UNIVERSITATEA BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA SISTEME DISTRIBUITE

LIBRĂRIE iOS PENTRU DEZVOLTAREA DE APLICAȚII ÎN LIMBAJUL SWIFT

LUCRARE DE DISERTAȚIE

Coordonator
Conf.dr. Andrei Păun

Absolvent
Florian Marcu

Cuprins

1. Introducere

Odată cu apariția și dezvoltarea explozivă a smartphone-urilor, aplicațiile pentru mobile au câștigat foarte multă popularitate. Dispozitivele mobile, precum telefoanele și tabletele, sunt din ce în ce mai des utilizate, motiv care explică și investirea multor resurse de către companii în îmbunătățirea acestora. S-a ajuns așadar la o performanță foarte bună a acestor dispozitive, fapt care încurajează din ce în ce mai mult utilizatorii să folosească smartphone-uri și să renunțe la variantele clasice pentru a-și întreprinde sarcinile de zi cu zi din realitatea virtuală (jocuri, email-uri, rețele sociale, știri, etc).

Cei mai mari doi giganți din industria sistemelor de operare pentru aplicații mobile sunt sistemele de operare dezvoltate de către Google și Apple, Android, respectiv iOS. Modul în care aplicațiile sunt distribuite deținătorilor de smartphone-uri este similar pentru toate tipurile de dispozitive. Anume, fiecare companie deține un magazin virtual, unde dezvoltatorii de aplicații își pun la dispoziție aplicațiile (cu prezentări, preț, actualizări, etc) și de unde utilizatorii normali își pot descărca/cumpăra aplicațiile dorite.

Apple Store este magazinul virtual deținut de Apple, pe când Google Play este magazinul celor de la Google. Exista și alte magazine virtuale pentru aplicații, dintre care cel mai important este magazinul celor de la Microsoft, pentru aplicații de telefoane ce rulează pe sistemul de operare Windows Phone (al cărui număr de dispozitive distribuite este total nesemnificativ în comparație cu cei doi giganți, dar care este totuși în creștere).

Deși dispozitivele ce rulează pe Android sunt mult mai răspândite decât iPhone-urile și iPad-urile, smartphone-urile, precum și sistemul de operare iOS, puse la dispoziție de Apple, sunt caracterizate printr-o stabilitate mult mai mare, consistență sporită, un design superior și o oarecare ușurință în dezvoltarea de aplicații, comparativ cu tabletele și telefoanele ce rulează pe Android. Instrumentele puse la dispoziție dezvoltatorilor de aplicații iOS sunt clar superioare celor existente pe Android. Explicația constă în faptul ca Android-ul este folosit de către foarte mulți producători de telefoane (LG, Samsung, Motorola, HTC, etc), fiecare dezvoltând separat varianta open-source de Android, pusă la dispoziție de Google. De asemenea, există o varietate enormă în dimensiunea telefoanelor, configurația procesoarelor și a memoriei, descrierea ecranelor, fapt care îngreunează foarte mult dezvoltarea de aplicații pe Android.

Consistența dispozitivelor iOS încurajează foarte mult utilizatorii să facă trecerea către iPhone și iPad. În plus, veniturile întregistrate de aplicațiile din Apple Store sunt copleșitor superioare celor din Google Play, fapt care încurajează orice startup de tehnologie să înceapă cu o aplicație pe Apple Store, în detrimentul unei aplicații pe Google Play. Această diferență de încasări se explică prin faptul că, deși dispozitivele Android sunt mult mai populare la nivel

internațional, în cadrul Statelor Unite domină iPhone-urile, iar cetățenii americani au o forță de cumpărare mult mai mare decât celelalte țări.

Se conturează așadar nevoia existenței oricărei afaceri sau startup pe Apple Store. Toți marii giganți din industria IT și-au dezvoltat deja aplicații pentru iPhone-uri. Facebook, Twitter, Google, Microsoft, Dropbox, etc au deja aplicații diverse pe Apple Store, deși în trecut aceștia mizau doar pe Web. Web-ul pierde din ce în ce mai mult din popularitate, iar acest declin este susținut și de evoluția progresivă a datelor de mobile 3G, 4G și LTE. Internet-ul mobil este din ce în ce mai accesibil, lucru care determină o schimbare radicală a lumii virtuale așa cum era ea cunoscută în urmă cu câțiva ani. Pe scurt, un utilizator de iPhone mediu, folosește mobilul pentru a-și citi mailurile (Gmail, Mailbox, etc), pentru a interacționa cu prietenii (Facebook), pentru a urmări ce se întâmplă în lume și a sta la curent cu lucrurile de care e interesat (Twitter), pentru a-și trimite poze prietenilor apropiați (Snapchat), pentru a-și organiza sarcinile zilnice (Asana, Trello, etc), pentru a vorbi cu oamenii ce se află departe în lume (Skype, Facetime) etc. Nici macar SMS-ul clasic, cum îl știam în urmă cu 2 ani, nu va mai rezista foarte mult, locul acestuia fiind luat de aplicații precum WhatsApp și Vibe.

Așadar, toate marile companii care dominau Web-ul în urmă cu puțini ani, au făcut trecerea către Apple Store. Și rezultatele sunt grozave, în termeni de profit și de creștere a masei de utilizatori. De exemplu, peste 60% din traficul Twitter vine de pe mobil!

Dezvoltarea unei aplicații pentru iPhone sau iPad se făcea în limbajul Objective-C, un limbaj care extinde funcționalitatea limbajului C. Este un limbaj de scop general, orientat pe obiecte, pe care se bazează API-urile create de Apple, Cocoa și Cocoa Touch.

Dar recent, Apple și-a dezvoltat propriul limbaj de programare pentru crearea de aplicații mobile, denumit Swift. Acesta este un limbaj inovativ pentru Cocoa și Cocoa Touch, interactiv, a cărui sintaxă este foarte concisă și a cărui eficiență este sporită, comparativ cu Objective C. Marele avantaj este că acest limbaj este integrat cu mare ușurință cu Objective C, putându-se folosi clase și metode comune.

Swift fiind un limbaj recent (a apărut în urmă cu mai puțin de un an), librăriile în Swift pentru lucruri foarte comune sunt greu de găsit. Într-adevăr, se pot folosi librăriile de Objective-C, care e un limbaj matur, foarte răspândit, dar acest lucru nu poate dura la nesfârșit, Apple încurajând deprecierea acestuia. În plus, chiar și în Objective-C, lipsesc lucruri elementare, comune aproape tuturor aplicațiilor, fiecare dezvoltator fiind pus în situația reinventării roții. Apple pune la dispoziție într-adevar, destule resurse prin intermediul Cocoa și Cocoa Touch, oferind programatorilor multă libertate și librării pe care să construiască. Dar majoritatea acestor librării nu fac decât să impună anumite paradigme de programare (pentru consistența generală a aplicațiilor) și să ușureze

comunicarea dintre codul aplicației cu hardware-ul efectiv (camera foto, agenda de contacte, rețeaua Wi-fi, etc). În momentul în care ne uităm la un strat mai sus, observăm că instrumentele nu sunt suficiente.

Să considerăm, de exemplu, o funcționalitate care este răspândită în peste 70% din aplicațiile sociale foarte populare - *pull to refresh*. Aceasta constă în acțiunea utilizatorului de a trage un ecran în jos, iar în momentul în care utilizatorul ridică degetul de pe touchscreen, ecranul respectiv se actualizează (făcând o cerere la un server, primind înapoi rezultatele și actualizând interfața). Acesta este modul în care un utilizator primește postări noi de la prieteni pe Facebook sau tweet-uri noi de la followeri pe Twitter. Această funcționalitate există în peste 70% din alte aplicații sociale (Tinder, Secret, Whisper, Snapchat, etc). Totuși, ea nu este suportată în interfața de programare Apple. O simplă căutare pe web, ne arată o implementare într-un proiect de Github, în Objective-C, care nu respectă nici măcar cele mai simple și comune principii de MVC (model - view - controller), și care nici măcar nu este extensibilă cu ușurință. Iată așadar un caz în care roata a fost reinventată de 70% din companiile ce dețin aplicații sociale.

Lucrarea de față prezintă o librărie dezvoltată în limbajul de programare Swift, prin care se propune implementarea celor mai populare funcționalități existente în cadrul aplicațiilor din Apple Store și care nu sunt deja suportate de către interfețele de programare puse la dispoziție de Apple.

Pentru un dezvoltator care va utiliza librăria în cauză, sarcina creării unei aplicații de iOS va fi ușurată semnificativ, atât ca timp, cât și ca resurse necesare. De asemenea, această librărie va adăuga un plus de consistență și stabilitate aplicațiilor, evitând bug-urile comune care sunt specifice acestor funcționalități populare. Framework-ul va servi ca un schelet pentru orice aplicație nouă.

Se creionează așadar rolul important al acestei librării în cadrul unui start up, care se află la început de drum. Într-un astfel de context, resursele sunt foarte limitate (timp, bani, programatori, etc). Este așadar vitală existența unui schelet pentru o aplicație de iOS, care să ușureze crearea unui produs de start (*minimum viable product*) și care să evite greșelile des întalnite (inconsistență, bug-uri, lipsa de teste, etc).

În capitolul al doilea va fi prezentat limbajul Swift, pentru a servi ca o pregătire pentru capitolele următoare, unde va fi prezentată efectiv librăria. Așadar, capitolul al doilea va conține o scurtă descriere a limbajului Objective-C, a interfețelor de programare (API) Cocoa și Cocoa Touch, urmând apoi ca limbajul Swift să fie prezentat. De asemenea, va fi explicată și pe larg motivația celor de la Apple pentru tranziția pe Swift. Deși Swift nu este în totalitate un limbaj funcțional, capitolul al doilea va surprinde câteva paradigme interesante de programare funcțională existente în Swift. În plus, Swift fiind un limbaj relativ

nou, este încă incomplet, la început de drum și are destul de multe bug-uri. Toate aceste lucruri vor fi și ele enumerate în capitolul al doilea.

Capitolul al treilea este capitolul central al acestei lucrări, întrucât conține prezentarea concretă a framework-ului Nucleus, introdus mai sus. Începând cu descrierea generală, vom continua cu structura proiectului în sine, după care vom prezenta funcționalitățile de bază încorporate pentru început în librărie. Vor fi prezentate arhitectura, implementarea și modul de folosire pentru funcționalități diverse, precum *pull-to-refresh* (descris mai sus), streaming, manipulare de date JSON, download și caching de imagini, comunicare cu servere, interacțiunea dintre tabele din interfață și modele, precum și alte plugin-uri, dintre care enumerăm existența unui chat, suport de video-uri Youtube și crearea cu ușurință a formularelor.

În cel de-al patrulea capitol este exemplificat în detaliu modul de utilizare al framework-ului Nucleus, considerând un caz practic de aplicație. În principiu, vom detalia crearea unei noi aplicații, care va folosi librăria în cauză. Ne vom referi la aplicația luată drept exemplu prin *Abroad* și ne propunem realizarea unei aplicații de iOS, care să faciliteze comunicarea și descoperirea de către conaționali din jurul utilizatorului. Evident, ideea aplicației este adresată oamenilor care locuiesc în străinătate. Partea cea mai importantă a acestei idei este faptul că reprezintă un exemplu foarte tipic tuturor aplicațiilor sociale existente momentan. Vom observa ușurința modificării acestei aplicații pentru a crea Facebook, Twitter, Snapchat, Secret, etc. Acest lucru se bazează pe faptul că toate funcționalitățile sunt consistente, iar workflow-urile utilizatorilor sunt foarte asemănătoare. Pe scurt, în toate aceste aplicații, utilizatorul se loghează, are posibilitatea de a posta, de a comenta la postări, de a-și actualiza profilul și de a vorbi cu alți utilizatori. Diferă doar ariile de utilizatori cu care poate vorbi (prieteni vs. followeri vs. oameni din jur, etc), gradul de vizibilitate al postărilor (prieteni vs. followings vs. oameni din jur, etc) și concepte de denumiri (status vs. tweet vs. post, etc). La o analiză atentă, putem observa că toate aceste diferențe nu țin de partea de client (aplicația iOS în sine), ci mai degrabă de server (modul în care procesează și returnează date), lucru care întărește și mai mult necesitatea existenței unei platforme precum *Nucleus*.

Tot în cel de-al patrulea capitol va fi prezentată pe scurt și partea de server care susține aplicația *Abroad*. Vor fi prezentate scurte introduceri în Play Framework, Scala și MySQL, tehnologii care permit crearea cu mare ușurință de către sisteme distribuite scalabile, care să suporte milioane de cereri pe minut.

2. Dezvoltarea aplicațiilor iOS

2.1 Descriere generală

iOS este un sistem de operare creat și dezvoltat de Apple Inc. și distribuit exclusiv pe hardware Apple. Este sistemul de operare care în prezent rulează pe mult dispozitive Apple, inclusiv iPhone, iPad și iPod touch.

A fost introdus pe piață în anul 2007, odată cu apariția iPhone-ului, iar de atunci a fost extins pentru a suporta și alte dispozitive Apple, precum iPod Touch (septembrie 2007), iPad (ianuarie 2010), iPad Mini (noiembrie 2012) și a doua generație de Apple TV (septembrie 2010). Din ianuarie 2015, în Apple Store existau peste 1.4 milioane de aplicații iOS, dintre care 725.000 sunt native pentru iPad. Aceste aplicații au fost descărcate colectiv de peste 75 de miliarde de ori. Apple a deținut 21% din smartphone-urile vândute în al patrulea trimestru din 2012, fiind depășită doar de Android. Pe 27 ianuarie, 2015, Apple a anunțat că aveau vândute peste un miliard de dispozitive iOS.

