceput pentru a fi pe deplin compatibil cu inițiativa .NET, profitând de ceea ce a învățat de la limbajele de programare cum ar fi C, C++ și Java. Limbajul de programare C# este proiectat pentru a fi un limbaj independent de platformă în spiritul Java, deși este implementat în special pentru platforma Windows. Acesta are o sintaxă similară cu sintaxa C, C++ și este proiectat pentru a fi un limbaj orientat pe obiecte. Există diferențe minore între sintaxa C++ și sintaxa C#. Similar cu Java, C# nu suportă moștenirea multiplă, în schimb oferă soluția oferită de limbajul Java: interfețele. Intefețele implementate de o clasă specifică anumite funcții pe care clasa este obligată să le implementeze. Intefețele impiedică pericolele moștenirii multiple menținând în același timp capacitatea de a lăsa mai multe clase să implementeze aceeași mulțime de metode.

Partea de server a fost scrisă utilizând mediul de dezvoltarea Microsoft Visual Studio 2015 . Microsoft Visual Studio este un mediu de dezvoltare integrat (IDE) oferit de către Microsoft. Este folosit pentru a dezvolta programe de calculator pentru Microsoft Windows, precum și de site-uri web. Visual Studio utilizează platforme de dezvoltare de software Microsoft cum ar fi Windows API, Windows Forms, Windows Presentation Fundation, Windows Store și altele. Visual Studio include un editor de cod, suportând IntelliSense (componentă de completare a codului) precum și refactorizare de cod. Depanatorul integrat (debugger) funcționează atât ca depanator la nivel de cod sursă și ca depanator la nivel mașină.

1.1 Arhitectura generală a aplicației



Figură 1 Arhitectură API

Aplicația folosește o arhitecture „3-tier” (arhitectură pe 3 niveluri). Separarea aplicației în aceste niveluri are ca scop creșterea mentenabilității și de asemenea creșterea scalabilității. Aceste avantaje rezultă din ușurința prin care se poate adopta o tehnologie nouă, fiind nevoie doar de modificarea unui singur nivel, nefiind nevoie de modificarea întregii aplicații. În figura 1 putem observa împărțirea aplicației în mai multe niveluri și nivelurile propriu-zise. În continuare vom detalia fiecare nivel și vom discuta despre responsabilitățile acestora.

1.2 Nivelul de service

Nivelul de service din cadrul aplicației de server se ocupă cu configurarea aplicației și cu modul în care utilizatorii interacționează cu serverul. În cadrul acestui nivel am configurat modul în care aplicația se ocupă de rezolvarea dependințelor, adică de configurarea librăriei „Autofac”, am configurat ruta implicită din aplicație, am creat un filtru personalizat pentru interceptarea mesajelor de validare a modelelor provenite din librăria „*Fluent Validation*” și cel mai important avem „controllerele” care se ocupă de interacțiunea server client. De asemenea în acest nivel avem șirul de caractere folosit de aplicație pentru a se conecta la baza de date. Vom vorbi în continuare mai detaliat despre fiecare configurare în parte.

1.2.1 Configurarea API-ului

În primul rând, în cadrul API-ului am definit o rută implicită. Aceasta este de forma „api/{controller}/{id}”. Astfel, toate cererile către server vor respecata această rută. Setarea rutei se face prin setarea obiectului *HttpConfiguration*.

```
config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

var jsonFormatter =
        config.Formatters.OfType<JsonMediaTypeFormatter>().First();
jsonFormatter.SerializerSettings.ContractResolver =
        new CamelCasePropertyNamesContractResolver();
```

În fragmentul de cod de mai sus putem observa crearea unei rute cu numele „DefaultApi” cu forma specificată mai sus. config este un obiect de tip *HttpConfiguration*.

De asemenea am configurat modul în care API-ul se așteaptă să primească și modul în care returnează obiectele de tip JSON[1]. Astfel am configurat API-ul să accepte numele obiectelor în notația *CamelCase*[2]. Această notație presupune ca prima literă din fiecare cuvânt să fie scrisă cu majusculă. Majoritatea limbajelor de programare folosesc acest tip de notație a numelui obiectelor și variabilelor.

În acest modul am configurat librăria *Autofac*. Astfel, am suprascris obiectul *DependecyResolver* din obiectul *HttpConfiguration* cu un obiect de tipul *AutofacWebApiDependencyResolver*. Obiectul cel din urmă conține un dicționar cu tipurile de obiecte pe care le poate construi.

```
public static void Init(HttpConfiguration config)
{
        var builder = new ContainerBuilder();

        builder.RegisterModule(new BusinessAutofacModule());

        builder.RegisterApiControllers(Assembly.GetExecutingAssembly());

        var container = builder.Build();

        config.DependencyResolver =
                new AutofacWebApiDependencyResolver(container);
}
```

Despre rezolvarea dependințelor și obiectul *BusinessAutofacModule* vom vorbi mai în detaliu atunci când vom discuta despre nivelul de business.

1.2.2 Controllere

Controllerele pentru framework-ul Web API deriva din clasa *ApiController*. Pentru a folosi șablonul arhitectural CQRS[3] am creat un controller de baza care moștenește din clasa *ApiController* și care are ca parametri pentru contructor un *CommandDispatcher* (dispecer de comenzi) și un *QueryDispatcher* (dispecer de interogari) și care inițializează două proprietăți ale controllerlui de baza. Despre CQRS și modul de funcționare vom vorbi în detaliu la nivelul de business.

[1] JSON – JavaScript Object Notation – tip de date asemănător cu XML, al cărui stil de scriere se aseamănă cu notația obiectelor de tip Array in JavaScript

[2] http://wiki.c2.com/?CamelCase

[3] CQRS – Command and query responsability segregation – șablon arhitectural http://martinfowler.com/bliki/CQRS.html

```
public class BaseController : ApiController
    {
        public BaseController(ICommandDispatcher commandDispatcher,
                 IQueryDispatcher queryDispatcher)
        {
            CommandDispatcher = commandDispatcher;
            QueryDispatcher = queryDispatcher;
        }

        protected ICommandDispatcher CommandDispatcher { get; }

        protected IQueryDispatcher QueryDispatcher { get; }
    }
```

Fiecare controller din aplicație va moșteni controllerul de bază de mai sus și va avea acces la proprietățile *CommandDispatcher*, resptectiv *QueryDispatcher*, care vor fi folosite de controllere pentru a utiliza logica aplicației. Putem obseva din codul sursă de mai sus că obictele nu depind de implementări concrete, depinzând de abstracțiuni, *ICommnadDispatcher* și *IQueryDispatcher* fiind doar interfețe, implementările concrete fiind injectate, aceste fiind un principiu al paradigmei de programare „inversarea controlului[4]". Utilizând aceste proprități, metodele din controller sunt foarte simple, având în cele mai multe cazuri 2-3 linii de cod.

```
[FluentValidation]
public IHttpActionResult Add(AddUserCommand command)
{
        CommandDispatcher.Dispatch(command);
        return Ok();
}
```

Putem observa din secvența de cod de mai sus că metoda din controller este foarte mică, având doar două linii, apelarea dispecerului de comenzi și returnarea unui rezultat la client, respectiv codul 200 (OK).

Mai putem observa în secvența de cod de mai sus și filtrul folosit la validarea modelelor. Acesta este folosit pentru a recupera erorile de validare returnate de biblioteca *Fluent Validation.* Acest filtru verifică modelul înaintea execuției codului din controller și dacă acesta conține erori atunci execuția codului este oprită. Mesajele de validare sunt întoarse la client în mod automat astfel încât aceștia să modifice datele astfel încât să corespundă cerințelor date.

Obiectele care se ocupă cu validarea modelelor se află în stratul de business și vom discuta despre acestea la subcapitolul dedicat stratului de business.

[4] Inversion of control – http://www.laputan.org/drc/drc.html

1.2.3 Securizarea serverului

Serverul nostru care se ocupă cu operațiile de OCR și operațiile adiționale de management al utilizatorilor și al conținutului este securizat, astfel încât conținutul și datele utilizatorului sunt în siguranță în fața unui atac. Pentru securizarea am ales să construiesc serverul peste middleware-ul Owin[5] și nu direct peste ASP.NET pentru că voi configura serverul astfel încât acesta să emită jetoane de autentificare OAuth folosind middleware-ul menționat main sus, deci configurarea tuturor componentelor pe aceași tehnologie este cea mai bună abordare. În plus vom folosi sistemul ASP.NET Identity, care este creat peste middleware-ul Owin și vom folosi această tehnologie pentru înregistrarea de noi utilizatori și de validarea datelor de autentificare are utilizatorilor deja înregistrați pentru emitererea de jetoane de autentificare.

Trebuie să menționez că serverul nostru trebuie să fie capabil să accepte solicitări de la orice fel de client, de aceea trebuie să permitem CORS[6] pentru server cât și pentru furnizorul de jetoane OAuth.

CORS este un mecanism care permite accesarea resurselor web restricționate dintr-un alt domeniu față de cel curent.

Dar înainte să înaintăm cu descrierea modului de funționare a tehnologiilor specificate mai sus voi vorbi un pic despre aceste tehnologii.

Owin definește o interfață standard între serverele web scrise în .NET și aplicațiile web. Scopul interfeței Owin este de a decupla serverul și aplicațiile, să încurajeze dezvoltarea de module simple pentru dezvoltarea web în .NET și fiind un standard deschis să stimuleze dezvoltarea ecosistemului .NET pentru dezvolatere web. Owin a fost dezvoltat astfel încât să rupă cuplajul strâns între ASP.NET și IIS[7] (Internet Information Services), serverul web creat de Microsoft pentru dezvoltarea aplicațiilor web. Implementarea Microsoft al acestui framework se numește *Katana*[8] și este deschis pentru dezvoltatori pe site-ul *CodePlex.*

OAuth[9] este un standard deschis pentru autorizare, utilizat în mod obișnuit de utilizatorii de internet pentru a autoriza site-urile web sau aplicațiile să accesze informațiile acestora dar fără a le oferi parolele. Acest mecanism este utilizat de site-uri precum Google, Facebook,

[5] Owin - http://owin.org/
[6] Crosso Origin Resource Sharing – https://www.w3.org/TR/cors/
[7] Interner Information Services – https://www.iis.net/
[8] Implementarea Microsft a standardului Owin - http://katanaproject.codeplex.com/documentation
[9] OAuth – https://oauth.net/

Microsft, Twitter, etc pentru a permite utilizatorilor să facă schimb de informații cu privire la conturile lor cu aplicații sau site-uri terțe (Joudeh, 2014).

Pentru implementarea sistemului de autorizare descris mai sus am fost nevoit să import utilizând managerul de pakete Nuget câteva pachete necesare pentru funcționarea sitemului. Acestea au fost :

- Microsft.AspNet.WebApi.Owin
- Microsoft.Owin.Host.SystemWeb
- Microsoft.AspNet.Identity.Owin
- Microsoft.AspNet.Identity.EntityFramework
- Microsoft.AspNet.Security.OAuth
- Microsoft.Owin.Cors

Aceste pachete sunt necesare pentru a permite serverului nostru Owin să ruleze în cadrul IIS utilizând sistemul de cereri ASP.NET. De asemenea trebuie să adăugăm suport pentru folosirea ASP .NET Identity și integrarea acestuia cu *Entity Framework.* Prin pachetul Microsoft.Owin.Cors vom permite CORS pe serverul nostru.

În procesul de implementarea al acestui sistem am renunțat la funcționalitățile clasei *Global.asax.* Aceasta a fost înlocuită cu o clasă nouă, numită *Startup.cs* care se va ocupa de funcționarea API-ului și în care vom configura sistemul de autentificare.

În cadrul clasei *Startup.cs* vom configura modul în care se comportă API-ul și vom inițializa sistemul de securitate.

Clasa de mai jos va fi inițializată atunci când aplicația noastră va porni. Putem observa adnotarea „assembly” care indică ce clasă va rula atunci când aplicația pornește. Metoda *Configuration* acceptă un parametru de tipul *IAppBuilder* acest paramtru fiind furnizat de gazdă la run-time. Parametrul *app* de tipul *IAppBuilder* este o interfață care va fi utilizată pentru a compune aplicația pentru serverul nostru Owin. Obiectul *HttpConfiguration* este utilizat pentru configurarea rutelor din cadrul API-ului așa că îl vom trimite ca parametru metodei *Register* discutate mai sus din clasa *WebApiConfig*.