Interfața utilizatorului pe iOS este bazată pe un concept de manipulare directă, folosind gesturi *multi-touch*. Elementele de control ale interfeței sunt alcătuite din slidere, întrerupătoare și butoane. Interacțiunea cu sistemul de operare include gesturi precum *swipe*, *tap*, *pinch* și *reverse pinch*, toate acestea având definiții și comportamente specifice în cadrul sistemului de operare și a interfeței *multi-touch*. Accelerometrele interne sunt folosite de unele aplicații pentru a răspunde la scuturarea dispozitivului sau rotirea în trei dimensiuni (schimbarea de la modul portret la modul peisaj, de exemplu).

Sistemul de operare iOS are câteva elemente în comun cu sistemul de operare OS X, cum ar fi *Core Foundation* și *Foundation*. Totuși, instrumentele de user interface sunt conținute în Cocoa Touch, și nu în Cocoa din OS X, prin urmare furnizează framework-ul UIKit, și nu framework-ul AppKit. Din acest motiv, nu este compatibil cu OS X, pentru aplicații. În plus, liniile de comandă din Unix nu sunt disponibile pentru utilizatori și sunt restricționate pentru aplicații, facând astfel iOS incompatibil cu Unix.

Anual, se lansează versiuni majore noi. Versiunea lansată curent este iOS 8.3, apărută pe 8 aprilie, 2015.

În iOS, există 4 straturi de abstracție:

- Core OS layer
- Core Services layer
- Media layer
- Cocoa Touch Layer

2.2 Objective-C

Objective-C este principalul limbaj de programare folosit în scrierea de software pentru sistemele de operare OS X și iOS. Este un limbaj construit peste limbajul de programare C și furnizează abilitări de orientare-obiect și o dinamică în timpul de rulare. Objective-C moștenește sintaxa, tipurile primitive și instrucțiunile de control din C, peste care adaugă sintaxa de definire a claselor și metodelor. De asemenea, mai adaugă și suport la nivel de limbaj pentru graful de manipulare a obiectelor și a literalilor, în timp ce furnizează și tipizare și legare dinamică, amânând multe resposabilități până la timpul de rulare. Printre lucrurile cele mai importante specifice limbajului Objective-C (adăugate peste C), amintim clasele (cu interfețe și implementări), proprietățile, metodele, protocoalele, categoriile, blocurile și excepțiile.

Interesant de menționat este că Objective-C este considerat un *superset* al limbajului C, ceea ce înseamnă ca orice program scris în C poate fi compilat cu un compilator de Objective-C.

Vom prezenta în continuare, sumar, principalele caracteristici ale acestui limbaj.

2.2.1 Interfețe și implementări

Objective-C cere ca interfața și implementarea unei clase să fie blocuri de cod declarate separat. Prin convenție, dezvoltatorii plasează interfața într-un fișier header și implementarea într-un fișier de cod. Fișierele header au în mod normal extensia .h (similar cu fișierele header din C), pe când fișierele de implementare au extensia .m (pentru a fi asemănătoare cu fișierele de cod C).

```
@interface classname : superclassname {
 // instance variables
}
+ classMethod1;
+ (return_type)classMethod2;
+ (return_type)classMethod3:(param1_type)param1_varName;

- (return_type)instanceMethod1With1Parameter:(param1_type)param1_varName;
- (return_type)instanceMethod2With2Parameters:(param1_type)param1_varName param2_callName:(param2_type)param2_varName;
@end
```

În exemplul prezentat deasupra, semnul “+” denotă o metodă a clasei (statică), iar semnul “-“ denotă o metodă de instanță, care poate fi deci apelată pe o instanță particulară a acelei clase.

```
@implementation classname
+ (return_type)classMethod
{
 // implementation
}
- (return_type)instanceMethod
{
 // implementation
}
@end
```

În figura de deasupra, este ilustrată sintaxa declarării părții de implementare a unei clasei.

2.2.2 Protocoale

În alte limbaje de programare (ca Java, de exemplu), protocoalele sunt denumite interfețe. Un protocol este o listă de metode și proprietăți pe care o clasă poate/trebuie să le implementeze.

Declararea unui protocol se face ca în modul următor:

```
@protocol NSLocking
- (void)lock;
- (void)unlock;
@end
```

Iar interfața unei clase care implementează protocolul declarat mai sus, arată ca în figura următoare:

```
@interface NSLock : NSObject <NSLocking>
//...
@end
```

2.2.3 Forwarding

Limbajul Objective-C permite trimierea unui mesaj către un obiect care s-ar putea să nu răspundă. În loc de a răspunde sau a omite mesajul, un obiect poate să-l trimită mai departe spre un alt obiect care ar putea răspunde. Tehnica

de *forwarding* poate fi folosită pentru a simplifica implementarea unor paradigme de programare, precum paradigma observatorului sau a proxy-ului.

2.2.4 Categorii

În momentul proiectării limbajului Objective-C, una dintre principalele preocupări a fost menținerea unui codebase foarte mare. Experiența venită din lumea programării structurate a demonstrat că unul dintre principalele moduri de îmbunătățire a codului este împărțirea acestuia în bucăți cât mai mici. Objective-C a împrumutat și extins conceptul de categorii de la implementările Smalltalk, pentru a ajuta cu acest proces.

Așadar, metodele din cadrul unei categorii sunt adăugate unei clase, la *runtime*. Categoriile permit așadar programatorului să adauge metode unei clase deja existente, fără a fi nevoit să recompileze acea clasa sau fără măcar să-i acceseze codul sursă. De exemplu, dacă un sistem nu conține o verificare de corectitudine a clasei String, el poate să adauge aceasta verificare fără modificarea sau accesarea codului sursă al clasei String.

Metodele din categorii devin de nedistins cu metodele din clasă, în momentul rulării programului. O categorie are acces complet la toate variabilele de instanță din interiorul clasei, inclusiv la variabilele private.

Blocuri

Blocurile sunt extensii non-standard ale Objective-C, care folosesc o sintaxă specială pentru a crea închideri (*closures*).

2.2.5 Automatic Reference Counting

ARC este o funcționalitate *compile-time*, care elimină necesitatea ca programatorii să administreze manual lucrul cu referințele de memorie (exercitat în trecut prin *retain* și *release*). Spre deosebire de *garbage collection*, care apare la runtime, ARC elimină consumul de resurse al unui proces separat de gestionare a referințelor. ARC și gestionarea manuală a memoriei nu sunt mutual exclusive, în sensul că programatorii pot continua să folosească cod non-ARC în proiecte care au funcționalitatea ARC activată.

2.3 Cocoa și Cocoa Touch

2.3.1 Descriere generală

Cocoa este API-ul (application programming interface) nativ, pus la dispoziție de Apple pentru sistemul de operare OSX. Pentru iOS, API-ul asemănător este numit Cocoa Touch, care include animații, recunoașterea gesturilor și un set diferit de controale grafice folosite pe dispozitive ce rulează pe sistemul de operare iOS.

Cocoa este alcătuită din trei principale librării de Objective-C, care sunt distribuite, compilate și care pot fi încărcate dinamic într-un program. Acestea sunt:

- ***Foundation Kit***. Este bazată pe Core Foundation și este o librărie generică, orientată pe obiect, furnizând instrumente pentru manipularea șirurilor de caractere, a containerelor și iteratorilor, calcul distribuit, cicluri și alte funcții care nu sunt direct legate de interfața grafică.
- ***Application Kit***. Această librărie conține cod folosit de programe pentru a crea și interacționa cu interfața grafică a utilizatorului.
- ***Core Data***. Este o librărie persistentă inclusă cu *Foundation* și *Cocoa*.

Cocoa Touch furnizează un strat de abstracție pentru iOS. El este bazat pe setul de instrumente Mac OS X Cocoa API și, precum acesta, este scris în Objective-C. De asemenea, Cocoa Touch permite folosirea hardware-ului și a funcționalităților găsite doar pe dispozitivele specifice iOS. Principalele funcționalități ale Cocoa Touch sunt:

- Core Animation
- Multitasking
- Gesture Recognizers

În plus, Cocoa Touch furnizează librării importante pentru dezvoltarea de aplicații iOS. Printre cele mai importante amintim:

- Foundation Kit Framework
- UIKit Framework
- GameKit Framework
- iAd Framework
- MapKit Framework

2.3.2 Paradigma Model-View-Controller

Cocoa Touch are la bază paradigma MVC (model-view-controller), care facilitează foarte mult dezvoltarea și reutilizarea de cod. Conceptele de bază ale unei aplicații sunt împărțite în trei clase de obiecte:

- ***Models***, care sunt clase ce reprezintă date neprelucrate, precum documente, setări, fișiere sau obiecte din memorie.
- ***Views***, care, precum indică și denumirea, sunt reprezentări vizuale ale modelelor de date
- ***Controllers***, sunt clasele ce conțin logica ce leagă modelele de views și menține starea acestora sincronizată.

Arhitectura Cocoa este o aplicație strictă de MVC. Prin furnizarea acestor librării, împreună cu suport pentru toate cele trei straturi MVC, Apple are scopul de a reduce cantitatea de cod pe care dezvoltatorii trebuie să o scrie.

Tot pe pattern-ul MVC se bazează și iOS SDK (*Software Development Kit*), librăria de bază pe care dezvoltatorii iOS o utilizează pentru a crea aplicații mobile pentru iPhone. Prezentăm în continuare câteva din elementele de bază care au fost folosite și îmbogățite de către librăria Nucleus, pentru a crea o imagine amplă a modului în care se îmbină elementele implementate.

În primul rând, ilustrăm, printr-o diagrama, paradigma MVC din iOS SDK, care a fost puțin modificată față de cea clasică:

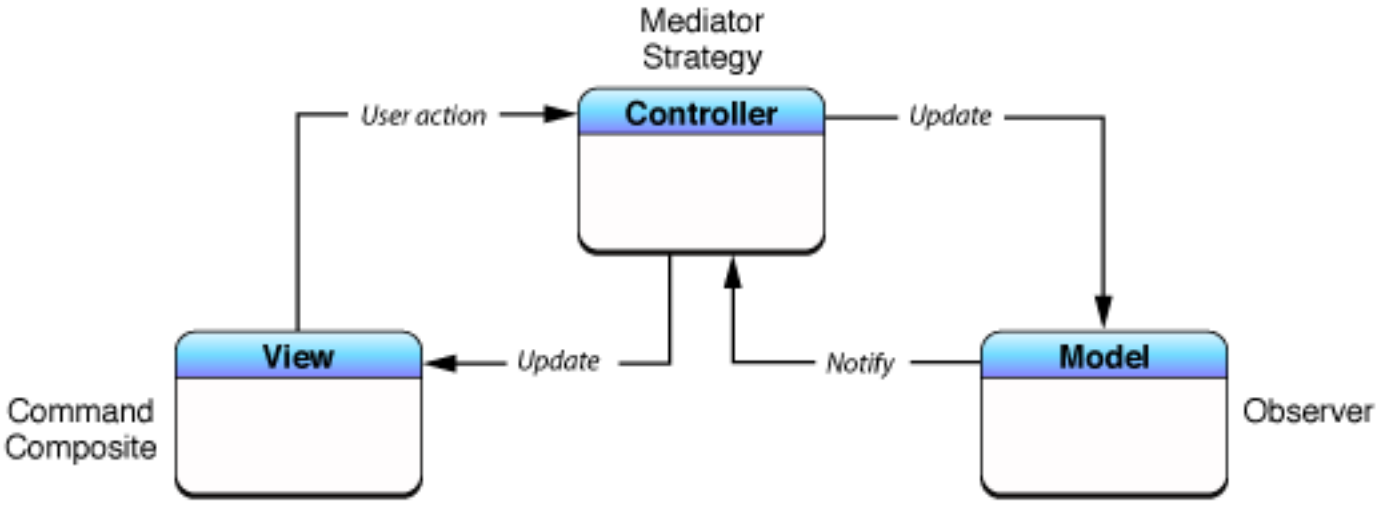

Figura 1

Pentru o comparație și mai ușoară, ilustrăm aici și varianta clasică de MVC:

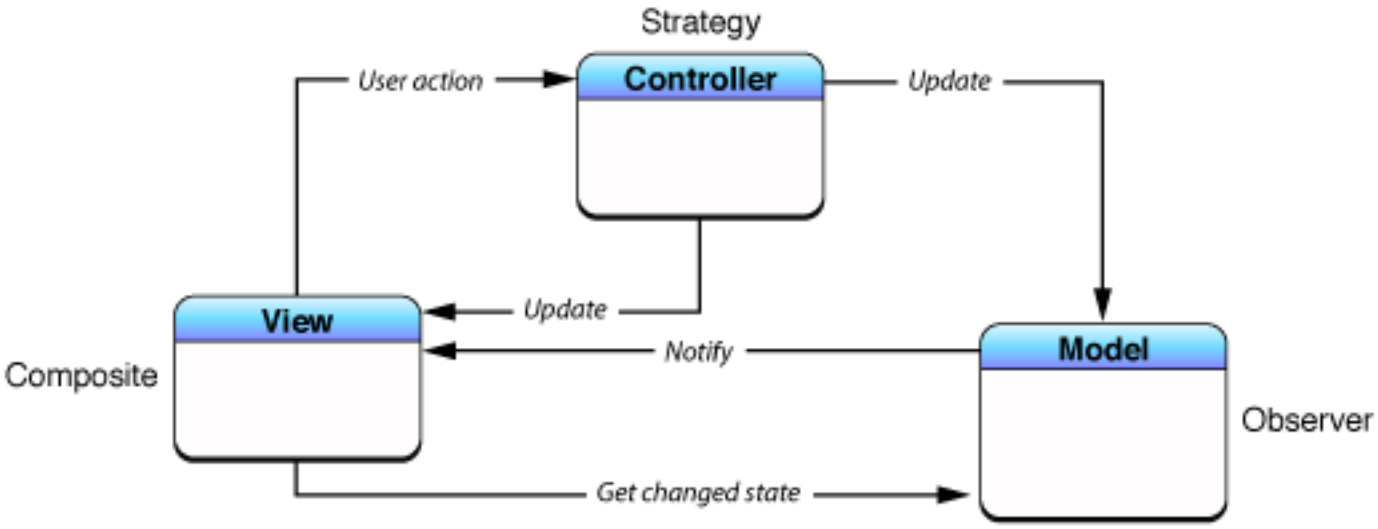

Figura 2

Model-View-Controller este o paradigma care este compusă din câteva alte paradigme esențiale. Aceste paradigme lucrează împreună, pentru a defini o separare funcțională și niște căi de comunicare ce sunt caracteristice unei aplicații MVC. Totuși, noțiunea tradițională asociază un set de paradigme elementare, diferite față de cele asociate de Cocoa. Diferența principală constă în rolurile date obiectelor de control și vizualizare.