În cele din urmă vom trimite ca parametru obictul *HttpConfiguration* metodei *UseWebApi* care va fi responsabilă de integrarea Web API-ului în serverul nostru Owin.

```
[assembly: OwinStartup(typeof(OcrApi.Startup))]
namespace OcrApi
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.UseCors(Microsoft.Owin.Cors.CorsOptions.AllowAll);

            ConfigureOAuth(app);
            var config = new HttpConfiguration();
            WebApiConfig.Register(config);
            app.UseWebApi(config);
            AutofacConfig.Init(config);
        }

        public void ConfigureOAuth(IAppBuilder app)
        {
            var oAuthServerOptions = new OAuthAuthorizationServerOptions()
            {
                AllowInsecureHttp = true,
                TokenEndpointPath = new PathString("/token"),
                AccessTokenExpireTimeSpan = TimeSpan.FromDays(1),
                Provider = new SimpleAuthorizationServerProvider()
            };

            app.UseOAuthAuthorizationServer(oAuthServerOptions);
            app.UseOAuthBearerAuthentication(new OAuthBearerAuthenticationOptions(
));
        }
    }
}
```

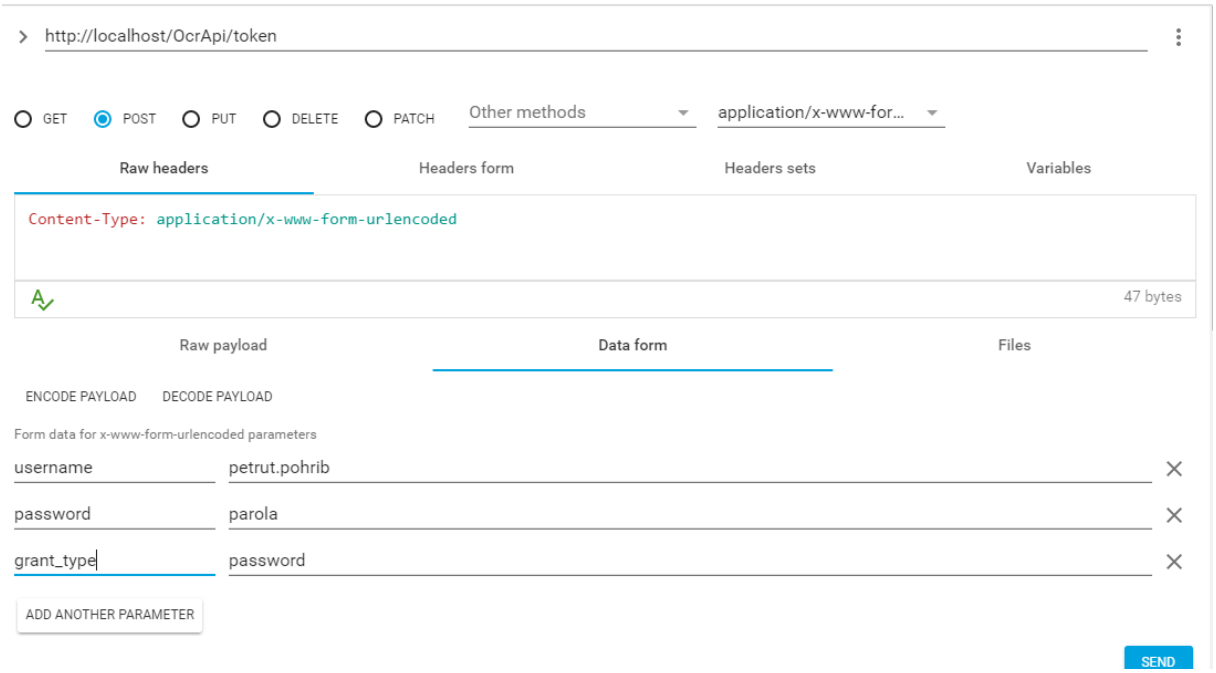
Pentru a utiliza fluxul de lucru pentru autentificare OAuth vom folosi metoda *ConfigureOAuth*, metodă apelată din metoda *Configuration.* În această metodă am creat o nouă instanță a clasei *OAuthAuthorizationServerOptions* în care am setat următoarele opțiuni:

- Calea unde jetoanele de autentificare vor fi generate, respectiv calea serverului urmată de */token*. Aici vom genera o cerere de tip *POST* pentru a genera un jeton;
- Am specificat durata de valabilitate a jetonului de autentificare și anume 24 de ore , astfel încât dacă un utilizator încearcă să folosească un jeton expirat, cererea lui va fi respină și va primi codul HTTP 401 (Unauthorized);
- Am specificat implementarea modului în care identitatea utilizatorului este validată, prin intermediul clasei *SimpleAuthorizationServerProvider*;

De asemenea, putem observa pe prima linie din metoda *Configuration* ca API-ul este configurat să accepte CORS.

Clasa care validează identitatea utilizatorului se află în stratul de business, dar o să vă descriu modul de funcționare în cadrul acestui strat, deoarece face parte dintr-un sistem unitar. Clasa care se ocupă de validare se numește *SimpleAuthorizationServerProvider* și moștenește din clasa *OAuthAuthorizationServerProvider* din care am suprascris două metode, *ValidateClientAuthentication* și *GrantResourceOwnerCredentials*. Prima metodă este responsabilă de validarea clientului, iar a doua este responsabilă de validarea datelor de autentificare. Dacă datele livrate sunt valide atunci vom emite un jeton în care vom adăuga niște date suplimentare pe care le vom folosi în logica aplicației, anume numele de utilizator și rolul utilizatorului.

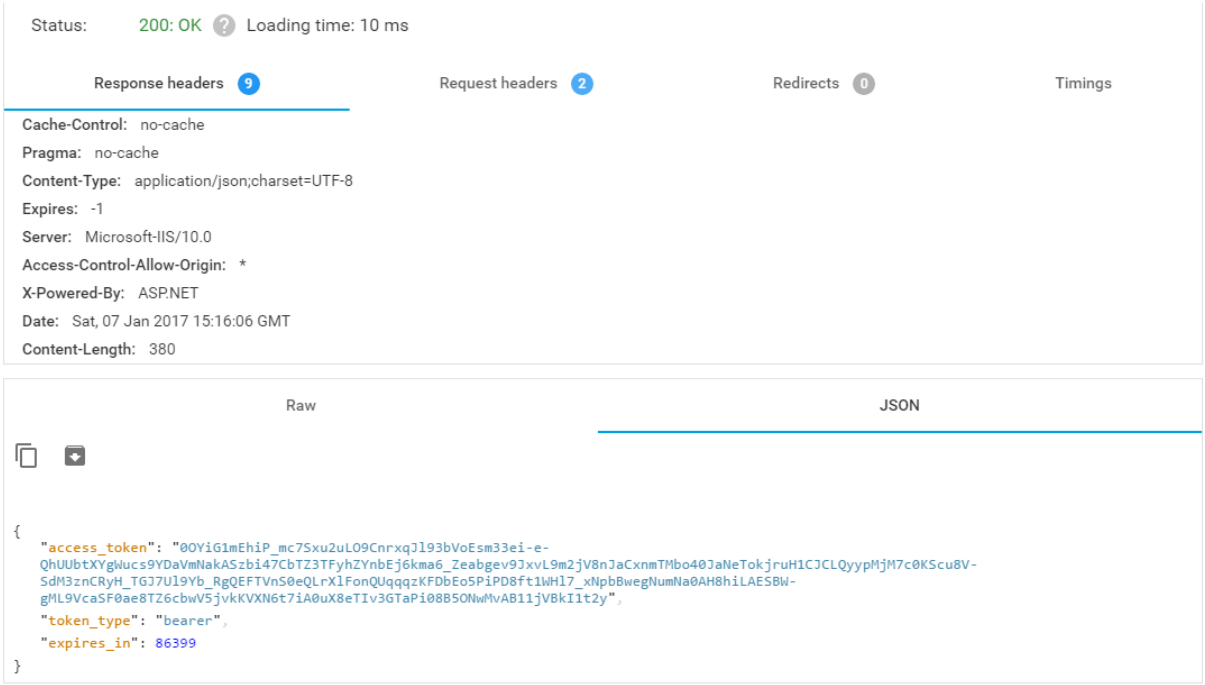
Pentru testarea mecanismului de autentificare o să testăm cu un nume de utilizator și o parolă. Pentru testarea API-ului voi folosi aplicația *Advanced Rest Console*.



După cum putem observa, se face o cerere de tip *POST* la adresa specificată de noi în setarea API-ului (/token). Observăm tipul conținutului, anume *x-www-form-encoded*, astfel corpul cererii va fi de forma

username=petrut.pohrib&password=parola&grant_type=password.

De asemenea, putem observa și tipul cererii de acordare a jetonului, adică o cerere pentru o parolă. Dacă totul funcționează cum trebuie atunci vom primi un jeton pe care îl vom utiliza în aplicație.



Jetonul obținut va fi utilizat în antetul cererilor la server, anume în câmpul *Authorization.* Jetonul o să fie prefixat de șirul de caractere „Bearer ” astfel făcându-se autorizarea pe server.

1.2.4 Detalii de conectare

Mai este notabil în acest nivel faptul că aici se configurează modul în care *Entity Framework* se conectează la baza de date. Acest șir de caractere se află în fișierul *web.config.* Acesta este principalul fișier de configurare al unei aplicații web care utilizează framework-ul ASP.NET. Acest fișier este un document XML care conține date referitoare la modul în care aplicația va acționa.

Șirul de conectare se află în secțiunea *configurations* într-o secțiune specială intitulată *connectionStrings* și are forma

```
    <add name="OcrApi" connectionString="Data Source=localhost\SQLEXPRESS;Initial
Catalog=OcrApi;Integrated Security=SSPI;" providerName="System.Data.SqlClient" />
```

Acest șir de caractere conține date referitoare la modul în care se face autentificarea, numele bazei de data la care se va conecta, locația bazei de data.

1.3 Nivelul business

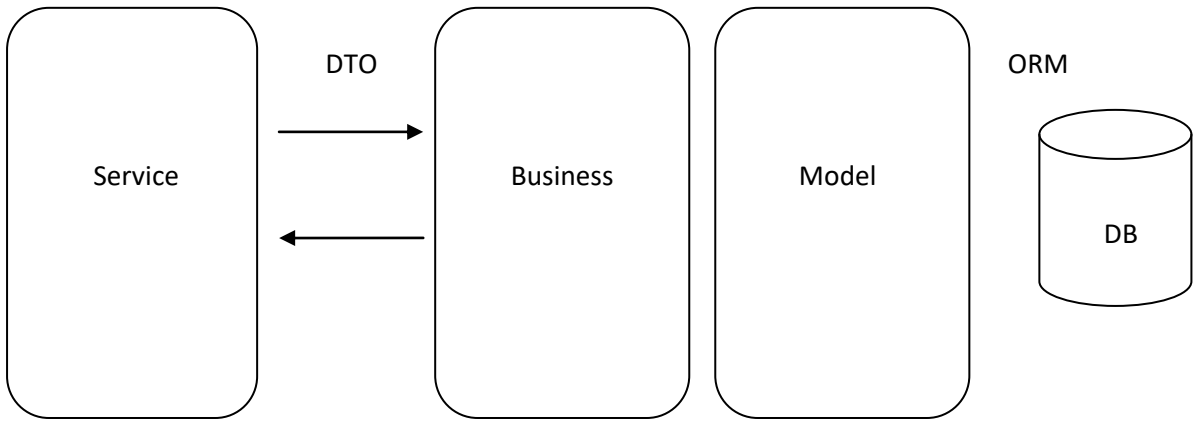
Nivelul de business se ocupă cu logica aplicației, aici aflându-se toate elementele de algoritmică și aici se găsesc o multitudine de configurări și tot aici avem logica care se ocupă de validarea modelelor care vin la nivelul superior, logica de validare a datelor de autentificare. Acest nivel logic mai conține un substrat, stratul de CQRS, care se ocupă de comunicarea dintre nivelul superior și acest nivel. În acest nivel vom vorbi și de modul în care aplicația realizează injectarea dependințelor.

1.3.1 CQRS (Command and Query Responsability Segregation)

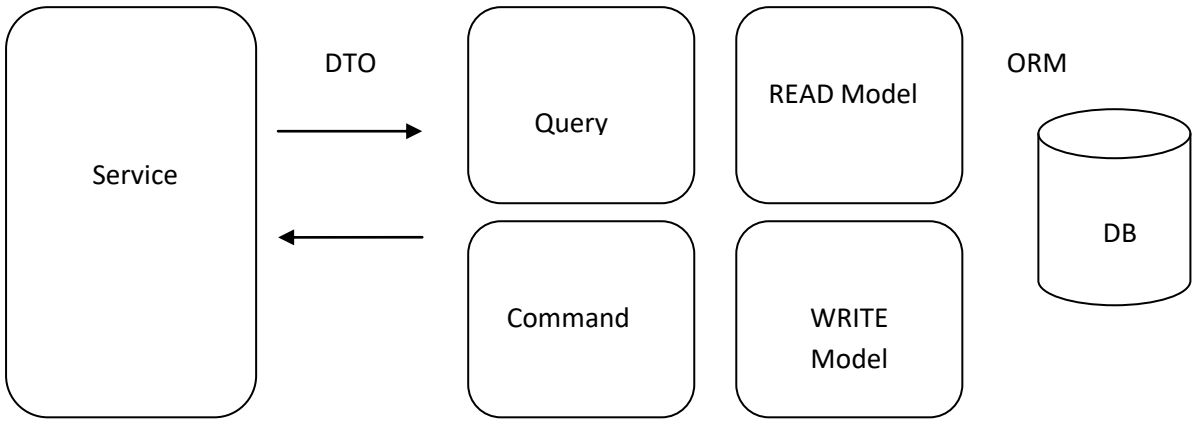
CQRS – segregarea responsabilităților comenzilor și interogărilor – este un șablon arhitectural ce își are rădăcinile în principiul software CQS – command-query serparation, mai bine spus este o implementare al acestui principiu. Acest principiu spune că fiecare metodă trebuie să fie o comandă sau o interogare, dar niciodată ambele. Am vorbit despre comenzi și interogări dar nu am definit aceste concepte. În continuare voi explica aceste concepte și voi explica modul în care funcșioneză CQRS și modul în care acesta a ost implementat în proiect (Stasch, 2015).

O comanda este singura modalitate prin care un sistem își poate schimba starea. Comenzile sunt responsabile pentru introducerea schimbărilor în sistem. Comenzile nu trebuie să returneze niciun rezultat.

O interogare poate fi asemănată cu o operație de citire. Acestea citesc starea sistemului, filtrează, agregă și transformă datele în cel mai util format. Poate fi executată de mai multe ori și acesta nu va modifica starea sitemului.



Mai sus putem observa modul în care aplicația noastră multi-layer ar arăta fără substratul de CQRS. Așa arată majoritatea aplicațiilor cu mai multe straturi, doar că în locul stratului de service avem un strat care se ocupă cu intefața grafică cu utilizatorul. Cu introducerea șablonului arhitectural CQRS aplicația are o altă așezare pe care o voi decrie în schema următoare.



Acum modelule de citire și scriere sunt separate doar la nivel logic, pasul următor pentru respectarea pe deplin al șablonului de proiectare CQRS fiind folosirea a două baze de data, una pentru citire și una pentru scriere. Dar o astfel de arhitectură ar introduce un nivel sporit de dificultate, ceea ce pentru a arhitectură mică ca cea din proiectul acesta nu ar aduce o valoare sporită. De aceea am ales ca pentru proiectul acesta să folosesc o arhitectură CQRS un pic modificată, singura variație fiind folosirea unei singure baze de date.

Am ales să implementez acest șablon de proiectare în aplicația noastră deoarece acesta aduce un plus când vine vorba de întreținerea aplicației, fiind foarte simplu de înțeles cum se apelează logica aplicației și permite îmbunătățirea acesteia, șablonul de proiectare fiind foarte ușor extensibil.

1.3.1.1 Implementarea comenzilor

Pentru modelarea comenzilor, descrierea mediului de execuție și modului în care acestea vor fi invocate am implementat o serie de trei interfețe pe care le voi detalia în cele ce urmează.

```
public interface ICommand
{
}
```

Avem interfața *ICommand* descrisă mai sus. Aceasta este o interfață *marker* care care îi sugerează dezvoltatorului că o clasă care implementează această interfață poate fi folosită ca și comandă.

```
public interface ICommandHandler<in TCommand> where TCommand : ICommand
{
    void Execute(TCommand command);
}
```

Interfața *ICommandHandler* descrisă mai sus are o metodă care primește ca parametru un obiect de tipul *ICommand* și este metoda responsabilă de executarea comenzii, după cum îi sugerează și numele. O clasă care va implementa această interfață va fi capabilă să-și rezolve dependințele utilizând un container de injectare de dependințe și este și ușor de testat întrucât nu depinde de implementări concrete.

```
public interface ICommandDispatcher
{
    void Dispatch<TCommand>(TCommand command) where TCommand : ICommand;
}
```

Interfața *ICommandDispatcher* va fi implementată de o clasă care se va ocupa de instanțierea manipulatorului de comenzi (en. command handler) potrivit.

În continuare voi prezenta implementarea dispatcherului de comenzi, clasa care implementează interfața *ICommandDispatcher*. Aceasta este clasa pe care controllerele din stratul de service o folosesc pentru a executa comenzile provenite de la utilizator.

Singura responsabilitate al acestei clase este de a găsi manipulatorul (en. handler) de comenzi potrivit. Obiectul de tipul *IComponentContext* este, după cum îi sugerează și numele, contextul containerul pentru injectarea dependințelor, container provenit din librăria *Autofac.* Utilizarea unui astfel de container ușurează mult implementarea șablonului de proiectare CQRS, nefiind nevoie să înregistrăm toate comenzile cu manipulatorul său și să instanțiem manual fiecare obiect.

```
public class CommandDispatcher : ICommandDispatcher
{
    private readonly IComponentContext _autofacContainer;

    public CommandDispatcher(IComponentContext autofacContainer)
    {
        _autofacContainer = autofacContainer;
    }
    public void Dispatch<TCommand>(TCommand command) where TCommand : ICommand
    {
        var handler = _autofacContainer.Resolve<ICommandHandler<TCommand>>();
        handler.Execute(command);
    }
}
```

Cum putem observa, această clasă are o singură metodă care obține o instanță a manipulatorului potrivit pentru comanda primită și apelează metoda *Execute* a acestuia.

Despre modul în care funcționează *Autofac* și rezolvarea automată a dependințelor voi vorbi mai în detaliu într-un subcapitol următor.

1.3.1.2 Implementarea interogărilor

Modul de implementare a interogărilor urmează o manieră similară cu cel al comenzilor, singura schimbare apare prin faptul că interogările trebuie să întoarcă un rezultat, de aceea mai apare o interfață marker.

```
public interface IQuery
{
}
```

Interfața *IQuery* este o interfață marker care specifică dezvoltatorului că o clasă care implementează această interfață poate fi folosită ca interogare.

```
public interface IQueryResult
{
}
```

Interfața *IQueryResult* este de asemenea o interfață marker utilizată pentru a specifica că o clasă va servi ca rezultat al unei interogări.

```
public interface IQueryHandler<in TQuery, out TQueryResult> where TQuery : IQuery
where TQueryResult : IQueryResult
{
    TQueryResult Retrive(TQuery query);
}
```

Interfața *IQueryHandler* este interfața care va fi implementată de clasele care vor avea rolul de manipulant al unei interogări. După cum putem observa, manipulantul este strâns legat de interogarea și de rezultatul interogării, respectiv de interfețele marker *IQuery* și *IQueryResult*, acest lucru fiind necesar pentru dispecerul de interogări pentru a putea obține o instanță a manipulantului.

```
public interface IQueryDispatcher
{
   TQueryResult Dispatch<TQuery, TQueryResult>(TQuery query) where TQuery : IQuery
   where TQueryResult : IQueryResult;
}
```

Putem observa că interfața *IQueryDispatcher* diferă de interfața *ICommandDispatcher* prin simplu fapt că metoda *Dispatch* returnează un obiect de tipul *IQueryResult* și nu *void*. Voi arăta și modul în care am implementat clasa *QueryDispatcher*.

```
public class QueryDispatcher : IQueryDispatcher
{
    private readonly IComponentContext _autofacContainer;

    public QueryDispatcher(IComponentContext context)
    {
        _autofacContainer = context;
    }

    public TQueryResult Dispatch<TQuery, TQueryResult>(TQuery query)
        where TQuery : IQuery where TQueryResult : IQueryResult
    {
        var handler =
           _autofacContainer.Resolve<IQueryHandler<TQuery, TQueryResult>>();
        var result = handler.Retrive(query);

        return result;
    }
}
```

Putem observa că implementarea este foarte asemănătoare cu ce de la clasa *CommandDispatcher* și diferența specificată mai sus, întoarecerea unui rezultat.

1.3.2 *Autofac* și rezolvarea automată a dependințelor

Autofac este o librărie care îmbunătățește modul prin care framework-ul ASP .NET Web API realizează rezolvarea automată a dependințelor. Această librărie are la bază ideea de inversare a controlului.

Inversarea controlului este o metodă de programare care spune că în loc să legăm clasele între ele și permiterea acestora să creeze noi instanțe de obiecte prin utilizarea operatorului *new*, modifici felul în care clasele sunt proiectate astfel încât dependințele sunt trimise ca parametru în momentul construcției clasei.

Dacă aplicația este construită având în vedere ideea de inversare a controlului atunci un framework, precum cel utilizat în acest proiect, va rezolva automat toate dependințele, astfel nemaifiind nevoie ca programatorul să utilizeze operatorul *new*.

După cum am văzut la stratul de service, am înlocuit rezolvitorul de dependințe existent în framework-ul ASP .NET Web API cu o instanță a bibliotecii *Autofac*.

Biblioteca *Autofac* are nevoie pentru rezolvarea dependințelor ca toate obiectele care vor fi rezolvate să fie înregistrate într-un container în care specificăm tipul obiectului, interfața pe care o implementează și durata de viață al obiectului. Biblioteca suportă și convenții de nume, scanare ansamblului de execuție, fiind motivul pentru care am ales să o folosesc în proiect. Mai jos o să ilustrez modalitatea de inregistrare al unui obiect.

```
builder.RegisterType<CommandDispatcher>()
                .As<ICommandDispatcher>()
                .InstancePerRequest();

builder.RegisterAssemblyTypes(Assembly.GetExecutingAssembly())
                .Where(t => t.Name.EndsWith("CommandHandler"))
                .AsImplementedInterfaces();
```

Clasa *BusinessAutofacModule* de care am vorbit în stratul de service conține doar astfel de înregistrări ale obiectelor în containerul de *Autofac*.

1.3.3 Validări

Pentru validarea datelor primite de la utilizator am folosit biblioteca *FluentValidation.* Am ales acestă bibliotecă pentru modul simple și elegant în care se face validarea datelor folosind interfața *fluent* și expresiile labda.

Astfel, pentru fiecare model validat, în cazul aplicației prezentate comenzi, a trebuit să construiesc o clasă care implementează clasa *AbstractValidator<T>* pentru fiecare comandă validată, iar în constructor să validez modelul.

```
RuleFor(u => u.Model.Password)
     .Must(p => p.Length >= 6)
     .WithMessage("Password must have at least 6 characters.");
```

Observăm în bucata de cod de mai sus maniera elegantă prin care se validează o componentă a modelului, în acest caz parola.

1.3.4 Procesare de imagini

Pentru a îmbunătăți rezultatele procesului de recunoaștere optică a caracterelor am implementat câteva funcționalități pentru a pregăti și optimiza imaginea. Pentru acest lucru am folosit librării standard .NET, algoritmi de procesare de imagini și o bibliotecă specializată pentru procesarea de imagini. Pașii pe care o imagine îi urmează sunt:

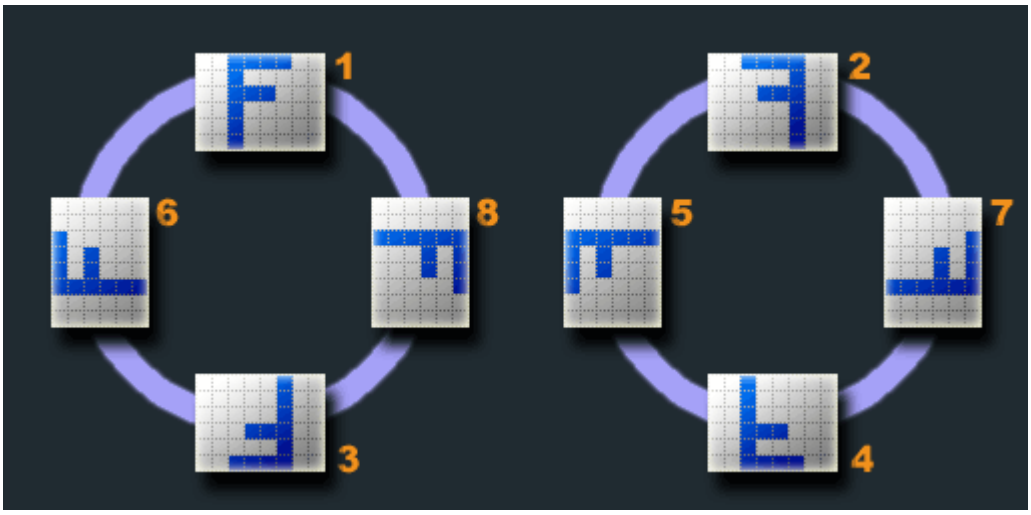
- Citirea datelor Exif[10] ale imaginii și rotirea acesteia în caz că este nevoie;
- Detectarea unghiului la care este înclinat textul;
- Eliminarea zgomotului;
- Binarizare;

În cele ce urmează vom discuta în detaliu despre modul în care au fost implementate cele enumerate de mai sus.

[10] Exchangeable image file format – http://www.cipa.jp/std/documents/e/DC-008-2012_E.pdf

1.3.4.1 Rotirea imaginii

Pentru a detecta orientarea imaginii, am decis să apelez la metadatele asociate acesteia. Există un standard denumit Exif care specifică metadatele pe care camerele de fotografiat, scannerele și alte dispozitive folosite la capturarea imaginii trebuie să adauge unei imagini. Printre aceste metadate există și un câmp care m-a interesat și anume câmpul *Orientation.* Acesta este un număr de la 1 la 8 corespunzător imaginii de mai jos.