În concepția originală, MVC este creat din paradigmele *Composite*, *Strategy* și *Observer*:

- ***Composite***: Obiectele de vizualizare într-o aplicație sunt de fapt o compunere de vizualizări îmbricate, care lucrează împreună în mod coordonat (*ierarhia view-urilor*). Aceste componente de interfață variază de la ferestre de afișare, precum table view-uri la view-uri individuale, precum butoane. Elementele de introducere de date și afișare se pot afla la orice nivel al structurii compuse.
- ***Strategy***: Un obiect de control implementează o strategie pentru unul sau mai multe obiecte de vizualizare. Obiectul de vizualizare se auto-limitează pentru a-și menține aspectele vizuale și delegă către controller toate deciziile despre comportamentul specific interfeței aplicației.
- ***Obsever***: Un obiect model păstrează obiectele interesate de aplicație, de obicei obiecte de vizualizare și le anunță de modificări ale stării.

Acest mod tradițional, folosind *Composite*, *Strategy* și *Obsever* este ilustrat in *Figura 2*: utilizatorul manipulează un view la un anumit nivel din structura *composite* și ca rezultat, un eveniment este generat. Un obiect de control primește acest eveniment și îl interpretează în aplicație, deci aplică o strategie. Această strategie poate să ceară (printr-un mesaj) unui obiect de tip model să-și schimbe starea sau să ceară unui obiect de vizualizare să-și schimbe aspectul sau comportamentul. Obiectul model, la rândul lui, notifică toate obiectele care sunt înregistrate ca și observatori, când își schimbă starea. Daca un observator este un obiect de vizualizare, acesta poate să-și schimbe aspectul în mod corespunzător.

Versiunea de MVC din Cocoa are câteva similarități cu versiunea tradițională și, de fapt, este chiar posibilă construirea unei aplicații bazată pe diagrama din *Figura 2*. Prin folosirea tehnologiei de legare, se poate crea cu ușurință o aplicație Cocoa MVC ale cărui view-uri să observe în mod direct obiectele model, pentru a primi notificări de schimbări ale stării. Totuși, există o problemă teoretică cu aceasta proiecție. Obiectele de vizualizare și modelele ar trebui să fie cele mai reutilizabile din aplicație. Obiectele de vizualizare reprezintă *look-and-feel*-ul sistemului de operare și al aplicațiilor pe care sistemul le suportă; consistența interfeței și comportamentul este esențial și acest lucru cere ca obiectele să fie cât mai reutilizabile. Modelele, prin definiție, încapsulează datele asociate unei probleme și execută operații pe acele date. Din punct de vedere al arhitecturii,

este cel mai bine ca modelele să fie separate total de către obiectele de interfață, deoarece acest lucru sporește gradul de reutilizare.

În cele mai multe aplicații Cocoa, notificările de schimbări de stare din modele sunt comunicate obiectelor de interfață prin intermediul obiectelor de control. *Figura 1* arată această configurație diferită, care se dovedește a fi mult mai curată, în ciudat implicării în plus a două paradigme noi. Obiectul de control încorporează atât rolul mediatorului (*Mediator pattern*), cât și paradigma de *Strategy*. El mediază așadar fluxul de date dintre modele și interfață în ambele direcții. Schimbările stării modelului sunt comunicate interfeței prin intermediul controllere-lor aplicației. În plus, obiectele de interfață încorporează paradigma *Command* prin intermediul implementării lor ca mecanism *target-action*. Acest macanism, care permite obiectelor de interfață să comunice cu *input* de la utilizator și alegeri ale acestuia, poate fi implementat atât în paradigma de coordonare cât și în cea de mediere. Totuși, arhitectura mecanismului diferă în funcție de controller. Pentru controllere-le care coordonează, obiectul de vizualizare este conectat la *target*-ul său (controller) în *Interface Builder* și îi este specificată o acțiune (*action selector*) care trebuie să se conformeze unei anumite semnături. Controllere-le coordonatoare, deoarece sunt delegați, pot de asemenea să fie responderi în lanț. Mecanismul de legare folosit de controllere-le de mediere, conectează de asemenea obiectele de interfață cu *target*-urile și permite semnăturilor de acționare un număr variabil de parametri, de tipuri arbitrare. În schimb, controller-ele de mediere nu se află pe lanțul de răspuns.

Există motive atât practice, cât și teoretice pentru varianta revizuită a paradigmei ilustrată în *Figura 2*, în special când vine vorba de paradigma de mediere. Controller-ele de mediere sunt derivate din clase concrete ale *NSController* și aceste clase, pe lângă implementarea paradigmei de mediere, oferă și multe funcționalități de care ar trebui să profite aplicațiile (cum ar fi gestionarea selecțiilor și valorilor implicite).

În aplicațiile Cocoa MVC, bine proiectate, obiectele de control, coordonatoare, adesea dețin controller-ele de mediere, care sunt arhivate în fișiere *nib*. În figura de mai jos se poate vedea relația dintre aceste două tipuri de obiecte de control:

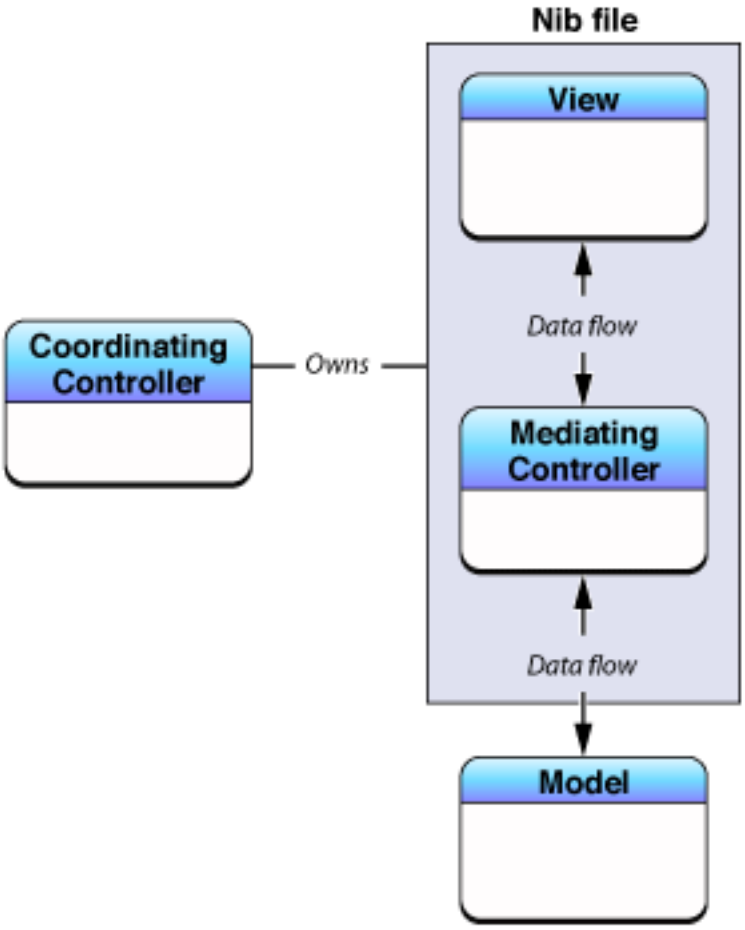

Figura 3

Având o idee generală a arhitecturii MVC, utilizată în Cocoa, putem așadar forma niște concluzii, care se aplică proiectării oricărei aplicații de iOS:

- Chiar dacă este posibilă folosirea unei instanțe a unei subclase personalizate a *NSObject* ca și controller de mediere, nu există niciun motiv pentru a implementa toată munca necesară pentru a face unul. Se folosește în schimb un obiect al clasei deja pregătite, *NSController*, proiectată pentru tehnologia de legare a Cocoa. Printre subclasele deja existente enumerăm *NSObjectController*, *NSArrayController*, *NSUserDefaultsController* și *NSTreeController*.
- Chiar dacă se pot combina rolurile MVC într-un singur obiect, cea mai bună strategie generală este ca rolurile să fie separate. Această separare sporește gradul de reutilizare al obiectelor și extensibilitatea programelor în care sunt folosite. Dacă într-adevăr se dorește interclasarea rolurilor MVC într-o singură clasă, se recomandă alegerea unui rol predominant pentru acea clasă și apoi (din motive de întreținere) folosirea categoriilor în același fișier de implementare pentru a extinde clasa în a suporta roluri noi.

- Un obiectiv al unei aplicații MVC bine structurate ar trebui sa fie folosirea cât mai multor obiecte reutilizabile posibil. În particular, obiectele de interfață și modelele ar trebui să fie extrem de ușor de reutilizat, deoarece comportamentul specific aplicației în cauză este foarte des concentrat cât mai mult în obiectele de control.
- Chiar dacă este posibil ca obiectele de interfață să observe direct modelele pentru a detecta schimbări de stare, adesea nu este cea mai bună soluție. Un obiect de interfață ar trebui întotdeauna să treacă printr-un controller de mediere pentru a afla schimbările modelelor. Există două motive principale pentru acest lucru:
    - Dacă se folosește mecanismul de legare pentru ca obiectele de interfață să observe direct proprietățile modelelor, se pierd toate avantajele pe care *NSController* și subclasele acesteia le oferă aplicației: selecția și gestionarea valorilor implicite (*placeholder*), precum și abilitatea de a executa sau anula modificări.
    - Dacă nu se folosește tehnica de legare, este necesară subclasarea unei clase de interfață deja existente pentru a adăuga abilitatea de a observa notificări de modificare, generate de un model.

- Trebuie încercat să se limiteze cât mai mult dependența dintre clasele aplicației. Cu cât dependența unei clase este mai sporită, cu atât ea este mai puțin reutilizabilă. Recomandările pot varia, în funcție de rolurile MVC ale celor două clase dependente:
    - O clasă de interfață nu ar trebui să depindă de o clasă model (deși câteodată acest lucru nu poate fi evitat, cu anumite *view*-uri foarte personalizate)
    - O clasă de interfață nu ar trebui să depindă de controller-ul de mediere
    - O clasă model nu ar trebui să depindă de nimic altceva, decât de alte clase model.
    - Un controller de mediere nu ar trebui să depindă de un model (deși, iarăși, uneori acest lucru nu poate fi evitat)
    - Un controller de mediere nu ar trebui să depindă de clasele de interfață sau de un controller de coordonare.
    - Un controller de coordonare depinde de clasele tuturor rolurilor MVC.

- Dacă librăria Cocoa oferă o arhitectură care rezolvă o problemă de programare și această arhitectura asignează roluri MVC obiectelor de tipuri specifice, se recomandă folosirea acelei arhitecturi. Proiectul va fi mult mai ușor de structurat. Arhitectura pentru documente, de exemplu, include un proiect *Xcode* care configurează obiectul *NSDocument* (*nib mode controll*er).

2.4 Swift

2.4.1 Descriere generală

Swift este un nou limbaj de programare pentru aplicațiile ce rulează pe iOS și OS X, care este contruit pe cele mai bune funcționalități ale C și Objective-C, fără constrângerile de compatibilitate impuse de C. Swift adoptă paradigme de programare sigură și adaugă funcționalități moderne, pentru a face programarea mai ușoară și mai flexibilă.

Apple lucrează de mulți ani la acest limbaj. Fundația limbajului constă în avansarea compilatorului, debugger-ului și infrastructurii librăriei deja existente. Gestionarea memoriei a fost simplificată cu ARC, despre care s-a discutat în paginile precedente. Limbajul este construit pe solida bază alcătuită din Foundation și Cocoa, care a fost modernizată și standardizată.

Acest limbaj apare ca o consecință a progresului realizat în materie de Objective-C, care a ajuns să suporte blocuri, literali de colecții și module. Datorită acestei temelii, a fost posibilă apariția Swift, un limbaj în care Apple planuiește să-și continue dezvoltarea de software.

Limbajul Swift este foarte familiar celor care sunt obișnuiți cu Objective-C. Adoptă lizibilitatea denumirii parametrilor din Objective-C și puterea generată de modelele obiect dinamice. De asemenea, furnizează acces simplificat către librăriile Cocoa deja existente, iar marele avantaj pentru cunoscătorii de Objective-C este interoperabilitatea dintre cele două limbaje, în sensul că se pot scrie aplicații în ambele limbaje, simultan (din Swift se poare apela cod de Objective-C și viceversa).

Deși baza celor două limbaje este comună, Swift introduce multe funcționalități noi și unifică porțiunile procedurale și cele orietante-obiect.

Apple susține că Swift este primul limbaj de calitate de înaltă clasă, pentru sisteme industriale, care este la fel de expresiv ca și limbajele de *scripting*. Acest lucru este într-adevăr susținut de existența unui REPL pentru Swift (*Read-Eval-Print-Loop*), similar cu cel existent pentru Scala, de exemplu. REPL permite programatorilor să experimenteze codul Swift și să vadă rezultatele imediat, fără a fi nevoie să construiască o întreagă aplicație.

În continuare, prezentăm o comparație între Swift și Objective-C (incluzând sintaxa, bineînțeles), pentru a deveni familiari rapid cu noul limbaj introdus în acest capitol.