Figură 2 Valori posibile pentru câmpul Orientation

Pentru a corecta orientarea fotografiei am utilizat librăria standard de imagini din framework-ul .NET și am verificat dacă imaginea conține câmpul dorit și în caz că acest câmp există imaginea să fie întoarsă. De aceea am creat o metodă de extindere (en. extension method) a clasei *Bitmap* care să verifice existența câmpului și rotirea și eventual întoarcerea imaginii în mod corespunzător.

Metodele de extindere permit adăugarea de metode la tipuri deja existente fără a crea un nou tip derivat, recompilarea sau aducerea de alte modificari tipului original. Metodele de extindere sunt un tip special de metode statice, dar sunt invocate ca metode ale tipului invocat. Aceste metode au o signatură specifică, astfel primul parametru este de tipul extins prefixat de operatorul *this.*

Mai jos avem implementarea metodei care are scopul de a verifica existența câmpului *Orientatin* și modificarea imaginii. Acest câmp are ID-ul 0x0112. Astfel în metodă se verifică dacă există acest ID, iar dacă acesta există se obține tipul de transformare necesară.

```
public static class ImageHelper
{
    public static void RotateByExifOrientation(this Bitmap image,
                                                bool updateExifData = true)
    {
        const int orientationId = 0x0112;

        if (!image.PropertyIdList.Contains(orientationId))
        {
            return;
        }

        var pItem = image.GetPropertyItem(orientationId);
        var fType = GetRotateFlipTypeByExifOrientationData(pItem.Value[0]);

        if (fType == RotateFlipType.RotateNoneFlipNone)
        {
           return;
        }

        image.RotateFlip(fType);
        if (updateExifData)
        {
           image.RemovePropertyItem(orientationId);
        }
     }

     private static RotateFlipType
                 GetRotateFlipTypeByExifOrientationData(int orientation)
     {
         switch (orientation)
         {
            case 1:
            default:
                return RotateFlipType.RotateNoneFlipNone;
            case 2:
                return RotateFlipType.RotateNoneFlipX;
            case 3:
                return RotateFlipType.Rotate180FlipNone;
            case 4:
                return RotateFlipType.Rotate180FlipX;
            case 5:
                return RotateFlipType.Rotate90FlipX;
            case 6:
                return RotateFlipType.Rotate90FlipNone;
            case 7:
                return RotateFlipType.Rotate270FlipX;
            case 8:
                return RotateFlipType.Rotate270FlipNone;
         }
     }
}
```

RotateFlipType este o enumerare existentă în framework-ul .NET, care specifică modul în care imaginea trebuie rotită și intoarsă, enumerare folosită de metoda *RotateFlip* din clasa *Bitmap.*

1.3.4.2 Detectarea înclinării textului

Pentru a detecta unghiul la care este înclinat dintr-o fotografie am ales să folosesc transformarea Hough[11]. Algoritmul de corecție a înclinării are la bază următoarele ideii:

- Găsirea de linii importante în imagine;
- Calcularea unghiului acelei linii;
- Calculul unghiului de înclinare ca fiind media unghiurilor;
- Rotirea imaginii;

Liniile sunt detectate utilizând algoritmul lui Hough. Fiecare pixel din imagine poate sta pe o infinitate de drepte. Pentru a găsi doar dreptele importante lăsăm fiecare pixel să voteze dreptele care trec prin el. Liniile cu cele mai multe voturi sunt selectate pentru determinarea unghiului de înclinare. Acest algoritm este destul de costisitor din punct de vedere computațional. Pentru scopul de a găsi unghiul la care e înclinat textul am avut nevoie doar de linia care trece în josul textului, ca și cum am sublinia textul. De aceea am restricționat numărul de pixeli care pot vota o dreaptă prin condiția ca pixelul curent să fie negru, iar pixelul imediat de sub el să fie alb. Ca o adăugare, acest algoritm funcționează mult mai bine pe imaginile binarizate[12]. (How to deskew an image, 2006) Dar despre binarizare vom vorbi mai târziu.

Algortimul calculează unghiul la care este inclinată imaginea pornind de la ecuația dreptei $y = m \cdot x + t$. Pornind de la această ecuație obținem că unghiul dreptei este obținut din ecuația $d = y \cdot \cos(\alpha) - x \cdot \sin(\alpha)$. Cum nu putem căuta într-un interval infinit, căutăm liniile care au $-20 \leq \alpha \leq 20$, folosind un pas de 0,2 și rotunjim unghiul d.

Astfel, algoritmul în pseudocod arată astfel:

1. Creăm o matrice Hough inițializată cu 0;
2. Parcurgem fiecare pixel din imagine;

[11] Transformarea Hough – http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm
[12] Imagine binară - https://en.wikipedia.org/wiki/Binary_image

3. Dacă satisface condiția ca pixelul curent să fie negru, iar pixelul imediat de sub el să fie alb atunci calculăm unghiul *d* și incrementăm perechea Hough[α*5][d] pentru $-20 \leq \alpha \leq 20$;
4. Sortăm primele 20 de perechi din matricea Hough care au cel mai mare valoare;
5. Calculăm unghiul de înclianare β ca media lui α din aceste perechi;
6. Intoarcem imaginea cu $-1 \cdot \beta$ grade;

Pentru a obține o eficiență mai mare am precalculat valorile pentru funcțiile *sin* și *cos*.

1.3.4.3 Emgu CV

Emgu CV este un bibliotecă .NET care funcționează ca un decorator peste biblioteca *Open CV*[13]. Astfel, *Emgu CV* permite apelarea funțiilor din biblioteca *Open CV* din limbajele compatibile .NET, printre care se află evident și C#.

Open CV este o bibliotecă open-source utilizată în domeniile de computer vision și învățarea automată. Această bibliotecă a fost construită pentru a oferi o infrastructură comună pentru aplicațiile de computer vision și pentru accelerarea utilizării percepției mașinilor în produsele comerciale. Acestă bibliotecă este foarte populară, lucru care rezultă din faptul că un număr mare de companii, printre care Microsoft, Google, Intel, IBM, Sony, utilizează această bibliotecă în produsele lor. Biblioteca a fost proiectată cu scopul de a fi eficientă din punct de vedere computațional și cu focalizare pe aplicațiile în timp real. Această bibliotecă este scrisă în limbajele de programare C și C++ și profită de procesarea multi nucleu și de accelerația hardware. Din păcate oferă suport doar pentru limbajele de programare C++, C, Python, Java, MATLAB. De aceea am folosit decoratorul *Emgu CV* pentru a realiza operațiile necesare de procesare de imagini din aplicație.

1.3.4.4 Eliminarea zgomotului

Zgomotul într-o imagine este reprezentat de existența în imagine a unor variații mici de luminozitate care apar în mod aleatoriu și care se aseamană cu mici puncte în cadrul imaginii. Zgomotul este un produs nedorit adăugat de aparatele care se ocupă cu captura de imagini.

Eliminarea zgomotului este o procedură costisitoare din punct de vedere computațional. De aceea am ales ca în loc să încerc extragerea zgomotului din imagine să încerc să

[13] Open Source Computer Vision Library - http://opencv.org/about.html

„netezesc” acest zgomot astfel încât în urma binarizării imaginea să nu mai prezinte zgomot. Pentru a realiza acest lucru am apelat la biblioteca *Emgu CV* și am folosit metoda de estompare gaussiană. Această metodă prezinta dezavantajul de a estompa și detaliile pozei, dar făcută în mod responsabil păstrează suficiente detalii, înlăturând și zgomotul.

1.3.4.5 Binarizarea imaginii

Binarizarea reprezintă procesul de convertire a unei imagini, astfel încât rezultatul acestei convertiri să fie o imagine în care pixelii acesteia să aibă doar două valori și anume alb sau negru.

Pentru rezultate cât mai optime am decis să folosesc biblioteca *Emgu CV* și să aleg metoda de binarizare Otsu cu prag de decizie adaptiv. Metoda propusă de Otsu presupune că o imagine conține două clase de pixeli (pixeli de fundal și pixeli de prim-plan) și calculează pragul care separă aceste două clase realizând astfel binarizarea imaginii. Acestă metodă împreună cu estomparea gaussiană oferă rezultate foarte bune în testele pe care le-am făcut.

1.3.4.6 Recunoașterea caracterelor

Pentru procesul de recunoaștere optică de caractere am apelat la biblioteca *Tesseract* din cadrul bibliotecii *Emgu CV*. *Tesseract* este o bibliotecă open-source care a are ca scop recunoașterea optică de caractere. A fost inițial dezvoltată de firma HP și a depenit open-source din anul 2005. Din anul 2006 compania Google se implică în dezvoltarea acestei librării.

Pentru a rula acest motor de recunoaștere optică a caracterelor avem nevoie, pe lângă biblioteca *Tesseract*, și de un fișier care conține datele de antrenament pentru limba respectivă. Momentan aplicația de server concepută de mine oferă suport doar pentru limba engleză, dar rezultatele întoarse de bibliotecă sunt satisfăcătoare și pentru alte limbi.

1.3.4.7 Corectarea textului

Pentru corectarea textului rezultat am utilizat API-ul *Microsoft Bing Spell Check* din suita de servicii online *Cognitive Services.* API-ul detectează erorile din textul dat returnează

sugestiile într-un obiect de tip JSON care este deserializat de aplicația de server și apoi urmează modificarea erorilor cu sugestiile primite de la API-ul *Microsoft Bing Spell Check.*

1.4 Nivelul de date

Nivelul de date este este o componentă foarte importantă în cadrul aplicației întrucât acesta are ca responsabilitate conectarea și managementul bazei de date. Aici se găsește codul utilizat pentru accesul la baza de date și efectuare de diverse operații pe această bază de date. În aplicația prezentată, conectarea, crearea și manipularea bazei de date este făcută prin intermediul framework-ului *EntityFramework Code-First*. De asemeni, pentru a ca aplicația stratul de business să lucreze cu abstracțiuni am implementat șablonul de proiectare *Repository*. În cele ce urmează voi detalia aceste concepte.

1.4.1 EntityFramework

EntityFramework este un framework ORM[14] care permite dezvoltatorilor să lucreze cu datele din baza de date ca obiecte specifice domeniului, eliminând nevoia de a mai scrie majoritatea codului pe care dezvoltatorii erau nevoiți să-l scrie pentru conectarea la baza de date și executarea interogărilor și sarcinilor specifice bazelor de date. Abordarea *Code-First* permite dezvoltatorilor să se concetreze pe conceperea claselor domeniului, clase care răspund cerințelor din aplicație decât să creiezi baza de date și apoi clasele specifice domeniului. Astfel API-ul *Code-First* va crea baza de date bazată pe clasele și configurările date.

În continuare voi arăta cum arată o entitate, adică o clasă din domeniu după care *EntityFramework* va genera un tabel în baza de date, o configurare și apoi vom vorbi despre clasa context și felul în care arată o migrare.

1.4.1.1 Entitățile

Entitățile sunt clase din domeniul aplicației utilizate de *EntityFramework* pentru a crea un tabel în baza de date. Voi prezenta două clase, *User* și *Role*, pentru a înțelege în subcapitolul următor configurările.

[14] Object/Relational Mapping

```
public class User : BaseEntity
{
    public string Email { get; set; }

    public string Password { get; set; }

    public string Name { get; set; }

    public Guid RoleId { get; set; }

    public Role Role { get; set; }

    public IList<Picture> Pictures { get; set; }
}
```

Clasa *BaseEentity* este o clasă care conține o singură proprietate, Id de tipul Guid, din care toate entitățile din aplicație vor fi derivate.

```
public class Role : BaseEntity
{
    public string Name { get; set; }

    public IList<User> Users { get; set; }
}
```

După cum putem observa, clasele sunt extrem de simple, având componente doar niște proprietăți simple. Clasa *Role* conține o listă de obiecte de tip *User* pentru ai specifica *EntityFramework* că există o relație unu la mai mulți între rol și utilizator. Din aceste clase *EntityFramework* va genera tabele în baza de date.

1.4.1.2 Configurări

După cum am vazut mai sus, clasele noastre nu conțin foarte multe detalii despre ce proprietați adiționale am dori să îndeplinească datele din acestea. Aici intervin configurările. Configurările sunt clase care moștenesc din clasa *EntityTypeConfiguration<T>* și specifică detalii adiționale despre entități.

După cum putem observa mai jos, am specificat pentru entitatea *User* că proprietățile *Email* și *Password* să fie obligatorii în baza de date, lungimea câmpului *Name* să fie de 30 de

caractere și am specificat cheia străină pentru tabela *Roles* care va fi generată pe baza entității *Role.*

```
public class UserMapping : EntityTypeConfiguration<User>
{
    public UserMapping()
    {
        Property(p => p.Email)
            .IsRequired();

        Property(p => p.Password)
            .IsRequired();

        Property(p => p.Name)
            .IsRequired()
            .HasMaxLength(30);

        HasRequired(s => s.Role)
             .WithMany(s => s.Users)
             .HasForeignKey(s => s.RoleId);
     }
}
```

După cum se poate observa configurările se fac la fel ca la biblioteca *FluentValidation* folosind interfața fluent și expresiile lambda.