Similarități între C și Swift:

- Majoritatea operatorilor de C există și în Swift, dar în Swift sunt și unii noi

- Acoladele sunt folosite pentru a grupa instrucțiuni în închideri (funcții)
- Variabilele sunt asignate folosind semnul =, și comparate prin ==. Este introdus un nou operator de identitate, ===, care verifică dacă două elemente referențiază același obiect.
- Instrucțiunile de control (*for, while, if, switch*) sunt asemănătoare, dar au o funcționalitate extinsă (de exemplu, *for in*, iterează orice tip de colecție, *switch* poate primi orice tip de date, etc)

Similarități între Objective-C și Swift:

- Tipurile de date numerice primitive (Int, Float, Double, UInt)
- Parantezele pătrate sunt folosite pentru vectori, atât pentru declararea acestora cât și pentru accesarea de elemente
- Metodele de clase (statice) sunt moștenite, exact ca metodele de instanțe. *self*, într-o metodă de instanță este obiectul al cărui metodă a fost apelată (echivalentul lui *this* din C++).

Diferențe dintre Objective-C și Swift:

- Instrucțiunile nu trebuie să se termine cu "*;*". Totuși, se pot folosi pentru a permite mai multe instrucțiuni pe aceeași linie.
- Fișierele header nu sunt obligatorii
- Folosește *weak typing/type inference*
- Are suport pentru șabloane (*generic programming*)
- Funcțiile sunt obiecte de primă clasă (*first-class objects*)
- Enumerațiile pot avea date asociate (tipuri de date algebrice)
- Operatorii pot fi redefiniți pentru clase (supraîncărcarea de operatori) și se pot crea noi operatori
- Șirurile de caractere suportă *Unicode*. Cele mai multe caractere *Unicode* pot fi folosite atât în identificatori, cât și în operatori.
- Nu există gestionare de excepții (deși poate fi emulată prin folosirea de închideri)
- Câteva comportamente greșite ale familiei de limbaje C, au fost schimbate:
    - *Pointer*-ii nu sunt expuși implicit. Nu este nevoie ca programatorul să urmărească și să marcheze nume pentru referențiere și dereferențiere.
    - Atribuirile nu returnează o valoare. Acest lucru previne greșeala clasică de a scrie *i = 0*, în loc de *i == 0*, prin aruncarea unei erori la compilare.
    - Nu este nevoie de folosirea instrucțiunilor *break* în *switch*. Cazurile individuale nu se propagă mai departe, decât dacă instrucțiunea *fallthrough* este folosită
    - Variabilele și constantele sunt întotdeauna inițializate și limitele vectorilor sunt întotdeauna verificate

- Overflow-ul pe întregi, care rezultă în comportament nedefinit pentru întregii cu semn în C, este detectat la rulare, în Swift. Programatorii pot alege să permită overflow-uri prin folosirea operatorilor aritmetici speciali &+, &-, &*, &/ și &%. Proprietățile *min* și *max* sunt definite în Swift pentru toate tipurile întregi și pot fi folosiți pentru a verifica securizat overflow-uri, spre deosebire de folosirea constantelor definite pentru fiecare tip în librării externe.

Pentru o familiarizare cu limbajul Swift și sintaxa acestuia, se poate arunca o privire pe 1 și Anexa 2.

2.4.2 Programare funcțională în Swift

Așa cum a fost conturat în *Introducere*, Swift suportă paradigma de programare funcțională. Pe scurt, programarea funcțională este o paradigmă care se concretizează într-un stil de a construi structuri și elemente în programe, tratând calculele ca niște evaluări de funcții matematice, evitând astfel schimbările de stare și datele mutabile (care-și schimbă valoarea). Este o paradigmă de programare declarativă, cea ce înseamnă că se programează cu expresii. În codul funcțional, valoarea de ieșirea a unei funcții depinde numai de argumentele din datele de intrare ale funcției, astfel încât apelul unei funcții *f* de două ori, cu aceleași date de intrare, va produce exact același rezultat *f(x)* de fiecare dată. Prin eliminarea efectelor colaterale (schimbări ale stării care nu depind de datele de intrare) programele sunt mult mai ușor de înțeles, comportamentul acestora devine mult mai predictibil. Acest lucru este factorul cel mai important care motivează dezvoltarea programării funcționale.

Programarea funcțională își are rădăcinile în *lambdas calculus*, un sistem formal dezvoltat în anii 1930 pentru a investiga calculabilitatea, definirea de funcții, funcțiile aplicații și recursia. Multe limbaje de programare funcțională pot fi văzute ca elaborări ale *lambda calculus*. Altă bine cunoscută paradigmă de programare declarativă, programarea logică este bazată pe relații.

Din moment ce există o stare distribuită minimală și fiecare funcție este ca o insulă într-un ocean (nu împarte date cu nimeni altcineva), codul este foarte ușor de testat.

Programarea funcțională a devenit populară mai ales că a făcut concurența și procesarea paralelă foarte ușoare. Companii precum Twitter, folosesc Scala (care este un limbaj funcțional) pentru a rula cod pe întregi clustere de mașini, evitând problemele clasice de concurență și paralelism (de care s-ar lovi dacă ar folosi un limbaj imperativ). Acesta este un motiv care argumentează folosirea unui limbaj funcțional și pentru partea de backend a aplicației exemplificate în capitolul al patrulea al acestei lucrări.

Despre programarea funcțională se pot scrie cărți întregi, dar acesta nefiind scopul acestei lucrări, vom considera un exemplu scurt, în Swift, pentru a vedea diferența clară între paradigma imperativă și cea funcțională (Swift suportându-le pe amândouă).

Să considerăm exemplul afișării numerelor pare, de la 1 la 10. Problema este trivială, iar într-o paradigmă imperativă (în Swift) codul ar arăta ca în figura următoare:

```
var evens = [Int]()
for i in 1...10 {
  if i % 2 == 0 {
    evens.append(i)
  }
}
println(evens)
```

Să observăm acum același exemplu, dar în paradigmă funcțională:

```
evens = Array(1...10).filter { $0 % 2 == 0 }
println(evens)
```

După cum se poate observa imediat, codul este mult mai scurt, mai concis, iar variabila *evens* nu mai este mutabilă.

Mai multe informații despre programarea funcțională se pot găsi în [2], [3] și [13].

3. Librăria Nucleus

3.1 Introducere

La o analiză atentă a aplicațiilor sociale existente în Apple Store, în special a celor foarte populare (Facebook, Twitter, Snapchat, Secret, Whisper, Jelly, etc) se remarcă, după cum a fost menționat și în introducere, o asemănare sporită. Capitolul de față identifică aceste asemănări consistente, le modelează în scopul obținerii unei soluții generale, aplicabile tuturor aplicațiilor în cauză, după care explică modul în care soluțiile au fost implementate în platforma Nucleus (conținând arhitectura și detalii de implementare).

Majoritatea aplicațiilor au partea de server complet separată de client. Server-ul rulează pe mașinile proprietarilor aplicației și servește drept sursă de date pentru clienții iOS. Partea de server se împarte de obicei în trei mari părți, cuprizând logica concretă a aplicației, baza de date și serviciul (API-ul) de tip RESTful. Fiecare dintre aceste trei componente pot conține la rândul lor multe alte elemente, dar din punct de vedere al arhitecturii, acele elemente vor fi independente față de celelalte componente principale, în sensul că nu va exista niciun fel de comunicare între ele.

Vom prezenta mai multe detalii despre arhitectura părții de backend a unei aplicații în subcapitolul următor. Ceea ce este important de menționat este că o aplicație iOS bine proiectată nu comunică decât cu partea de API pentru a anunța server-ul de toate acțiunile întreprinse de utilizatori și pentru a face rost de datele de care este interesată aplicația (autentificare, postări, comentarii, poze, profil, etc). De cele mai multe ori, serviciul RESTful este folosit de toți clienții (fie ei Android, iOS sau web). Acesta este principiul de bază pentru apariția așa numitului concept de *cloud-computing*.

Se conturează așadar nevoia unei metode de comunicare dintre aplicațiile iOS și serviciul web pus la dispoziție pe server. Metoda cea mai comună, folosită de aproape toate aplicațiile existente, presupune transpunerea datelor în format JSON și trimise la client/serer, în funcție de sensul de comunicare dorit.

Faptul că acest lucru este comun tuturor aplicațiilor, întărește ideea că are sens ca în aplicația noastră să avem suport pentru manipularea datelor JSON. Deși Swift conține deja acest lucru, dacă ne uităm la un strat de abstractizare mai sus, și anume cel în care transformăm datele JSON în propriile modele ale aplicației, resursele puse la dispoziție de limbaj nu mai sunt de folos.

Pe lângă suportul necesar pentru transformarea datelor neprelucrate JSON în modele ale aplicației (obiecte concrete, cu sens), comunicarea cu server-ul presupune și existența unui strat în aplicație care să se ocupe cu autentificarea, pregătirea datelor pentru a fi trimise spre server, precum și acționarea în momentul în care sunt returnate date de la server. Vom discuta

aici, concret despre *NLFNucleusAPI*, *NLFURLSession*, *kNucleusDefaultEndpointBaseURL* și *NLFNucleusAPIRequest*, clasele care se ocupă efectiv de comunicarea cu server-ul.

Acum că datele de la server pot fi obținute și integrate cu ușurință în aplicație, intervine partea de afișare și expunere a lor. Au fost identificate câteva fluxuri comune în interfețele aplicațiilor sociale din Apple Store, iar pentru aceste elemente similare s-a creat suport în librăria Nucleus.

Cel mai simplu astfel de caz este descărcarea de imagini. Întrucât iOS SDK-ul nu pune la dispoziție o metodă eficientă de descărcari de imagini, acest lucru trebuie implementat de programatori. Mai exact, în momentul în care sunt afișate multe imagini în interfață, descărcarea acestora trebuie să se facă asincron, pentru a oferi utilizatorului o experiență plăcută. În momentul în care descărcarea a fost finalizată (cu succes sau cu eroare), trebuie un mecanism prin care acest lucru să fie comunicat interfeței. Adăugăm aici și un mecanism de *caching*, prin care se evită descărcarea unei imagini de mai multe ori. Vom discuta așadar în secțiunea corespunzătoare despre *NLFDownloadManager* și *NLFDownloadableImageView*.

În SDK-ul pentru iOS, clasele de tip *View Controller* sunt cele mai importante când vine vorba de implementarea unui ecran într-o aplicație. Mai exact, fiecărui ecran din aplicație îi corespunde un *View Controller*. Pentru ecranele în care sunt listate obiecte (listă de obiecte), există un tip special de *View Controller*, numit *Table View Controller*. Acest *Table View Controller* primește o sursă de date, din care crează celule grafice pe care le afișează pe ecran. Programatorul poate specifica sursa de date și are control și asupra celulelor, prin intermediul protocoalelor implementate de *Table View Controller* (*UITableViewControllerDelegate*). Problema este că, în varianta clasică, pentru orice ecran care expune o listă de entități, este necesară suprascrierea metodelor din protocol și implementarea acestora. De exemplu, pentru o listă de utilizatori și una de postări, se crează *Table View Controllere* diferite, iar diferența dintre ele va consta în implementări relativ diferite ale metodelor din protocol. Apare așadar foarte mult cod duplicat. Nucleus pune la dispoziție o soluție pentru această problemă, permițând unui *Table View Controller* să afișeze date de orice fel, în orice fel de celulă. Acest lucru a fost realizat cu paradigma *Adapter* (*adapter pattern*), iar în secțiunea corespunzătoare vom detalia acest mecanism, prezentând *NLFNucleusTableViewController* și *NLFTableRowAdapterProtocol*.

În momentul expunerii către utilizatori a unei liste de entități (listă de postări, listă de comentarii, listă de utilizatori, etc), de obicei nu sunt afișate toate entitățile deodată, întrucât acestea sunt foarte multe, iar comunicarea server-client ar dura o perioadă foarte îndelungată de timp. Se preferă în schimb expunerea unui număr limitat de entități, după care, la cererea utilizatorului, se afișează mai multe. Cererea utilizatorului se poate realiza prin apăsarea unui buton, prin acțiunea de *pull-to-refresh* sau prin simpla acțiune de scroll la sfârșitul

listei. Întrucât la fiecare serie de primire de noi date trebuie realizată o nouă cerere la server, urmată de decodarea datelor JSON și afișarea modelelor noi la sfârșitul/începutul listei deja expuse pe ecran, se creionează necesitatea unui mecanism care să se ocupe de întregul flux. Vom prezenta în acea secțiune clasele *NLFNucleusStream* și *NLFNucleusStreamifiedTableViewController*, precum și protocoalele *NLFNucleusJSONDecoder* și *NLFNucleusStreamableOject*.

O altă funcționalitate care este extrem de dificilă de implementat este integrarea unui chat în timp real. Acest lucru implică foarte multe provocări tehnice atât pe client, cât și pe partea de server, iar realizarea unui chat eficient este extrem de complicată. Vom prezenta în acest capitol abordarea considerată de Nucleus, care pune la dispoziția programatorilor un chat ce poate fi personalizat cu ușurință și care se remarcă prin simplitate și eficiență. În plus, toată munca pe partea de server este evitată, folosindu-se un server public și gratuit, deținut de către Firebase.

Librăria de iOS nu pune la dispoziție un control de text care să conțină și o valoare implicită, programatorii fiind nevoiți să-și implementeze propria clasă pentru un astfel de câmp de text. Vom prezenta pe scurt și clasa *NLFNucleusTextView*, care va evita acest lucru.

În momentul în care un dezvoltator dorește afișarea unui clip de Youtube în propria aplicație, această sarcină se dovedește foarte complicată, deoarece Google nu pune la dispoziție decât o clasă de vizualizare, lăsând în sarcina programatorului crearea unui view controller și configurarea acestuia. Nucleus se ocupă și de această parte, ușurând munca programatorului, care va avea posibilitatea să afișeze un clip de Youtube prin simpla instanțiere a clasei *NLFYoutubePlayerViewController*.

În ecranele în care utilizatorul este nevoit să introducă date (precum postarea unui tweet, adăugarea unui post pe facebook, a unei poze, etc) logica pe care programatorul este nevoit să o implementeze se dovedește a fi destul de complicată datorită varietății de câmpuri existente (text, poze, input-uri, checkbox-uri, etc), dar din fericire suficient de generală pentru a fi extrasă și abstractizată într-o clasă generică. Clasa *NLFNucleusFormTableViewController* este o clasă ce ușurează lucrul cu formularele, în special al randării acestora pe ecran, permițând programatorului să se concentreze pe logica fluxurilor de utilizare, și nu pe afișarea formularului în sine. Tot aici, vom prezenta și câteva celule deja suportate de Nucleus pentru formulare, cum ar fi *NLFNucleusFormTableViewCell*, *NLFNucleusTextFieldFormCell* și *NLFNucleusComposeCommentFormCell*.

În final, obținerea locației utilizatorului prin intermediul clasei *CLLocationManager* pusă la dispoziție de Apple se dovedește a fi destul de complicată și necesită mult efort din partea programatorului pentru a gestiona

anumite cazuri particulare. Clasa *NLFNucleusLocationManager*, din librăria Nucleus, gestionează aceste cazuri pentru programator, lăsându-l să se concentreze doar pe manipularea coordonatelor utilizatorului și nu pe modul de obținere al acestora.

3.2 Comunicarea cu server-ul

Peste 90% din aplicațiile iOS (excluzând jocuri, bineînteles) au un server unde sunt reținute toate datele necesare despre utilizatori, platformă, etc și care gestionează întreaga logică. Aplicațiile iOS au așadar nevoie de un suport de comunicare cu orice tip de serviciu. Întrucât majoritatea serviciilor sunt de tip REST și primesc, respectiv returnează, date în format JSON, introducem în librăria Nucleus suport pentru aceste lucruri frecvent întâlnite.

În primul rând, introducem constanta *kNucleusDefaultEndpointBaseURL*, care reprezintă adresa pe web a serviciului REST corespunzător aplicației. Este datoria programatorului să configureze această constantă, întrucât ea diferă de la aplicație la aplicație.

NLFURLSession este o clasă ce moștenește clasa din iOS SDK *NSURLSession*, și a fost adăugată în scopul de a păstra platforma consistentă în denumiri și pentru a face posibilă dezvoltarea viitoare în jurul acestei sesiuni. Aceasta este clasa principală care realizează cererile către server și care ascultă rezultatele primite drept răspuns.

O clasă mai interesantă o reprezintă *NLFNucleusAPIRequest*, ce încorporează o cerere generală către un server. Conține un dicționar cu cheile și valorile tuturor parametrilor, precum și subcalea *endpoint*-ului unde se va duce cererea. Practic, în momentul în care se dorește trimiterea unei cereri spre un server, se configurează un obiect de tipul *NLFNucleusAPIRequest* și se trimite cererea prin intermediul clasei *NLFNucleusAPI*.

NLFNucleusAPI este clasa care primește cererea configurată și care inițiază sesiunea de comunicare cu server-ul. Pe lângă cererea deja configurată, aceasta mai primește ca parametru și o închidere (*closure)*, despre care s-a discutat în capitolul precedent, care reprezintă practic o metodă ce va fi apelată când sosește răspunsul de la server. Este practic o metodă de prelucrare a datelor primite ca și răspuns și este datoria celui care inițiază cererea să o configureze.

În capitolul următor va exista un exemplu despre cum funcționează întreg procesul de cerere și răspuns.

3.3 Descărcarea eficientă de imagini

În iOS SDK, descărcarea unei imagini se realizează poare realiza sincron sau asincron. Soluția sincronă este, în mod evident, ineficientă și sunt foarte rare cazurile în care aceasta este necesară. De obicei, nu este recomandată blocarea firului de execuție al interfeței grafice pentru descărcarea unei imagini (și în general, pentru orice alt tip comunicare cu servere), deoarece utilizatorul nu mai poate accesa nimic din aplicație pe parcursul comunicării cu o entitate la distanță. Într-o rețea de viteză redusă (E sau 3G, de exemplu), comunicarea poate dura o perioadă foarte lungă de timp, timp în care interfața aplicației ar îngheța și utilizatorul ar fi nevoit să aștepte, fără să fie capabil să mai facă nimic. Acest lucru implică o experiență a utilizatorului foarte săracă, ceea ce l-ar putea determina să părăsească aplicația.

Totuși, sunt unele cazuri, rare ce-i drept, în care cererile la server chiar trebuie executate sincron. De exemplu, dacă principalul element din ecranul curent este o anumită poză, și fără ea ecranul curent nu ar avea sens (un ecran de editare de poze, etc). Totuși, chiar și în aceste cazuri deosebite, se recomandă folosirea unui alt element (*placeholder*), care să țină locul pozei, până aceasta este descărcată (un indicator, o bară de progres, un *spinner*, etc). Astfel, utilizatorului i se oferă impresia de rapiditate și i se aduce la cunoștință faptul ca aplicația nu este înghețată, ci se întâmplă niște lucruri pentru care el trebuie să aștepte.

Eliminând așadar cazul sincron, prezentăm clasa *NLFDownloadManager*, care reprezintă componenta principală a librăriei Nucleus ce se ocupă de descărcarea de imagini, în mod asincron.

În general, acțiunea de descărcarea a unei imagini se execută în 3 pași:

- Având un URL al locului pe Internet unde se află imaginea dorită, se inițiază cererea asincronă la severul unde este imaginea stocată
- În momentul rezolvării cererii există două cazuri:
    - *succes*: imaginea a fost descărcată cu succes, caz în care datele neprelucrate obținute de la server pot fi transformate într-un obiect de tipul *UIImage*.
    - *eroare*: caz în care, în funcție de aplicație se pot face o serie de operații care să trateze eroarea
- Dacă descărcarea imaginii s-a efectuat cu succes și, implicit, s-a obținut obiectul de tip *UIImage* corespunzător, acesta este trimis mai departe obiectului care a inițiat descărcarea (de obicei, unui *view controller*, sau chiar unui *view*, după cum vom vedea mai departe), urmând ca acesta să efectueze operațiile dorite cu imaginea obținută (să o afișeze, de obicei)

Clasa *NLFDownloadManager* conține suport pentru toți acești pași, iar în plus, realizează și o modalitate de *caching* al imaginilor descărcate.

Studiem în continuare anatomiei clasei *NLFDownloadManager*, care subliniază ușurința de a face lucruri relativ complicate, într-un mod simplu, datorită limbajului Swift. Singura metodă public a clasei *NLFDownloadManager* are signatura următoare

public func downloadImage(url: String, useCache: Bool, inBackground: Bool, observer: AnyObject)

și după cum sugerează și numele, se ocupă în totalitate de descărcarea unei imagini. Primește următorii parametri:

- *url*: un URL al imaginii,
- *useCache*: o variabilă booleană, care comunică manager-ului dacă se va folosi *caching* sau nu pentru descărcarea imaginii curentă
- *inBackground*: o variabilă booleană, care indică tipul de cerere (sincronă sau asincronă)
- *observer*: este obiectul care deține manager-ul curent ce a inițiat cererea, și care îi observă activitatea. Observatorul este obiectul care este notificat în momentul în care imaginea a fost descărcată cu succes sau cu eroare.

Clasa *NLFDownloadManager* conține de asemenea și o metodă privată cu signatura prezentată mai jos, care are rolul de a notifica observatorul activității curente că imaginea a fost descărcată cu succes, pasându-i acestuia un obiect de tip *UIImage*, reprezentând imaginea descărcată:

func postImage(image: UIImage)

Notificarea trimisă are numele *kNLFDownloadManagerDidDownloadImage* și este propagată prin intermediul metodei *postNotificationName* a centrului implicit de notificări, pus la dispoziție de iOS SDK. Iată cum arată apelul de trimitere al imaginii către observatorii ce ascultă notificarea de descărcare cu succes:

NSNotificationCenter.defaultCenter().postNotificationName(kNLFDownloadManagerDidDownloadImage, object: self, userInfo:["image": image])

Pentru a ilustra modul de folosire al acestui manager de descărcare de imagini, librăria Nucleus pune la dispoziție un element grafic, care afișează o imagine, dându-se un URL, descărcată anterior în mod asincron. Este vorba de clasa *NLFDownloadableImageView*, care îmbogățește clasa *UIImageView*, pusă la dispoziție de iOS SDK, dar care nu se ocupă de descărcarea de imagini. În plus față de *UIImageView*, *NLFDownloadableImageView* are un URL al unei imagini și un manager de descărcare (obiect de tipul *NLFDownloadManager*).

În momentul creării unui control de tipul *NLFDownloadableImageView*, acesta se înregistrează la a asculta notificările denumite *kNLFDownloadManagerDidDownloadImage*, iar în momentul în care URL-ul unui astfel de obiect este setat, se începe procedura de descărcare asincronă a

imaginii. Când obiectul curent este notificat de către manager-ul de descărcare că imaginea a fost descărcată cu succes, acesta preia imaginea și își actualizează câmpul *image*, existent în clasa de bază *UIImageView*, ceea ce rezultă în afișarea imaginii pe ecran.

S-a pus așadar la dispoziția utilizatorilor librăriei Nucleus un element grafic care realizează o descărcare asincronă de imagini, rezultând în afișarea cu succes a imaginii pe ecran.

Procesul prezentat până acum reprezintă o soluție bună în cazurile în care setarea URL-ului unei imagini se efectuează o singură dată. Totuși, în cazul în care aceste obiecte fac parte, de exemplu, din celulele unui tabel (table view) dintr-un *Table View Controller*, tehnica prezentată până acum se dovedește a fi ineficientă. Explicația constă în faptul ca iOS SDK are propriul mecanism de reciclare a celulelor. Pe scurt, există doar un număr foarte limitat de celule (doar un ecran de celule), iar în momentul în care utilizatorul face *scroll* pe *table view*, unele celule sunt schimbate în altele (reciclate). Această schimbare se efectuează doar în interior (se setează câmpurile din nou), fără a modifica nicio adresă de memorie. Prin urmare, la fiecare acțiune de *scroll*, se setează URL-urile imaginilor pentru fiecare celulă iarăși, rezultând așadar într-o nouă inițiere de descărcare a imaginilor, lucru care evident este foarte ineficient, încetinind comparabil viteza de răspuns a aplicației și crescând vizibil memoria și utilizarea procesorului dispozitivului. Mai multe detalii despre modul în care iOS SDK reciclează celulele unui *table view*, se găsesc în [9] și [11].

Acest lucru reprezintă principalul motiv pentru care un mecanism de caching este indispensabil, și implicit, introdus în librăria Nucleus. Clasa *NLFDownloadManager* stochează într-un câmp static toate obiectele de tip *UIImage* care au fost procesate pe parcursul vieții aplicației. Având aceste obiecte deja reținute în memorie, în momentul în care se realizează o cerere de descărcare de imagini, se verifică mai întâi dacă imaginea cerută există deja în memorie. Dacă răspunsul este afirmativ, se returnează direct imaginea din memorie, fără a mai iniția legătura cu server-ul, economisind astfel timp și resurse. În momentul reciclării celulelor dintr-un *table view*, deși *view*-urile se reinițializează mereu, într-o ordine relativ aleatorie, nu mai este nevoie descărcarea efectivă a imaginilor, fiind suficientă returnarea lor direct din memorie.

O parte negativă al acestui proces este faptul că dacă o imagine este modificată pe server, aplicația de iOS nu poate fi înștiințată de acest lucru și va folosi în continuare imaginea stocată în *cache*. Abia când utilizatorul intră a doua oară în aplicație va putea vedea noua imagine. Există tehnici mai avansate de *caching* care rezolvă și această problemă, însă în cazurile cele mai frecvente, acest compromis este acceptabil, deoarece imaginile nu se modifică aproape niciodată.

S-a realizat așadar o modalitate eficientă de descărcare de imagini, care să fie compatibilă cu procesul de reciclare al celulelor unui *table view* și care să nu blocheze interfața utilizatorului. Mai mult, s-a pus la dispoziția programatorilor și un element grafic, *NLFDownloadableImageView*, care poate fi folosit cu mare ușurință și care îi ferește din a mai petrece timp înțelegând întreg procesul, putându-se concentra mai mult pe nevoile specifice aplicației în curs de dezvoltare.

3.4 Îmbogățirea clasei *UITableViewController*

Această secțiune prezintă modul în care Nucleus adaugă funcționalități tehnice clasei *UITableViewController*, clasă deja existentă în iOS SDK. Pentru mai multe detalii premergătoare despre clasa *UITableViewController*, facem referire la [10], unde se află explicații complete și la [11], unde se pot găsi informații despre protocolul *UITableViewControllerDelegate*, elementul central al funcționalității adăugate în această parte din librăria Nucleus.

3.4.1 Adapter design pattern

Pentru a înțelege mai bine modul în care lucrează clasa *NLFNucleusTableViewController*, trebuie făcută o scurtă trecere în revistă a paradigmei adaptorului.

Pe scurt, definiția din lumea reală este că un adaptor ajută două interfețe incompatibile să lucreze împreună. Paradigma practic permite unor clase incompatibile să lucreze împreună, prin convertirea interfeței unei clase într-o interfață compatibilă cu clienții.

Există două tipuri de astfel de paradigmă:

- Object adapter pattern
- Class adapter pattern

Vom prezenta doar prima dintre ele, întrucât este singura folosită în contextul *NLFNucleusTableViewController*.