1.4.1.3 Clasa context

Clasa context, in aplicația are numele de *OcrApiContext*, moștenește din clasa *DbContext* și este o clasă foarte importantă pentru *EntityFramework.* Această clasă este puntea de legătură dintre domeniul aplicației și baza de date. Aceasta este cea mai importantă clasă care este responsabilă cu interacțiunea cu datele ca și obiecte.

Această clasă va conține mulțimea tuturor entităților și de asemenea aici vor fi setate configurările pentru crearea tabelelor.

1.4.1.4 Migrări

Utilizând obiectele descrise mai sus *EntityFramework* va genera o migrare care va fi folosită la crearea, respectiv actualizarea bazei de date. Migrările sunt clase generate automat

care moștenesc din clasa *DbMigration* și care au două metode *Up,* respectiv *Down,* iar aici sunt descrise modificările bazei de date.

1.4.2 Șablonul de proiectare *Repository*

Șablonul de proiectare *Repository* este potrivit lui Martin Fowler un mediator între obiectele din domeniu și nivelurile de mapări de date care folosește o interfață asemenea unei colecții pentru a accesa obiecte din domeniu. Astfel un client al unui *Repository* va efectua operații cu obiecte ca și când ar avea o colecție de astfel de obiecte, inclusiv filtrare, adăugare, ștergere, iar *Repository* va efectua operațiile fără să expună detalii de implementare.

Pentru a implementa acest șablon am definit o interfață generică care acceptă obiecte derivate din *BaseEntity,* astfel *Repository* să poată fi utilizat pe entitățile din aplicație, care definește operațiile CRUD (create, rename, update, delete) plus niște operații de filtrare.

```
public interface IRepository<T> where T : BaseEntity
{
    void Add(T entity);

    void Delete(T entity);

    T Get(Guid id);

    IQueryable<T> GetAllQuery();

    IEnumerable<T> Find(Expression<Func<T, bool>> predicate);

    void Update(T entity);

    void Save();
}
```

Clasa care implementează această interfață va avea un membru detipul *OcrApiContext* pe care vom executa aceste operații.

Șablonul de proiectare este destul de simplu dar are o deosebită importanță în practică fiind intens discutat prin articolele de specialtate.

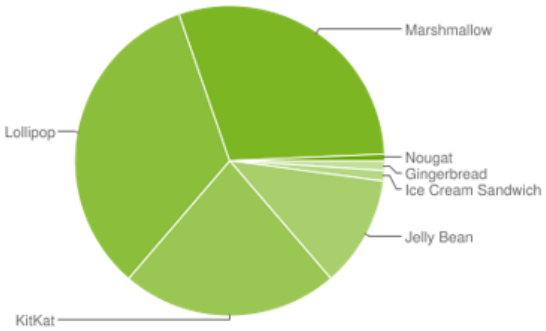
Capitolul 2: Dezvoltarea clientului Android

Aplicația de Android este construită cu scopul de a facilita comunicarea cu API-ul a utilizatorului prin oferirea unei interfețe grafice intuitive și foarte rapidă și facilități de efectuare de fotografii și integrarea mai multor tehnologii astfel încât aplicația să aibe un consum mic de resurse. Tot odată aplicația este construită pentru a demonstra flexibilitatea API-ului prin comunicarea dintre două platforme aparent incompatibile. Aplicația a fost scrisă folosind limbajul de programare Java folosind mediul de dezvoltare Android Studio.

Pentru a crea aplicații pentru sistemul de operare trebuie să apelăm la API-ul Android. Android API este scris în limbajul Java și este oferit de compania Google în regim deschis și gratuit . Există mai multe versiuni ale acestui API, fiecare corespunzătoare unei versiuni a sistemului de operare Android. La momentul scrierii acestei lucrări, 11.01.2017, API-ul Android a ajuns la versiunea 25, corespunzător versiunii de Android 7.1 Nougat.

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	1.0%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	1.1%
4.1.x	Jelly Bean	16	4.0%
4.2.x		17	5.9%
4.3		18	1.7%
4.4	KitKat	19	22.6%
5.0	Lollipop	21	10.1%
5.1		22	23.3%
6.0	Marshmallow	23	29.6%
7.0	Nougat	24	0.5%
7.1		25	0.2%

Data collected during a 7-day period ending on January 9, 2017.



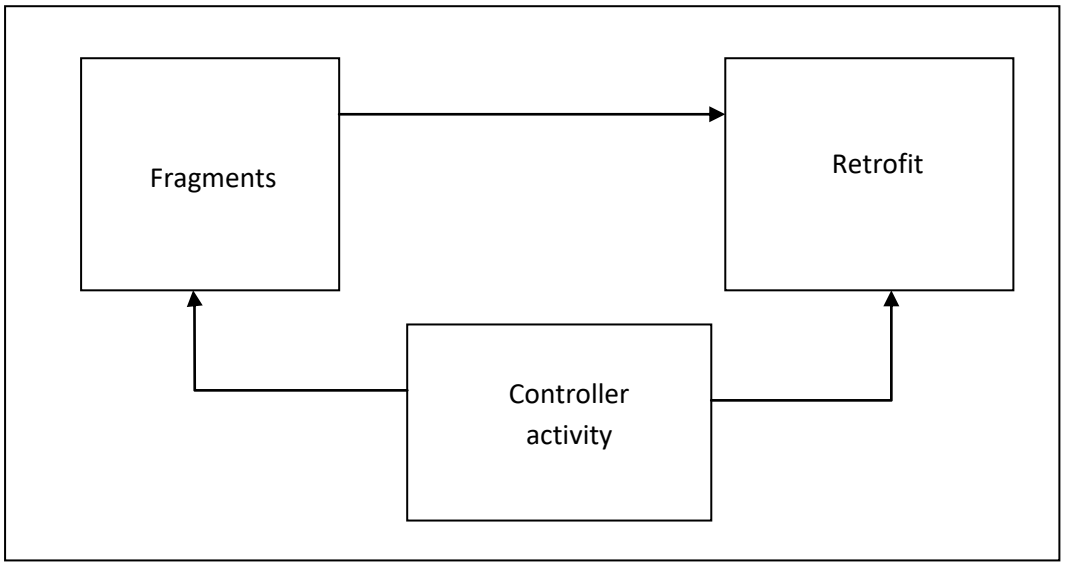
Figură 3 Distribuția versiunilor de Android la data de 11.01.2017[15]

[15] https://developer.android.com/about/dashboards/index.html

Pentru a acoperi un număr cât mai mare de dispozitive, dar să am acces și la cele mai noi versiuni ale funcționalităților pe care le-am implementat în aplicație am decis să aleg nivelul minim al API-ului Android pe care aplicația să îl suporte să fie nivelul 21 corespunzător versiunii de Android 5.0 Lollipop.

2.1 Arhitectura generală a aplicației

Arhitectura aplicației este una destul de simplă, arhitectură ce se aseamănă destul de mult cu arhitectura MVC (model-view-controller).



Figură 4 Arhitectura clientului Android

În figura 4 putem observa că aplicația are trei componente principale care au în mare parte responzabilitățile părților componente din șablonul arhitectural MVC. Astfel, biblioteca *Retrofit* este responsabilă de furnizarea modelelor, activitatea Android joacă rolul unui controller-ului, iar fragmentele îndeplinesc sarcina de a afișa datele la utilizator. În cazul nostru, activitatea nu are un rol mare în procesarea datelor, acestea fiind procesate și obținute de fiecare fragment în parte, aceasta fiind și diferența dintre arhitectura aplicației și MVC. Astfel fragmentele obțin datele necesare pentru operațiile pe care le execută, iar activitatea este responsabilă de modul în care fragmentele sunt afișate. În următoarele capitole voi descrie modul în care biblioteca *Retrofit* funcționează, despre fragmente și modul de comunicare al acestora cu activitatea. Voi mai descrie și câteva fragmente cu funcționalități mai deosebite, cu precădere cel care se ocupă cu realizarea fotografiilor.

2.2 Retrofit

Retrofit este un client REST pentru Java și Android construit de către firma *Square* care furnizează un framework puternic pentru interacțiunea cu API-uri și trimiterea de cereri în rețea utilizând biblioteca *OkHttp*.

OkHttp este de asemenea construit de către firma *Square* și este un client HTTP pentru Java și Android care îmbunătățește modul prin care aplicațiile comunică prin rețea.

Librăria face descărcarea datelor în format JSON sau XML[16] de la un API foarte simplă, tot odată convertind datele în obiecte ale domeniului, în cazul nostru în obiecte Java. *Retrofit* realizează conversia obiectelor JSON în obiecte Java implicit utilizând biblioteca *GSON*.

GSON este o bibliotecă Java realizată de către cei de la Google care are ca scop conversia obiectelor Java în obiecte JSON și invers. Biblioteca funcționează și pe obiecte Java deja existente la care nu avem acces la codul sursă.

Pentru ca biblioteca să funcționeze aplicația trebuie să obțină permisiunea de a accesa rețeaua, respectiv internetul. În cadrul aplicațiilor de Android, acest lucru se face adăugând în fișierul *AndroidManifest.xml* următoarea permisiune:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

Următorul pas este să obținem bibliotecile *GSON* și *Retrofit*. Pentru a realiza acest lucru trebuie să includem în fișierul *app/build.grandle* următoarele dependințe:

```
dependencies {
    compile 'com.google.code.gson:gson:2.4'
    compile 'com.squareup.retrofit2:retrofit:2.1.0'
    compile 'com.squareup.retrofit2:converter-gson:2.1.0'
}
```

În acest moment biblioteca este gata de utilizare. Pentru că *Retrofit* este o bibliotecă ce pune accent pe siguranța tipurilor trebuie să creăm clasele Java care vor fi corespunzătoare cu obiecte JSON primite sau expediate. Clasele sunt extrem de simple, conținând doar proprietăți și metodele de modificare și expunere ale acestora.

[16] Extensible Markup Language - https://www.w3.org/XML/

Pentru a putea utiliza această bibliotecă în aplicație trebuie să construim o interfață în care să specificăm metodele API-ului utilizând adnotări Java. În această interfață se specifică tipul cererii HTTP (GET, POST, PUT, DELETE), adresa la care se află metoda și tipul conținutului unde este cazul și alte configurări.

```
public interface OcrApiInterface {

    @POST("api/user/add")
     Call<ResponseBody> addUser(@Body AddUserModel model);

    @GET("api/user/get/{id}")
    Call<UserModel> getUser(@Path("id") String id);

    @DELETE("api/user/delete")
    Call<ResponseBody> deleteUser(@Query("id") String id);

    @PUT("api/user/change")
    Call<ResponseBody> changeUserRole(@Body ChangeRoleModel model);

    @GET("api/user/all")
    Call<List<UserModel>> getAll();

    @Multipart
    @POST("api/content/add")
    Call<ResponseBody> addImage(@Part MultipartBody.Part file);
}
```

În secvența de cod de mai sus putem observa o parte din metodele API-ului despre care am vorbit în capitolul 1 mapate la o interfață Java. Se poate observa că se lucrează doar cu obiecte concrete, iar apelul la API se face prin apelarea metodelor din această interfață.

În continuare trebuie să obținem o instanță a unui obiect care să implementeze interfața definită mai sus. Obiectul este creat de *Retrofit*, dar trebuie să mai configurăm anumite setări cum ar fi adresa API-ului, convertorul pentru JSON și pentru că API-ul descris în această lucrare a fost securizat, am adăugat un intereceptor pentru cererile HTTP care să adauge în antetul acestora jetonul de autentificare. Voi descrie în continuare modul prin care am făcut aceste lucruri.

În primul rând voi vorbi despre modul în care am făcut interceptarea cererilor HTTP. Acest lucru a fost realizat prin intermediul bibliotecii *OkHttp* pe care se bazează biblioteca *Retrofit*. Interceptorul oprește cererile HTTP adaugă în antetul acestora câmpul *Authorization* în care adaugă jetonul obținut în urma unei cereri la intrarea în aplicație la API cu numele și parola utilizatorului. Mai jos puteți observa maniera de implementare a interceptorului.

```
public class AuthorizationInterceptor implements Interceptor {
    private String authorizationToken;

    public AuthorizationInterceptor(String token){
        authorizationToken = token != null
                ? "Bearer " + token
                : "";
    }

    @Override
    public Response intercept(Chain chain) throws IOException {
        Request request = chain.request();
        request = request.newBuilder()
                .addHeader("Authorization", authorizationToken)
                .method(request.method(), request.body())
                .build();
        Response response = chain.proceed(request);
        return response;
    }
}
```

Se poate observa că interceptorul primește ca parametru la construire un șir de caractere care reprezintă jetonul de autentificare pe care îl prefixează cu șirul de caractere „Bearer” și în metoda suprascrisă *intercept* adaugă jetonul în antetul cererii.

În continuare vom discuta despre modul în care se obține o instanță a unui obiect ce implementează interfața definită mai sus și setarea diferitelor obțiuni și a interceptorului.

```
public class ApiServiceBuilder {
    private static String URL = "http://192.168.0.102/OcrApi/";
    private OcrApiInterface OcrApi;

    public ApiServiceBuilder(AuthorizationInterceptor interceptor){
        OkHttpClient client = new
OkHttpClient.Builder().addInterceptor(interceptor).build();
        Retrofit retrofit = new Retrofit.Builder()
                .baseUrl(URL)
                .addConverterFactory(GsonConverterFactory.create())
                .client(client)
                .build();
        OcrApi = retrofit.create(OcrApiInterface.class);
    }

    public static void setUrl(String url){
        URL = "http://"+ url + "/OcrApi/";
    }

    public OcrApiInterface GetService(){
        return OcrApi;
    }
}
```

Clasa definită mai sus are responsabilitatea de a obține o instanța a serviciului nostru care este configurată complet. Astfel, putem obsera setarea adresei URL a API-ului, setarea convertorului JSON, și anume GSON, modificarea clientului HTTP implicit cu unul modificat care să conțină interceptorul de autentificare. Clasa expune o singură metodă care returnează o instanță a serviciului nostru care va fi utilizată în aplicație. Pentru apelul la API se va crea un obiect de tipul definit mai sus și se va apela metoda *GetService()* apoi se vor apela metode interfeței din obiectul obținut.

2.3 Fragmente

Tradițional, fiecare ecran dintr-o aplicație Android era implementat într-o activitate separată. Dar activitățile trebuie să se ocupe de mai multe lucruri în cadrul unei aplicații cum ar fi ciclul de viață, interacțiunea cu sistemul de operare și altele. Având toate aceste sarcini, activitățile tind să devină mai greoaie. Aici intervin fragmentele. Introduse o dată cu lansarea versiunii de Android 3.0, fragmentele se adresează problemei enunțate mai sus prin faptul că iau povara interfeței cu utilizatorul ale activităților, lăsându-le acestora restul responsabilităților. Astfel, putem avea o singură activitate care se ocupă cu interacțiunea cu sistemul și managementul fragmentelor, iar fragmentele să-și facă treaba într-un mod mai rapid decât cel tradițional.

Similar cu activitățile, fragmentele au și ele un ciclu de viață propriu în cadrul aplicației. Voi detalia câteva dintre acestea care au fost folosite cel mai des în cadrul aplicației.

Cel mai des folosit eveniment în aplicație din cadrul unui fregment a fost evenimentul *onCreateView*. Acesta este evenimentul pe care sistemul Android îl invocă atunci când este timpul ca fragmentul să-și randeze interfața grafică. Aici programatorul trebuie să returneze un obiect de tip *View*. Imediat după ce evenimetul *onCreateView* s-a terminat se apelează metoda *onViewCreated*. Aici programatorul populează cu date interfața grafică și obține referințe la diferite elemente grafice.

De cele mai multe ori fragmentele trebuie să comunice între ele, dar ca acestea să fie reutilizabile acestea nu trebuie să aibe referințe directe între ele. Fragmentele comunică între ele prin intermediul activității la care sunt asociate, iar două activități nu trebuie să comunice direct niciodată. Pentru a realiza această comunicare API-ul Android definește o modalitate prin care un fragment poate comunica cu activitatea cu care este asociat. Conform modalității propuse în API-ul Android, se definește o interfață pe care activitatea trebuie să o

implementeze. Apoi fragmentul, care are un membru de tipul interfeței implementate de activitate, la declanșarea evenimentului *onAttach,* eveniment declanșat atunci când un fragment este atașat unui context, va încerca să convertească contextul la tipul interfeței. Mai jos putem vedea un exemplu de un astfel de procedeu.

```
@Override
    public void onAttach(Context context){
        super.onAttach(context);

        try{
            activity = (IUserListFragmentListener) context;
        }catch (ClassCastException e){
            throw new ClassCastException(context.toString()
                    + " must implement IExpiredLogin");
        }
    }
```

Astfel, când fragmentul dorește să comunice cu alt fragment, de exemplu când am selectat un obiect dintr-o listă și dorim să vedem mai multe detalii, va apela metoda din interfață, iar apoi activitatea va transmite datele primite fragmentului dorit. Un exemplu de o astfel de interfață este prezentat mai jos.

```
public interface IUserListFragmentListener extends IExpiredLogin {
    void onUserSelected(String id);
}
```

Modul prin care activitatea comunică cu un fragment este prin intermediul unui obiect de tip *Bundle.* Mai jos putem vedea implementarea metodei din interfeă și modul prin care activitatea transimte mesajul unui fragment.

```
@Override
    public void onUserSelected(String id) {
        FragmentManager fragmentManager = getFragmentManager();
        FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();
        EditUserFragment editUserFragment = new EditUserFragment();
        Bundle args = new Bundle();
        args.putString(USER_TO_UPDATE_ID_KEY, id);
        args.putString(USER_MANAGER_KEY, userDetailsManagerJson);
        editUserFragment.setArguments(args);
        fragmentTransaction.replace(R.id.fragment_container, editUserFragment);
        fragmentTransaction.addToBackStack(USER_LIST_FRAGMENT);
        fragmentTransaction.commit();
    }
```

2.4 Activitatea controller

Clasa *Activity* este o componentă importantă din aplicațiile de Android și modul prin care obiectele de acest tip sunt puse la o laltă și lansate în execuție reprezintă o componentă importantă în modul prin care aplicațiile destinate sistemului de operare Android funcționează.

O activitate, în sens tradițional, reprezintă un singur lucru pe care un utilizator poate să-l facă. Aproape toate activitățile interacționează cu utilizatorul, astfel acestea sunt responsabile pe lângă alte lucruri de crearea ferestrei în care este plasată interfața cu utilizatorul.

Pentru a putea fi utilizate într-o aplicație, toate activitățile trebuie să apară în fișierul *AndroidManifest.xml.*

În aplicația de Android prezentată în această lucrare, activitatea nu se ocupă cu nimic legat de partea de interfață cu utilizatorul, cum a fost menționat și în sub capitolul precedent pentru a câștiga un spor de performanță. Aceasta are ca atribuții doar managementul fragmentelor și asigurarea condițiilor necesare rulării acestora, cum ar fi permisunile. Dar înainte să detaliez partea de permisiuni, vom vorbi despre un fișier important din arhitectura unei aplicații Android, care a mai fost menționat înainte, dar despre care nu am vorbit și este vorba de *AndroidManifest.xml*.

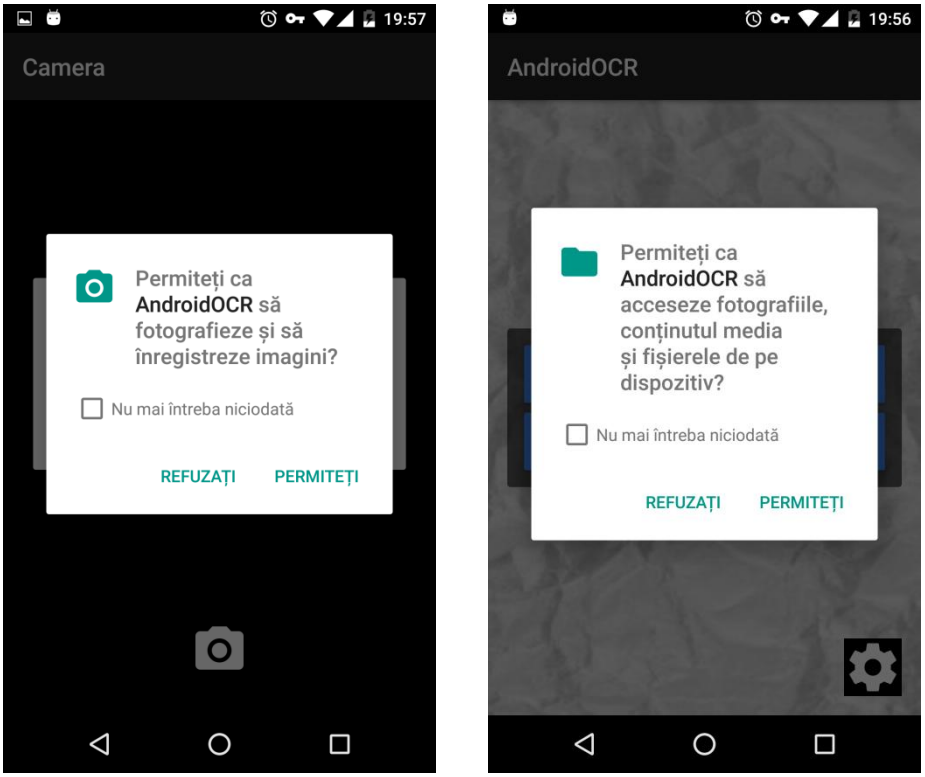
AndroidManifest.xml este un fișier foarte important în cadrul aplicațiilor Android care are ca scop principal oferirea informațiilor esențiale despre aplicație sistemului de operare Android, informații fără de care sistemul nu ar putea rula aplicația. Printre cele mai importante lucruri pe care fișierul specificat mai sus le conține menționez: specifică numele pachetul Java pentru aplicație, care trebuie să fie unic, lista componentelor aplicației, cum ar fi activitățile, nivelul minim al API-ului Android pentru care aplicația este compatibilă, permisiunile aplicației și altele. Orice aplicație Android trebuie să conțină un astfel de fișier în rădăcina proiectului, astfel aceasta nu va rula.

În cadrul sistemului de operare Android, există anumite restricții care limitează accesul la anumite resurse sau secvențe de cod cu scopul de a proteja secvențe de cod critice sau care ar putea afecta experiența utilizatorului. Aceste restricții poartă numele de *permisiuni.* Orice permisiune este identificată printr-un nume unic. Pentru ca o aplicație să acceseze o secțiune mai importantă, aceasta trebuie să ceară o permisiune. Pentru a cere o permisiune, aceasta trebuie să fie trecută în fișierul *AndroidManifest.xml*.

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

În secvența de cod de mai sus putem observa permisiunile pe care clientul de Android descris în această lucrare le cere. Permisiunile sunt foarte sugestive, astfel orcine poate să-și dea seama că aplicația dorește permisiuni asupra camerei de fotografiat, accesarea memorie interne și acces la internet.

O particularitate a versiunii de Android 6.0 este că aplicațiile nu mai cer permisiuni atunci când sunt instalate ci în momentul în care aplicația este instalată și aceasta intră într-o zonă de cod restricționat.



Figură 5 Permisiunile la rulare cerute de clientul de Android

După cum putem observa în figura 5 aplicația afișează niște mesaje sugestive care explică utilizatorului ce intenționează aplicația să facă, iar acesta poate permite accesul sau nu. Astfel aplicația trebuie să se adapteze în funcție de preferințele utilizatorului. De aceste lucruri se ocupă activitatea cu rol de controller din aplicație.

2.5 Interfața cu utilizatorul

Pentru interfața cu utilizatorul am ales un design simplu care să fie ușor de înțeles pentru utilizator. Pentru a realia acest lucru, interfața grafică utilizeaz texte sugestive, pictograme sugestive pentru butoane și o așezare foarte intuitivă.

Interfața grafică în cadrul aplicațiilor Android este definită prin utilizarea unor fișiere stocate în format *.xml*, iar fiecare fragment și anumite elemente grafice au definite un asemenea fișier. Mai jos putem observa cum arată un fișier în care este definită interfața grafică a unui fragment.