Ilustrăm prin intermediul *Figurii 4* modul în care acționează acest model:

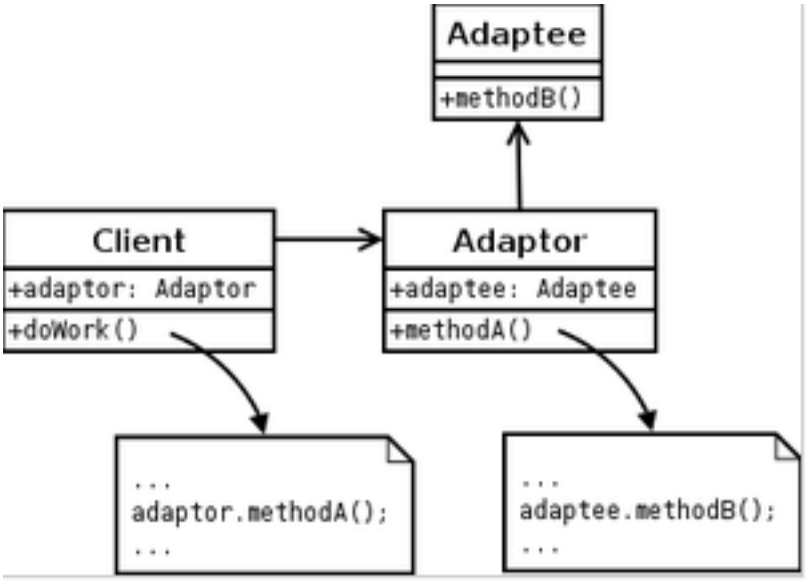

Figura 4

După cum se observă din figură, adaptorul conține o instanță a clasei pe care o împachetează. În această situație adaptorul face apeluri către instanța obiectului împachetat.

Mai multe explicații despre acest model arhitectural se pot găsi în [4].

3.4.2 Anatomia clasei *NLFNucleusTableViewController*

După cum a fost menționat și mai sus, un obiect de tipul *UITableViewController* primește o sursă de date pe care o transpune în celulele unui tabel, prin intermediul funcțiilor enumerate în protocolul *UITableViewControllerDelegate*. Deși această abordare este foarte generală, în cazul aplicațiilor sociale este foarte des nevoie de listarea unor anumite obiecte (utilizatori, poze, postări, comentarii, etc). Singurele diferențe dintre toate aceste tipuri de ecrane o reprezintă modelele (Utilizator, Comentariu, Poză, etc) și celulele în care sunt afișate (un utilizator are o afișare diferită față de un comentariu, de exemplu).

Profitând de această observație, s-a creat clasa *NLFNucleusTableViewController*, care are rolul afișării unui tabel, indiferent de modelele de date cu care este populat și indiferent de celulele pe care trebuie să le expună, făcând astfel sarcina de a afișa o listă ce conține orice fel de elemente un lucru trivial.

Premergător, introducem protocolul *NLFTableRowAdapterProtocol*, care are rolul de a descrie un obiect adaptor. Analizând cu atenție cele două metode din acest protocol (Anexa 1), observăm că orice obiect care implementează acest protocol nu face altceva decât să transforme orice obiect primit ca parametru într-un obiect de tip UITableViewCell. Cea de-a doua metoda primește ca parametru un obiect model și returnează o estimare a înălțimii celulei respective. Iată așadar tehnica adaptorului pusă în practică. Ca și exemplu, un adaptor de tipul *NLFTableRowAdapterProtocol*, ar putea primi ca parametru un obiect de tipul *User* și să returneze o celulă de tipul *UITableViewCell*, ale cărei câmpuri și valori să fie populate cu descrierea utilizatorului respectiv.

Mai departe, clasa *NLFNucleusTableViewController* moștenește direct din *UITableViewController*, primind așadar toate funționalitățile existente acolo. În plus, conține două câmpuri noi, *adaptersArray* și *backingObjects*, ambele fiind vectori.

adaptersArray este elementul central al funcționalității nou adăugate. Acest vector este populat prin intermediul metodei *use* care are signatura următoare:

public func use(adapter: NLFTableRowAdapterProtocol, classRef: AnyClass)

Practic, când se apelează pe un obiect de tipul *NLFNucleusTableViewController* metoda *use(adaptor, numeClasă)*, i se comunică controller-ului că pentru afișarea tuturor obiectelor de tipul *numeClasă* se va folosi adaptorul *adaptor*. *adaptor*, care implementează protocolul *NLFTableRowAdapterProtocol*, va transforma practic orice obiect de tipul acelei clase, într-o celulă pentru a popula *table view*-ul.

backingObjects reprezintă un vector, ce poate conține orice fel de obiecte (modele) care vor fi transformate apoi de către adaptor în celule. Practic *backingObjects* reprezintă sursa de date pentru *NLFNucleusTableViewController*.

Pentru implementarea integrală a acestui view controller, se poate verifica Anexa 1.

Marele avantaj al acestui mecanism este că, pentru orice ecran nou ce presupune listarea unor anumite modele (utilizatori, comentarii, postări, etc), este necesară doar crearea unui obiect de tip *NLFNucleusTableViewController*, specificarea adaptorului și specificarea vectorului ce reprezintă sursa de date.

Un avantaj colateral este faptul că, *NLFNucleusTableViewController* suportând mai mulți adaptori pentru diferite modele, este posibilă listarea în cadrul aceluiași tabel a unor obiecte de tipuri diferite (se pot lista în același tabel și postări, și comentarii, și utilizatori, etc), fără a fi nevoie de alte specificații din partea programatorului.

Dacă se dorește utilizarea de celule personalizate, acestea evident trebuie implementate separat, dupa cum vom vedea în capitolul următor, când vom exemplifica folosirea acestor noi componente. De asemenea, este necesară implementarea unui adaptor diferit pentru fiecare model care urmează să fie listat, dar acest lucru este inevitabil. Avantajul enorm este că, datorită faptului că aplicațiile trebuie să fie consistente din punct de vedere al interfeței, este nevoie doar de o singură implementare pe model, lucru care reduce codul într-o cantitate semnificativă.

3.5 Fluxuri de date

După cum a fost menționat în introducerea acestui capitol, se observă existența unui șablon de listare de informații în aproape toate aplicațiile iOS, și anume, împărțirea primirii de informații în serii de dimensiuni mai mici.

Ca și exemplu, să considerăm listarea de către tweet-uri pe ecranul principal din aplicația Twitter. Dacă toate acele tweet-uri ar fi primite printr-o singură cerere la server, afișarea lor în aplicație (și chiar descărcarea rezultatului) ar fi imposibilă, deoarece pentru utilizatorii care urmăresc multe persoane, cantitatea de tweet-uri ar putea ajunge să ocupe foarte mult spațiu, depășind capacitatea rețelei și chiar a hard disk-ului telefonului. În schimb, Twitter afișează mai întâi un număr mic de tweet-uri (aproximativ 100), urmând să afișeze mai multe doar dacă utilizatorul o cere prin diferite mijloace (*pull to refresh* sau *scroll* la sfârșitul listei). Practic, întreaga listă este fragmentată în dimensiuni mai mici și este trimisă către aplicație în mai multe serii, reducând astfel traficul pe rețea (care este foarte important când utilizatorii folosesc datele mobile) și timpul de returnare al datelor, incluzând aici și timpul de afișare al obiectelor.

Strategii similare sunt folosite și de alte aplicații, precum Facebook, Snapchat, etc.

Apare așadar nevoia de avea un obiect care să manipuleze acest întreg mecanism. Datele trebuie să fie consistente, trebuie evitate duplicatele și totul trebuie să se întâmple fluent.

Au fost deja prezentate clasele prin care se rezolvă cererile la server și modul în care datele primite ca și răspuns sunt prelucrate de către predicatul trimis ca parametru clasei *NLFNucleusAPI*.

3.5.1 Clasa *NLFNucleusStream*

Introducem în continuare clasa *NLFNucleusStream*, care are rolul principal să obțină și să mențină consistente datele preluate prin serii de cereri. Pentru ca un astfel de obiect să fie complet, este nevoie de mai mulți parametri, descriși în continuare:

- **configurația cererii** - un obiect de tip *NLFNucleusAPIRequest*, care să încorporeze *endpoint*-ul unde se va duce cererea, împreună cu parametrii necesari (date adiționale, precum sesiunea de autentificare, id-ul utilizatorului, id-ul unei postări, etc)
- **un obiect care decodează JSON** - În momentul în care serverul returnează o listă de rezultate, acest rezultate JSON trebuiesc convertite în obiecte model din aplicație. De exemplu, dacă serverul returnează pentru un astfel de flux, o listă de utilizatori, obiectul care decodează JSON va transforma datele din format JSON în obiecte ale clasei Utilizator.
- ***objects*** - este un vector care reține absolut toate obiectele primite de la server, în ordine. De exemplu, dacă se fac 3 cereri la acelasi endpoint, și se răspunde cu serii de 50 de utilizatori, vectorul *objects* va conține 150 de utilizatori.

NLFNucleusStream are două funcții principale:

- *loadTop(n)* - execută o cerere la server, comunicându-i acestuia că se dorește returnarea celor mai noi *n* elemente, din fluxul deja existent.
- *loadMore(n)* - execută o cerere la server, comunicându-i că se dorește returnarea următoarelor *n* elemente, din fluxul deja existent

Modul în care funcționează concret aceste două metode este un detaliu de implementare și se poate vedea în Anexa 2.

3.5.2 Clasa *NLFNucleusStreamifiedTableViewController*

După cum sugerează și numele, acest *view controller* encapsulează un obiect de tipul *NLFNucleusStream*, având rolul principal de a susține interacțiunea dintre activitățile utilizatorului și obiectul fluxului de date.

NLFNucleusStreamifiedTableViewController moștenește toate funcționalitățile clasei *NLFNucleusTableViewController*, prezentată anterior, la care se adaugă și funționalitățile obiectului de flux de date, de tipul *NLFNucleusStream.*

În principiu, *NLFNucleusStreamifiedTableViewController* este o clasă ce gestionează un ecran pe care sunt afișate date într-un tabel, ale cărui date sunt culese treptat de la server, în mai multe serii.

Pentru crearea unui astfel de obiect este suficientă menționarea descrierii obiectului de flux care va fi folosit, ce constă în specificarea unui obiect al clasei *NLFNucleusAPIRequest*, reprezentând descrierea *endpoint*-ului și a parametrilor trimiși server-ului, și al unui obiect *NLFNucleusJSONDecoder*, care are rolul să decodeze lista de modele, în format JSON, primită de la server. Întregul set de acțiuni suportate sunt deja implementate, salvând astfel timp programatorului.

Principalele două funcționalități, la nivel de interfață cu utilizatorul, sunt:

- *pull to refresh*. Funcționalitate descrisă anterior, prin care dacă utilizatorul trage suficient de mult de *view controller*, în jos, se face o cerere la server și se actualizează tabelul curent cu datele cele mai recente, care nu existau deja în tabel (de exemplu, tweet-urile noi care s-au postat de la ultima cerere)
- *load more*. Funcționalitate care afișează mai multe obiecte la sfârșitul tabelului, dacă utilizatorul a ajuns la sfârșitul listei de obiecte.

Calculele de coordonate, actualizarea stream-ului de date, precum și redesenarea tabelului în momentul primirii răspunsurilor de la server sunt toate implementate deja în *NLFNucleusStreamifiedTableViewController.*

Foarte important de menționat aici este faptul că această clasă profită în totalitate de funcționalitățile clasei de bază, *NLFNucleusTableViewController*, existând așadar posibilitatea specificării de diverși adaptori ce transformă modelele în celule.

3.6 Serviciul de mesagerie

Nucleus conține o implementare pentru mesagerie, integrată cu server-ul de date pus la dispoziție de *Firebase*, despre care se pot citi mai multe lucruri în [6].

Provocarea principală când vine vorba de implementarea unui chat este actualizarea în timp real. Strategii precum interogarea bazei de date la anumite perioade de timp (din cinci în cinci secunde, de exemplu), nu sunt eficiente, consumând mult trafic și nerespectând în totalitate conceptul de real-time.

De aceea, s-a ales o soluție eficientă, a cărei implementare s-a dovedit a fi destul de simplă, deoarece s-a utilizat suportul pus la dispoziție de *Firebase*.

Modul în care funcționează comunicarea dintre obiectele din clasa *ChatRoomViewController* și server-ul *Firebase*, din platforma Nucleus este descris în continuare:

- În momentul creării unui ecran de tipul *ChatRoomViewController*, se crează un *socket* între aplicația de iOS, aflată pe telefonul utilizatorului și server-ul *Firebase*, configurat automat în momentul creării contului de

Firebase (programatorul practic crează acel cont o singură dată, în momentul integrării chat-ului în aplicație)

- *Socket-ul* creat permite existența fluxurilor de date de citire și scriere. În momentul în care un utilizator trimite un mesaj din interfața telefonului, acest mesaj este scris în fluxul de scriere al *socket*-ului mobilului, care reprezintă fluxul de citire al server-ului. Server-ul citește așadar mesajul trimis de acel utilizator și îl memorează în propria bază de date, având însă grijă să scrie mesajul și în fluxul de scriere al destinatarului (dacă acesta există). Dacă destinatarul este intrat pe chat, fluxul acestuia de citire primește mesajul, care ulterior este afișat pe ecran.

Avantajul comunicării prin intermediul *socket*-urilor este că aceasta nu consumă foarte mult trafic, iar conexiunea este deschisă non-stop, discuția având într-adevăr loc în timp real.

Un programator care dorește integrarea unei astfel de funcționalități în interfață, nu trebuie decât să instanțieze clasa *ChatRoomViewController* și să îi specifice expeditorul și destinatarul, două modele de tipul *User*, care să aibă câmpul id (doar id-urile și mesajele sunt reținute în baza de date *Firebase*). Orice discuție care a avut loc precedent între cei doi utilizatori este încărcată automat pe ecran în momentul deschiderii acestuia. Se observă așadar ușurința și eficiența pe care o aduce în plus această funcționalitate existentă în Nucleus.

Mai multe detalii despre modul de folosire al acestei clase se află în capitolul următor, când exemplificăm un astfel de ecran în aplicația Abroad.