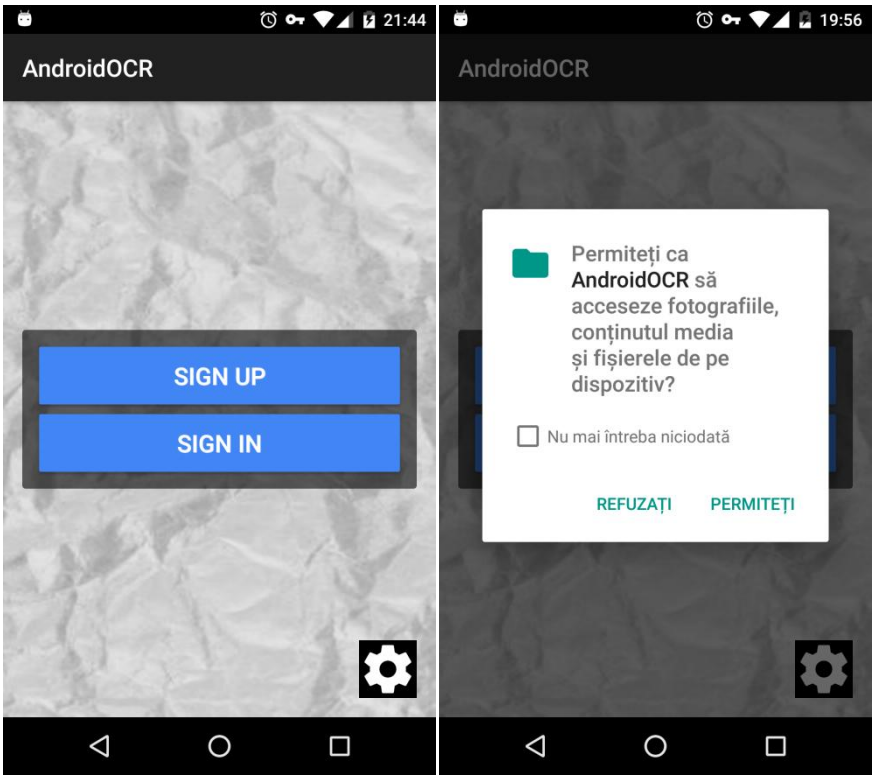
```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.petru.cameraocr.CameraActivity"
    android:background="#000000">

    <TextureView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/textureView"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true" />

    <ImageButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/cameraButton"
        android:src="@drawable/ic_photo_camera_white_48dp"
        android:contentDescription="@string/camera_btn"
        android:backgroundTint="#00FFFFFF"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        android:layout_marginBottom="20dp" />
</RelativeLayout>
```

3 Manual de utilizare

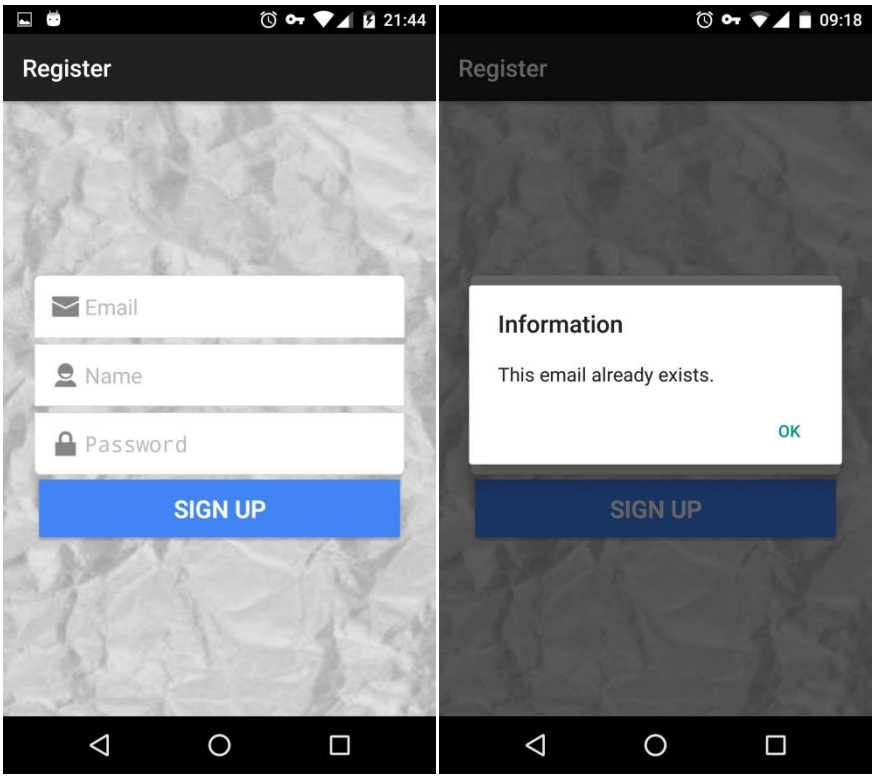
În momentul în care se lansează aplicația, utilizatorului îi este prezentat un ecran în care îi sunt prezentate două opțiuni: aceea de a-și crea un cont nou de utilizator și opțiunea de a se înregistra cu un cont deja existent. Tot aici, utilizatorului îi este cerută permisiunea ca aplicația să acceseze spațiul de stocare al telefonului, dacă aplicația este la prima rulare sau permisiunea a fost retrasă din setări.



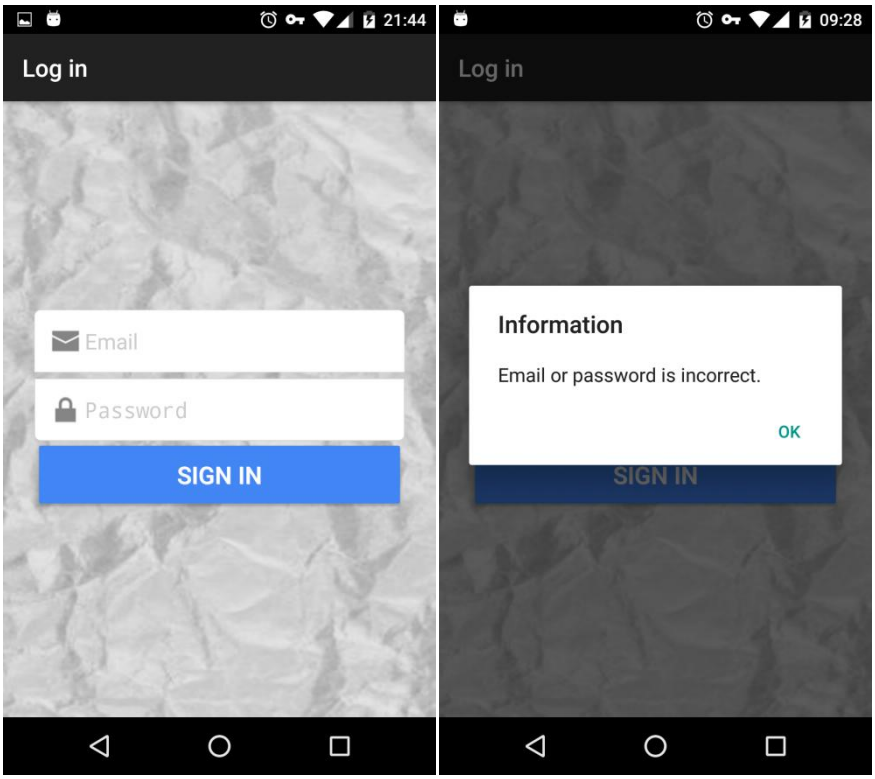
Dacă utilzatorul alege opțiunea de a se înregistra (SIGN UP) atunci îi este prezentat un nou ecran in care acesta trebuie să introducă detaliile de conectare cerute. Aceste detalii sunt:

- Adresa de email;
- Numele utilizatorului;
- Parola asociată contului;

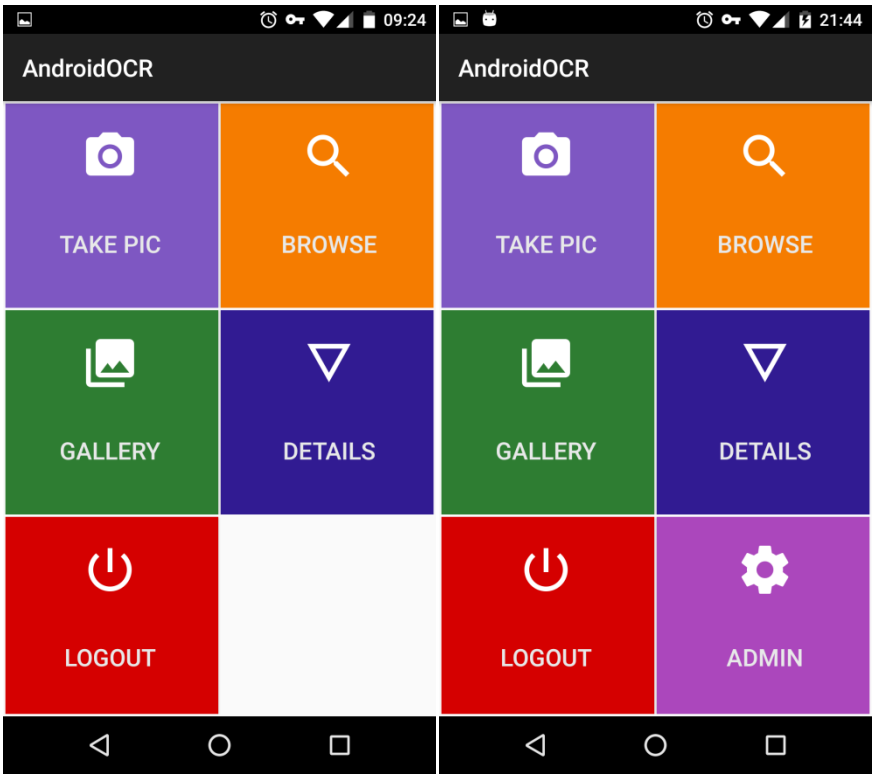
Dacă datele de conectare sunt valide, atunci utilizatorul va fi redirectat la fereastra de logare în aplicație, altfel îi va fi prezentat un mesaj în care i se specifică ce anume a greșit în completarea datelor.



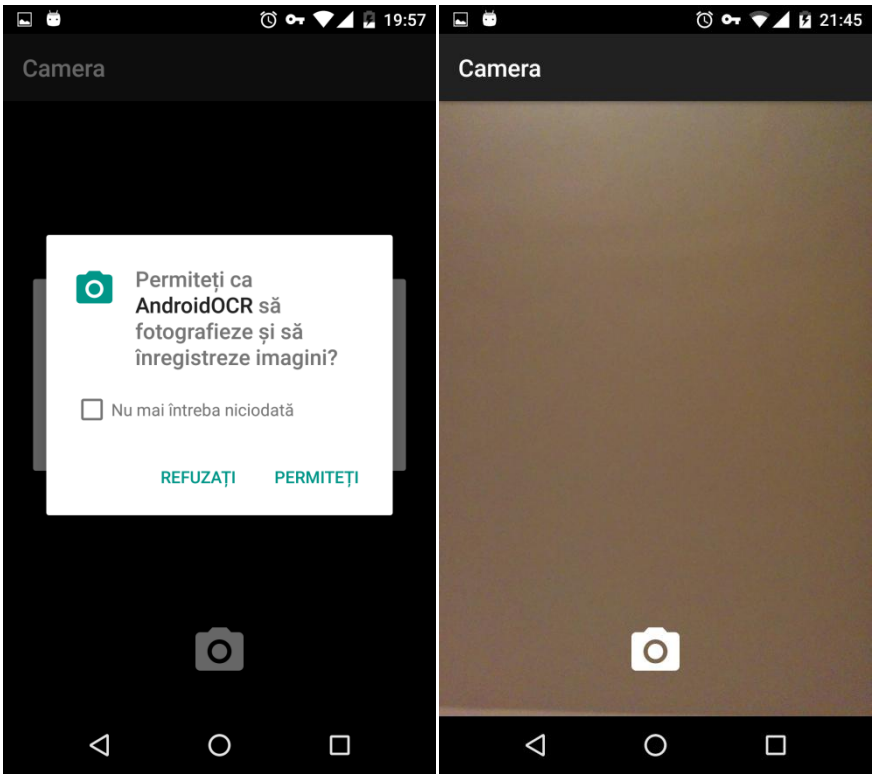
Fereastra de logare în aplicație este asemănătoare cu cea de înregistrare în aplicație. În cadrul acestei ferestre utilizatorul trebuie să-și introducă adresa de email și parola pentru a se loga. Dacă logarea este reușită atunci utilizatorul este trimis la fereastra principală a aplicației, altfel i se prezintă un mesaj.



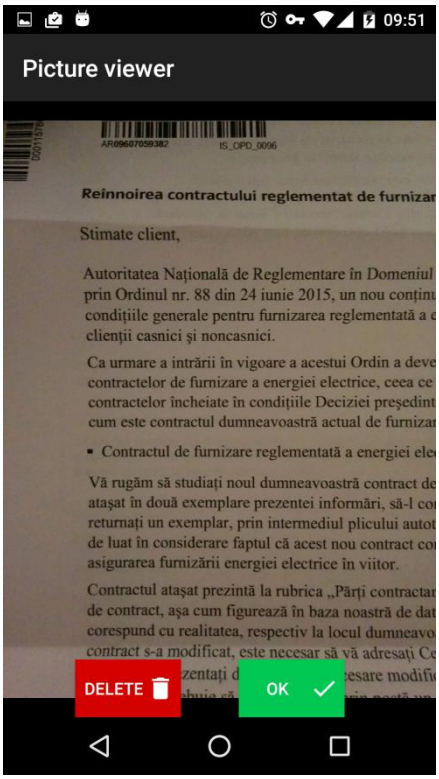
Fereastra principală a aplicației este une simplă și intuitivă în care utilizatorului, în funcție de rolul pe care îl îndeplinește în aplicație, îi sunt prezentate funcționalitățile pe care le poate accesa. Astfel, un utilizator normal nu poate accesa funcția de administrare. Mai jos este prezentată interfața pe care o vede un utilizator normal, în stânga, și cea pe care o vede un administrator.



În cazul în care utilizatorul alege opțiunea *TAKE PIC* atunci este redirecționat la ecranul în care este implementată o camera de fotografiat simplistă cu o interfață foarte simplă în care utilizatorul poate realiza fotografii. În cazul în care acesta nu a acordat permisiuni pentru cameră acestea îi sunt solicitate într-un dialog. După cum se observă în imaginile de mai jos, fereastra are un singur buton folosit pentru a realiza o fotografie.