3.7 Suport pentru afișarea și manipularea formularelor

Într-o aplicație iOS, de foarte multe ori este necesară existența formularelor. Formularele sunt acele ecrane sau porțiuni de ecrane, unde utilizatorul are posibilitatea să introducă date, care ulterior sunt trimise la server. Aceste date pot căpăta diferite forme: text, checkbox-uri, blocuri de selecție, controale de comutare, etc.

De asemenea, faptul că toate aceste elemente grafice pot avea structuri diferite și design diferit, crearea unui ecran generic, care să suporte orice fel de formular se dovedește a nu fi trivială.

NLFNucleusFormTableViewController rezolvă această problemă, punând la dispoziție o interfață generică, prin care se pot crea și manipula orice fel de formulare.

Moștenind direct din clasa *UITableViewController*, clasa *NLFNucleusFormTableViewController* consideră orice formular drept un tabel (cu

celule). La crearea unui astfel de obiect, programatorul specifică celulele dorite în formular (orice fel de celule), precum și ordinea acestora, lăsând mai departe afișarea și manipularea câmpurilor pe seama librăriei Nucleus. Toate celulele trimise ca și parametri acestui *view controller* trebuie să moștenească direct sau indirect din clasa *NLFNucleusFormTableViewCell*, care are o singură funcție, ce returnează înălțimea celulei curente, recomandându-se suprascrierea acestei metode de către orice clasă derivată.

Tot în Nucleus se află și două tipuri de celule costumizate, care au fost necesare în realizarea aplicației din capitolul următor. Acestea sunt:

- *NLFNucleusTextFieldFormCell*, ce reprezintă o celulă conținând un element de introducere de text de tipul *NLFNucleusTextView*, despre care se va discuta în secțiunea următoare
- *NLFNucleusComposeCommentFormCell* - ce îmbogățește clasa *NLFNucleusTextFieldFormCell*, cu un element de tipul *UIButton*, pe care utilizatorul poate să-l apese. Vom vedea în capitolul următor cum se realizează manipularea acțiunilor acestui tip buton.

3.8 Obținerea simplificată a locației

SDK-ul de iOS pune la dispoziția dezvoltatorilor o clasă numită *CLLocationManager*, care în interacțiune cu protocolul *CLLocationManagerDelegate*, oferă posibilitatea obținerii locației curente a oricărui utilizator (după ce acesta a oferit permisiunea de colectare a locației, aplicației).

Aceste două entități menționate nu oferă totuși o posibilitate rapidă de aflare a coordonatelor, utilizarea lor dovedindu-se complexă și consumatoare de timp. Ele sunt mai degrabă potrivite pentru aplicații de orientare (ca Google Maps de exemplu) sau de GPS.

Pentru aplicațiile sociale, unde locația este necesară de foarte puține ori (numai la înregistrare, de exemplu), s-a adăugat librăriei Nucleus clasa *NLFNucleusLocationManager*, care simplifică procesul de obținere a locației, deși are la bază tot cele două entități aflate în SDK-ul pentru iOS.

Modul în care *NLFNucleusLocationManager* funcționează se bazează pe paradigma observatorului. Un obiect (*view controller*, de obicei) se înregistrează să observe manager-ul de locație, care îl înștiințează în momentul în care locația a fost obținută. În momentul în care locația a fost obținută, managerul de locație se oprește și se auto-distruge, deoarece nu mai este nevoie de acesta, aplicația cunoscând deja locația utilizatorului în cauză.

3.9 Elemente grafice

3.9.1 Clasa *NLFNucleusTextView*

NLFNucleusTextView este o clasă implementată în scopul afișării unui câmp de text, care să conțină o valoarea implicită (*placeholder*). Acest tip de control nu există în librăria de iOS și este foarte des întâlnit în aplicațiile din Apple Store.

Pentru a nu mai obliga programatorii să implementeze de fiecare dată acest control, utilizatorii librăriei Nucleus pot folosi direct clasa *NLFNucleusTextView*, care are întregul comportament deja implementat.

Clasa *NLFNucleusTextView* are ca și clasă de bază clasa *UITextView*, căreia i se mai adaugă și un obiect de tipul *UILabel*, care are rolul de a afișa *placeholder*-ul specificat. De asemenea, la acțiunea utilizatorului de a introduce date în căsuța de text sau de a șterge caractere, noul control implementat gestionează eticheta implicită, afișând-o și ștergând-o, atunci când este cazul.

3.9.2 Clasa *NLFYoutubePlayerViewController*

Youtube fiind foarte popuar, există multe cazuri în care se dorește încorporarea unui film de pe Youtube în aplicație. Acest lucru nu este trivial deloc, necesitând mult timp și multă cantitate de cod pentru o versiune funcționabilă a unui astfel de control.

De aceea, platforma Nucleus conține clasa *NLFYoutubePlayerViewController*, care face foarte ușoară integrarea în orice aplicație a unui video de pe Youtube.

Acest controller folosește API-ul de iOS pus la dispoziție de către Google, despre care se pot afla mai multe lucruri din [7]. Mai exact, noul controller adăugat implementează metodele protocolului *YTPlayerViewDelegate*, oferindu-i posibilitatea programatorului să afișeze un videoclip de pe Youtube doar prin specificarea ID-ului clipului.

4. *Abroad* - aplicație socială bazată pe librăria Nucleus

4.1 Descriere generală

Abroad este o aplicație socială în care utilizatorii au posibilitatea să comunice cu conaționalii din jurul lor. După cum a fost menționat și anterior, această aplicație se adresează persoanelor care locuiesc sau călătoresc în străinătate.

În momentul actual nu există o aplicație prin care oamenii de aceeași naționalitate să comunice în mod eficient, recurcându-se la metode mai complicate, precum crearea de grupuri pe Facebook (*Români în Londra, Francezi în București*, etc). Activitatea desfășurată pe aceste tipuri de grupuri încurajează existența unei aplicații specializate pe comunicarea dintre conaționali. De exemplu, pe unul dintre grupurile *Români în Londra*, se postează în medie cam 50 de statusuri pe zi, numărul total de like-uri și comentarii depășind media de 3000 pe zi.

Temele dezbătute și publicate pe acest tip de grupuri sunt diverse, oscilând de la anunțuri de căutare de slujbe și până la organizarea de întâlniri pe teme diverse (politică, fotbal, birocrație, etc).

Au fost integrate în aplicația Abroad niște funcționalități de bază, care să permită utilizatorilor comunicarea eficientă între ei. Lucrurile cele mai importante sunt enumerate în continuare.

În primul rând, utilizatorii au posibilitatea să se logheze cu Facebook, evitând așadar procesul greoi și încet de înregistrare clasică, în care utilizatorii erau forțați să introducă datele în căsuțe de text. Facebook are un număr de utilizatori enorm de mare, prin urmare este acceptabilă presupunerea că orice utilizator care face parte din publicul țintă pentru Abroad are și un cont de Facebook în avans. Marele avantaj al utilizării întregistrării cu Facebook este faptul că aceasta pune la dispoziția aplicației și naționalitatea utilizatorului în cauză (în funcție de profilul acestuia), această informație fiind cea mai valoroasă pentru buna desfășurare a activităților pe aplicație. Tot în momentul înregistrării se realizează și colectarea coordonatelor utilizatorului, care ulterior sunt trimise la server. Acest lucru ajută la poziționarea geografică a utilizatorilor, restrângându-se astfel posturile vizibile doar la o anumită arie geografică.

După autentificare, aplicația conține trei ecrane principale, accesibile prin cele trei tab-uri existente în meniul de jos: *Home*, *Messages* și *Me*.

În ecranul de *Home* utilizatorul poate vizualiza fluxul de statusuri postate de către conaționalii din jurul acestuia. Acestea sunt ordonate cronologic (cel mai

recent fiind afișat primul) și oferă și posibilitatea de actualizare, prin intermediul funcționalităților din Nucleus.

Ecranul *Me* este identic ca și aspect, proiectare și funcționalități, cu ecranul *Home*, diferența constând în faptul că în acest ecran sunt listate doar statusurile postate de utilizatorul curent.

Când un utilizator deschide un astfel de status (făcând *tap* pe celulă), aplicația deschide un ecran de detalii ale acelui status, afișând detaliile statusului, ale autorului acestuia, precum și lista de comentarii adresate acelui status. În plus, se pune la dispoziția utilizatorilor și posibilitatea de a adăuga un comentariu nou acestui status, dând naștere astfel conversațiilor publice.

În cel de-al doilea tab, *Messages*, sunt listate toate persoanele cu care utilizatorul curent a avut cel puțin o conversație în trecut. În momentul deschiderii unei astfel de conversații, toate mesajele schimbate între cei doi sunt afișate pe ecran, încorporând pe deasupra și funcționalitatea de mesagerie, care a fost prezentată în platforma Nucleus.

O altă funcționalitate importantă este postarea unui nou status, care va fi propagat apoi tuturor conaționalilor din jurul utilizatorului. Acest lucru se face prin ecranul deschis prin apăsarea pe butonul de *Compose* din dreapta sus. Acest ecran conține un formular (după cum a fost definit în Nucleus), ce pune la dispoziție un câmp de text, cu text implicit (placeholder), text pe care ulterior utilizatorul îl trimite la server, prin apăsarea butonului de postare.

Modul în care au fost implementate toate aceste ecrane este descris în secțiunile următoare. Ceea ce este important de menționat este asemănarea aplicației Abroad cu aplicațiile sociale din Apple Store. Să considerăm, de exemplu, aplicația Facebook. Utilizatorii postează statusuri, care sunt apoi văzute de către prietenii lor și la care prietenii lor pot comenta. Pe pagina de profil un utilizator își poate vedea propriile postări și de asemenea are posibilitatea de mesagerie, unde poate conversa în mod privat cu orice prieten. Toate aceste lucruri sunt suportate și de aplicația Abroad, singura diferență făcându-se în modul în care statusurile sunt vizibile. În timp ce la Facebook aceste postări sunt vizibile prietenilor, Abroad propaghează postările tuturor conaționalilor din jurul celui care a postat.

O asemănare la fel de izbitoare se observă și în comparația aplicației Abroad cu Twitter. Posturile devin tweet-uri, iar conaționalii devin followeri.

În subcapitolele următoare, vom descrie pe rând arhitectura aplicației de iOS și modul în care implementarea acesteia s-a realizat cu ajutorul librăriei Nucleus, urmând ca spre final să fie prezentată pe scurt partea de server ce susține logica și datele aplicației.

4.2 Aplicația client pentru iOS

Structura interfeței cu utilizatorul a aplicației Abroad a fost descrisă sumar în secțiunea precedentă. În această parte vom descrie arhitectura aplicației, urmând ca spre final să explicăm în detaliu modul în care ecranele au fost implementate, folosind funcționalitățile librăriei Nucleus.

Se pot distinge următoarele părți majore ale arhitecturii clientului de iOS:

- Comunicarea cu server-ul
- Autentificarea cu Facebook
- Modelele aplicației
- Elementele de interfață
- Structura de View Controllere

Vom descrie în continuare, fiecare parte a aplicației Abroad.

4.2.1 Autentificarea cu Facebook

Această componentă a aplicației *Abroad*, a fost implementată cu ajutorul librăriei pusă la dispoziție de Facebook, denumită FBSDKLoginKit și care oferă suport de adăugare de buton de autentificare, de gestionare a permisiunilor, precum și de colectare a informațiilor utilizatorilor de Facebook.

Din moment ce această librărie deja există și este accesibilă tuturor, nu se va insista foarte mult pe implementarea efectivă, deoarece există foarte multă documentație pe Internet. Mai multe detalii despre procesul de autentificare cu Facebook se pot găsi în [5].

4.2.2 Comunicarea cu server-ul

În componenta de comunicare cu server-ul există 3 elemente majore:

- Codificarea datelor
- Decodificarea datelor
- Comunicarea efectivă

Pentru codificare și decodificare, s-a folosit ca și format comun cunoscut atât de server, cât și de aplicația iOS, formatul JSON. Codificarea datelor constă practic în pregătirea formatului parametrilor ce urmează a fi trimiși către server (userID, statusID, input-uri de text, etc). Decodificarea constă în transformarea datelor din format JSON primite de la server, în modele ale aplicației.

Comunicarea efectivă cu server-ul se face prin intermediul clasei *AbroadApi* (Anexa 3), care folosește abilitatea platformei Nucleus de a crea conexiuni cu server-ul. Se poate observa utilizarea clasei *NLFNucleusAPIRequest* pentru a

crea cereri spre server. De asemenea, clasa *AbroadApi* conține toate interacțiunile aplicației cu server-ul. Se poate observa așadar crearea de fluxuri de date pentru toate cele cinci ecrane introduse mai sus, precum și înregistrarea utilizatorilor, crearea de noi statusuri și postarea de noi comentarii.

Partea de decodificare este încorporată în modele. Toate modelele aplicației, care se regăsesc ca și logică și depozitare și pe server, conțin un constructor special, care primește ca și parametru un dicționar de date de tip JSON, ocupându-se cu transformarea acestor date în obiectul model în sine. Mai multe detalii de implementare se pot găsi în Anexa 3, unde se află implementarea modelului *AbroadUser*.

Se observă că această parte de comunicare cu server-ul este foarte concisă, datorită suportului librăriei *Nucleus*, care se ocupă de lucrurile generale necesare în cazul unei astfel de aplicații. Lucrurile implementate în *Abroad* sunt doar acele lucruri specifice aplicației (modelele).