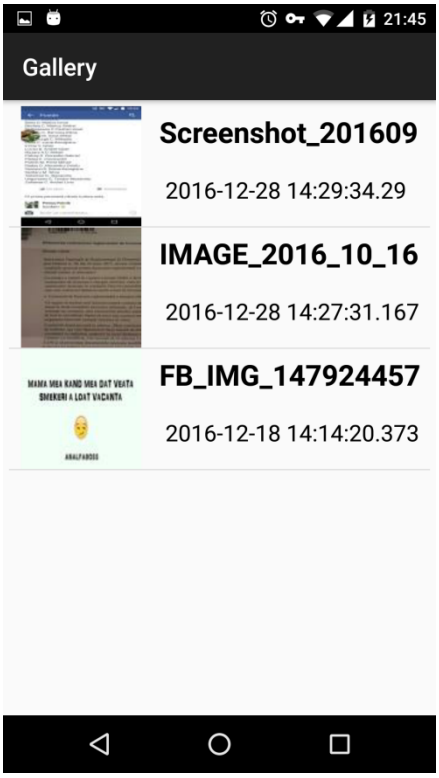


După ce utilizatorul a realizat o fotografie, aceasta îi va fi afișată pe ecran, iar acesta are la dispoziție opțiunea de a șterge imaginea făcută, să accepte imaginea și să pornească procesul de recunoaștere optică a caracterelor, sau să meargă înapoi la ecranul precedent prin intermediul butonului *back* și să păstreze imaginea în memoria telefonului.

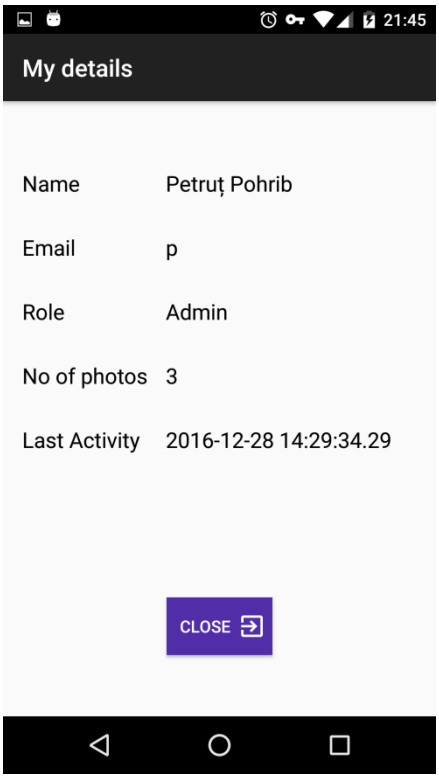


Utilizatorul mai are la dispoziție opțiunea de a încărcă o fotografie din memoria telefonului. Astfel, când acesta apasă butonul *BROWSE* din ecranul principal al aplicației, exploratorul de fișiere implicit al sitemului va fi afișat pe ecran, iar când utilizatorul va selecta o imagine aceasta va fi afișată în vizualizatorul de fotografii al aplicației prezentat mai sus.

Atunci când utilizatorul va apăsa butonul *GALLERY* din ecranul principal al aplicației, acestuia îi vor fi afișate toate imaginile pe care acesta le-a încărcat pentru a fi procesate de API. Astfel utilizatorul poate vizualiza un istoric al activității acestuia și poate executa din nou procesul de recunoaștere optică de caractere pe o imagine deja încărcată. Galeria de imagini are un design simplu în care imaginile încărcate sunt afișate într-o listă, unde fiecare imagine are numele acesteia, data la care a fost încărcată și un *thumbnail*. Când o imagine este selectată din galerie, aceasta este afișată în vizualizatorul de fotografii prezentat mai sus.

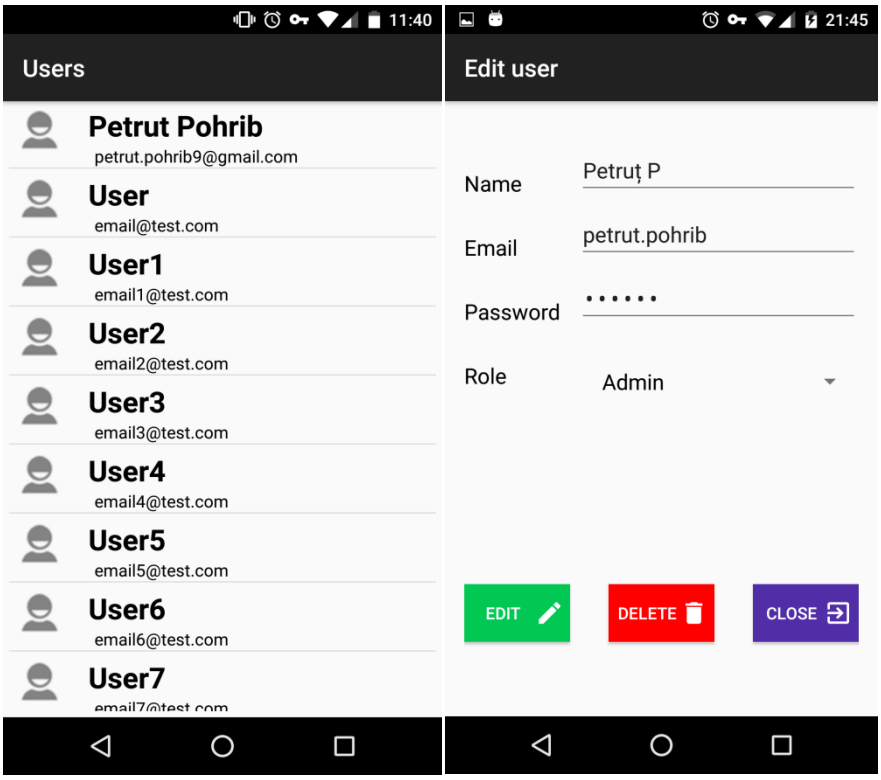


Mai avem pe ecranul pricipal o opțiune de a vedea detalii despre contul utilizatorului. Aici există detalii despre numele, emailul utilizatorului, numărul de fotografii existente pe contul acestuia, data ultimei activități.

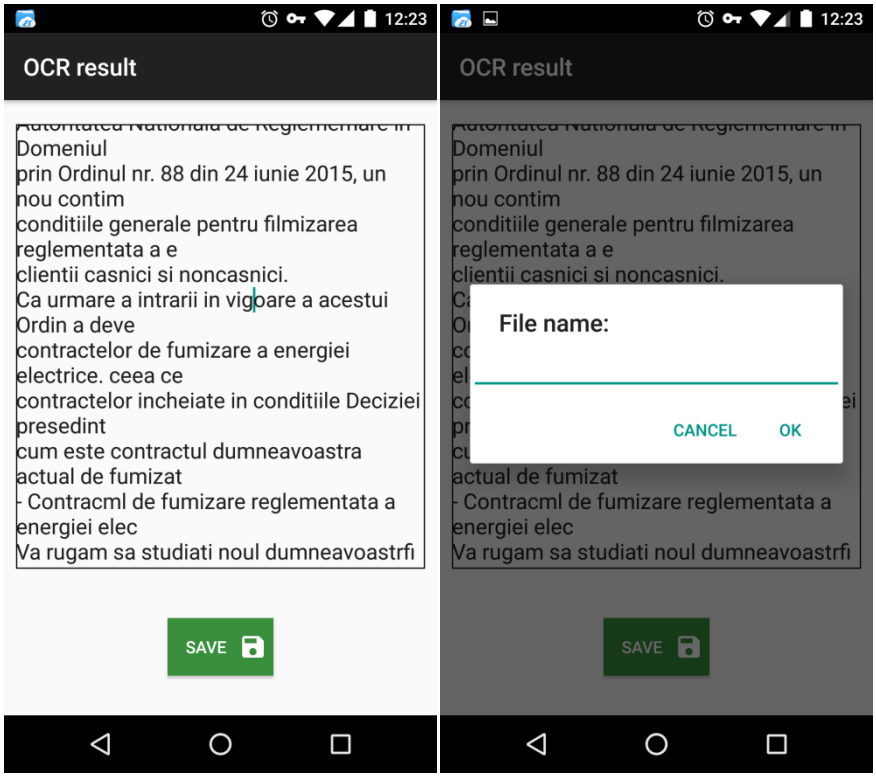


Avem deasemenea opționea de *LOGOUT*. Atunci când această opțiune este selectată, datele stocate în sesiunea curentă sunt eliminate, iar utilizatorul este trimis la primul ecran din apicație.

În cazul în care un utilizator logat are rol de admin în cadrul aplicației, atunci acesta are la dispoziție butonul *ADMIN*. Când acesta a fost apăsat, utilizatorul vede o listă cu toți utilizatorii aplicației. Acesta poate selecta un utilizator din acea listă și este redirectat la un ecran de unde poate edita utilizatorul selectat, schimba rolul acestuia și îl poate șterge.



Rezultatele procesării de imagine și afișarea rezultatului se va face în următorul fragment. Tot de aici, utilizatorul va putea salva rezultatul obținut într-un fișier text.



4 Concluzii

Aplicațiile prezentate în prezenta lucrare au fost concepute pentru a facilita accesul la o tehnologie foarte utilă în practică dar despre care nu se vorbește foarte mult și de a înțelege problemele care apar în implementarea unei aplicații care să fie capabilă de recunoaștere optică de caractere. De asemenea, prin implementarea acestor aplicații am dorit să învăț ca programator lucrul cu diferite tehnologii și arhitecturi care au fost explicate pe parcursul lucrării.

Implementarea API-ului a ridicat anumite probleme deoarece am dorit să lucrez cu șabloane arhitecturale foarte moderne și să folosesc tot felul de tehnologii care se folosesc foarte des în industria IT și să construiesc un API robust, rapid, ușor de testat și ușor de înțeles pentru cei care se vor uita pe codul sursă. De asemenea, am învățat anumite lucruri despre procesarea de imagini, lucruri de care nu eram conștient înainte. Am avut multe de învățat de pe urma acestei aplicații și sunt mulțumit de modul în care aceasta arată în momentul de față.

Implementarea clientului de Android a ridicat de asemenea unele probleme întrucât nu am lucrat cu platforma și nici limbajul Java nu este unul din punctele mele forte. Dar o dată ce am înțeles modul în care aplicațiile Android funcționează și arhitectura generală a unei aplicații totul a decurs foarte bine. Am reușit să accelerez timpii de descărcare a resurselor de la API prin intermediul intergrării unor librării moderne și modul în care aplicația reacționează la comenzile utilizatorului.

În concluzie, aplicațiile prezentate în prezenta lucrare sunt niște aplicații care folosesc tehnologii foarte moderne din care am avut foarte multe de învățat.

Bibliografie

Abbyy. (2016, 11 13). Retrieved 11 13, 2016, from Abbyy: https://www.abbyy.com/finereader/about-ocr/what-is-ocr/

Blog Web API. (2016, 11 13). Retrieved 11 13, 2016, from Blog Microsoft: https://blogs.msdn.microsoft.com/martinkearn/2015/01/05/introduction-to-rest-and-net-web-api/

Entity Framework. (2016, 11 13). Retrieved 11 13, 2016, from Entity Framework: http://www.entityframeworktutorial.net/what-is-entityframework.aspx

Fluent Validation. (2016, 11 13). Retrieved 11 13, 2016, from Github: https://github.com/JeremySkinner/FluentValidation

Frăsinaru, C. (2016). *curs Programare Avansată.* Iași: Facultatea de Informatică, Universitatea „Alexandru Ioan Cuza" Iași .

How to deskew an image. (2006, Aprilie 25). Retrieved Decembrie 10, 2016, from Code project: https://www.codeproject.com/kb/graphics/deskew_an_image.aspx

Joudeh, T. (2014, Iunie 1). *Token Based Authentication using ASP.NET Web API 2, Owin, and Identity*. Retrieved November 16, 2016, from Bit of technology: http://bitoftech.net/2014/06/01/token-based-authentication-asp-net-web-api-2-owin-asp-net-identity/

Microsoft .NET Core. (2016, 11 13). Retrieved 11 13, 2016, from Microsoft: https://www.microsoft.com/net/core#windows

Microsoft. (2016, 11 13). Retrieved 11 13, 2016, from Microsoft: https://msdn.microsoft.com/en-us/library/z1zx9t92.aspx

Retrofit. (2016, 11 13). Retrieved 11 13, 2016, from Retrofit: https://square.github.io/retrofit/

Stasch, M. (2015, Aprilie 10). *CQRS - Simple architecture*. Retrieved Ianuarie 9, 2017, from Future processing: https://www.future-processing.pl/blog/cqrs-simple-architecture/

Statista. (2016, 11 13). Retrieved 11 13, 2016, from Statista: : https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

Wikipedia. (2016, 11 13). Retrieved 11 13, 2016, from Java (programming language): https://en.wikipedia.org/wiki/Java_(programming_language)