4.2.3 Modelele aplicației

Datorită modului în care aplicația *Abroad* interacționează cu utilizatorii, s-a conturat necesitatea existenței următoarelor modele, descrise în continuare:

- *AbroadUser* - este o clasă ce conține informații despre un utilizator, precum nume, prenume, poza de profil, coordonate geografice, etc. Această clasă este instanțiată de fiecare dată când se dorește transpunerea în cod a conceptului de utilizator (profile, utilizator curent, autori de postări, etc)
- *AbroadPost* - este clasa ce reprezintă transpunerea în cod a conceptului de postare. Toate statusurile existente și vizibile în aplicație sunt concretizate prin această clasă. Lucrurile reținute aici sunt autorul, textul, data creării, etc.
- *AbroadComment* - reprezintă clasa ce concretizează un comentariu. Are autor, text și data creării.
- *AbroadMessage* - reprezintă un mesaj pe chat trimis între doi utilizatori (conține expeditor, destinatar, text, poză și data trimiterii)
- *AbroadDetailedStatus* - conține detaliile unui obiect de tipul *AbroadPost*

De notat este faptul că singura interacțiune dintre aceste modele și librăria Nucleus, este că fiecare din aceste modele extind protocolul *NLFNucleusStreamableOject*, din Nucleus, ceea ce permite modelelor să facă parte din ecrane ce corespund unor fluxuri de date dintre client și server. În rest, nu există nicio altă interacțiune, întrucât modelele sunt elemente specifice unei aplicații în sine, și nu unei librării.

4.2.4 Elementele de interfață

Elementele de interfață specifice aplicației sunt reprezentate în principiu de asocierea unei celule (*UITableViewCell*) fiecărui model existent (și prezentat mai sus). Datorită funcționalităților din Nucleus, acest lucru este suficient pentru o implementare rapidă a interfeței cu utilizatorul. Asocierea dintre modele și celule se realizează în două etape, după cum a fost deja prefigurat în secțiunea despre paradigma adaptorului:

- Se crează clasa adaptor, ce implementează protocolul *NLFTableRowAdapterProtocol*, din Nucleus. Dându-se practic un model, se crează o celulă grafică (*UITableViewCell*), considerând câmpurile existente în model
- Dacă celula ce se dorește a fi asociată este mai complexă decât o simplă celulă de tipul *UITableViewCell*, se poate crea și o nouă clasă de celule, care să extindă *UITableViewCell* și să realizeze afișarea complexă a modelului în cauză.

În aplicația Abroad, există implementări pentru următoarele clase de adaptori:

- *AbroadPostRowAdapter*,
- *AbroadUserRowAdapter*,
- *AbroadCommentRowAdapter*
- *AbroadDetailedStatusRowAdapter*

Rolul lor este descris de numele în sine și nu se va insista pe descrierea amănunțită a acestora. De remarcat faptul că s-au creat și clase de celule specifice pentru fiecare model în parte, deoarece s-a dorit un design mai complex.

Să considerăm acum un exemplu, în care să evidențiem faptul că etapele de mai sus sunt suficiente pentru realizarea de ecrane complexe. Ecranele *Home* și *Me*, descrise anterior, sunt bazate pe capacitatea librăriei Nucleus de a afișa în *view controlle*r elemente ce provin din fluxuri de date.

Să considerăm afișarea și sincronizarea tuturor statusurilor postate de către conaționali (tab-ul *Home*). Pentru acest lucru, se crează un obiect de tip *NLFNucleusStream*, căruia i se specifică parametrii doriți (în cazul de față, doar userID-ul). Acest obiect este atașat unui *view controller* care extinde clasa *NLFNucleusStreamifiedTableViewController*, descrisă în capitolul anterior. În acest moment, *view controller*-ul afișează statusurile returnate de server, pe fluxul de date, are suport pentru *pull-to-refresh*, primește datele în serii de câte 50 de statusuri, iar în momentul în care utilizatorul navighează la sfârșitul listei, se execută o cerere la server pentru încă o serie de statusuri. Iată așadar ușurința creării unui ecran complex, în doar câteva linii de cod.

Într-un mod similar au fost implementate și ecranele în care sunt listate conversațiile, precum și ecranele în care este afișată o listă de comentarii.

4.2.5 Structura controalelor de vizualizare

În această secțiune va fi pus în lumină rolul important al librăriei Nucleus, funcționalitățile acesteia dând posibilitatea programatorilor să implementeze *view controllere* cu logică complexă în doar câteva linii de cod. Vom observa că în aproape toate cazurile, controllerele din Nucleus sunt suficiente pentru a afișa liste și formulare conținând logică complicată.

Ecranul în care se execută autentificarea cu Facebook este gestionat de către un obiect de tipul *LandingScreenViewController*. Acesta gestionează toate acțiunile care au legătură cu autentificarea cu Facebook (permisiuni, succes, erori, etc). În momentul în care autentificarea a avut loc cu succes, și toate detaliile necesare despre utilizator au fost trimise la server, în siguranță, utilizatorului autentificat i se oferă automat posibilitatea de intrare în aplicația efectivă. Acest lucru se realizează prin instanțierea unui *view controller* de tipul *NLFNucleusNavigationController*, care la rândul său conține un *view controller* de tipul *AbroadTabBarController*, ce are rolul de a gestiona afișarea și interacțiunea cu cele trei tab-uri din meniul de jos.

Fiecare din cele trei tab-uri are câte un *view controller* asociat, controlate și ele de către principalul view controller al aplicației, *AbroadTabBarController*:

- Home: *HomeScreenViewController*
- Messages: *MessagesScreenViewController*
- Me: *ProfileScreenViewController*

Fiecare listă afișată în aplicație este fie o listă de tipul *NLFNucleusTableViewController*, fie una de tipul *NLFNucleusStreamifiedTableViewController* (în funcție de funcționalitățile dorite).

De asemenea, ecranele cu formulare, precum ecranul de adăugare de post nou sau ecranul de comentare, sunt gestionate de *view controllere* de tipul *NLFNucleusFormTableViewController*.

Ecranul de mesagerie dintre doi utilizatori este realizat pe suportul clasei *ChatRoomViewController*, din cadrul librăriei Nucleus.

4.3 Aplicația server

Partea de server, ce susține aplicația Abroad a fost implementată în Scala, datele fiind stocate într-o bază de date MySQL. Aplicația a fost generată folosind Play Framework, o platformă pentru web, ce facilitează crearea de servicii web scalabile.

4.3.1 Scala și Play Framework

Scala este un limbaj de programare pentru software general. Scala are suport deplin pentru programare funcțională și un sistem de tipuri statice foarte puternic. Acest lucru permite programelor scrise în Scala să fie foarte concise și așadar mai mici în dimensiuni decât alte limbaje de programare de scop general. Multe decizii de arhitectură ale Scala au fost inspirate din critica adusă limitărilor impuse de Java.

Codul sursă scris în Scala este compilat în bytecode Java, astfel încât codul executabil rezultat rulează pe mașina virtuală Java. Librăriile Java pot fi folosite direct în Scala și invers. Ca și Java, Scala este orientată pe obiecte și folosește sintaxa cu acolade, rămasă din C. Spre deosebire de Java, Scala are foarte multe funcționalități de programare funcțională, precum Scheme, Standard MLand și Haskell, incluzând *currying*, inferența de tip, imutabilitatea, evaluarea de tip *lazy* și *pattern matching*.

Scala are deasemenea și un sistem de tipizare avansat, care suportă tipuri de date algebrice, covarianță, contravarianță, tipuri de ordine înaltă și tipuri anonime.

Alte funcționalități Scala, care nu sunt prezente în Java includ suprascrierea de operatori, parametrii opționali, parametrii numiți și excepții neverificate.

Play Framework este o platformă de aplicații web, scrisă în Scala și Java, care facilitează dezvoltarea de servicii web. Play este practic o alternativă mult mai bună la vechile instrumente Java pentru web.

Play se concentrează pe productivitate, aplicații mobile și web moderne, precum și pe consumul de resurse minim și predictibil (procesor, memorie, fire de execuție), rezultând în aplicații de scalabilitate largă și foarte performante.

4.3.2 Arhitectura aplicației

Aplicația server are trei componente principale:

- setarea și expunerea rutelor, care sunt interogate de către aplicația client
- implementarea algoritmilor corespunzători rutelor, în *AbroadAPIController*
- interacțiunea cu baza de date, în *AbroadDatabaseController*

Setarea și expunerea rutelor se realizează cu mare ușurință prin procesul pus la dispoziție de platforma Play. Astfel, pentru a seta o astfel de rută, se adaugă configurarea acesteia în fișierul *routes*. Considerăm exemplul punctului de acces în care aplicația de iOS cere statusurile care trebuie afișate pe ecranul *Home*, pentru un anumit utilizator:

GET /newsfeed/user_id/:userID controllers.AbroadAPIController.newsFeed(userID: String, skip: Long = 0)

Acest lucru comunică aplicației că în momentul în care un client accesează punctul */newsfeed/user_id/:userID*, aplicația trebuie să returneze ca răspuns rezultatul metodei *newsFeed(userID, skip)*, a clasei *AbroadAPIController.* Iată așadar ușurința setării unui punct de acces.

AbroadAPIController este partea din aplicația server care gestionează toată logica. De exemplu, această parte are grijă să trimită seriile corecte de statusuri pentru o anumită cerere, să actualizeze în mod corect detaliile utilizatorilor comunicate de către clienți, etc. Practic aici se află toate funcțiile care au rolul de a răspunde oricăror cereri venite din partea clienților.

AbroadDatabaseController gestionează interacțiunea cu baza de date. Prin intermediul acestei clase se obțin informații din baza de date și se actualizează detalii (actualizări, inserări, etc). *AbroadAPIController* comunică în mod direct cu *AbroadDatabaseController*, pentru a asigura consistența aplicației.

Pe partea de server mai există și configurarea modelelor, asupra careia nu se va insista prea mult, fiind aproximativ aceleași modele ca și în aplicația de client. De asemenea, mai există și o parte cu vizualizări, care în cazul de față se ocupă doar cu afișarea și expunerea de date în format JSON.

5. Concluzii

Librăria Nucleus pune la dispoziția dezvoltatorilor de aplicații mobile ce rulează pe sistemul de operare iOS, un set de funcționalități, preimplementate, care au rolul de a facilita procesul de dezvoltare al unei astfel de aplicații. Pe lângă faptul că programatorii economisesc foarte mult timp, utilizarea platformei Nucleus asigură o consistență sporită a aplicațiilor sociale, în general. Workflow-urile la care sunt supuși utilizatorii sunt similare între ecrane, elementele grafice afișate sunt aceleași peste tot, iar interacțiunile cu serviciile web sunt consistente între ele.

Limbajul Swift, noul limbaj dezvoltat de Apple, este un limbaj modern, care evită toate limitările impuse de către vechile limbaje, iar faptul că suportă concepte de programare funcțională îl propulsează în categoria limbajelor concise, în care se scriu aplicații ușor de scalat. Faptul că poate coexista cu Objective-C, îi oferă și mai multă putere și popularitate, programatorii de Objective-C putând să facă trecerea foarte ușor pe noul limbaj. Deși deocamdată mai are unele probleme, limbajul Swift este cel susținut și recomandat de Apple și probabil că în câțiva ani, când va ajunge la maturitate, va înlocui pe deplin vechiul Objective-C.

Paradigmele de programare folosite de către SDK-ul Apple sunt unele dintre cele mai moderne și transpun în cod concret, concepte din lumea reală. Structuri precum paradigma observatorului, adaptorului, a mediatorului, MVC, etc fac implementarea aplicațiilor foarte ușoară, oferind codului lizibilitate și extensibilitate.

Au fost accentuate pe parcurs asemănările enorme dintre majoritatea aplicațiilor sociale (Facebook, Twitter, Snapchat, etc) existente pe Apple Store, iar prin librăria Nucleus s-a încercat crearea unui set de funcționalități comun, ușor de folosit și ușor de extins, rămânând totuși în sfera paradigmelor de programare utilizate și sugerate de Apple și profitând de toate facilitățile expuse de limbajul Swift.

Începând cu elemente de dimensiune relativ restrânsă, precum descărcarea eficientă de seturi de imagini, anumite componente de interfață grafică, de uz general, și culminând cu *view controllere* care să transforme listarea de către obiecte care provin din fluxuri de date, în serie, comunicate de către un server, într-o sarcină trivială pentru un programator, platforma Nucleus reprezintă un început foarte bun pentru viitoarele aplicații sociale, făcând posibilă crearea unui produs minim viabil pentru un startup în doar câteva zile, scutind programatorii de a trece mereu prin aceleași probleme care au fost rezolvate de foarte multe ori, de către orice altă aplicație socială deja existentă.

După cum s-a observat și în capitolul în care s-a prezentat aplicația socială Abroad, ca și exemplificare de folosire a librăriei Nucleus, noua librărie

este foarte ușor de manevrat. Clasele sunt independente, publice și ușor de extins. De asemenea, Nucleus conține multe funcționaliăți complexe, care de multe ori pot fi generatoare de bug-uri pentru programatori nexeperimentați. Folosind Nucleus, toate aceste probleme dispar, iar cei care o folosesc se pot concentra mai mult pe produsul în sine și nu pe modul în care să rezolve problemele tehnice.

În concluzie, dat fiind contextul actual, în care produsele Apple câștigă din ce în ce mai multă popularitate și în care aplicațiile sociale de pe Apple Store sunt din ce în ce mai multe și mai de succes, corelat cu proaspăta apariție pe piață a limbajului Swift, crearea librăriei Nucleus se sincronizează perfect cu aceste elemente, punând la dispoziția întregii industrii de aplicații sociale, un mijloc de viitor, prin care se eficientizează dezvoltarea obișnuită de aplicații mobile pentru iOS, cu ajutorul implementării consistente a funcționalităților clasice prezente în majoritatea aplicatiilor sociale deja existente.

Bibliografie

[1] **Amin Naserpour, Robert Cope, Thomas Erl**, *Cloud Computing Design Patterns*, Editura Prentice Hall, 2015, ISBN 978-0133858563, 592 pagini

[2] **Bill Venners, Lex Spoon, Martin Odersky**, *Programming in Scala*, Editura Artima, 2011, ISBN 978-0981531649, 859 pagini

[3] **Chris Eidhof, Florian Kugler, Wouter Swierstra**, *Functional Programming in Swift*, Editura Florian Kugler, 2014, ISBN 978-3000480058, 206 pagini

[4] **Erich Gamma, John Vlissides, Ralph Johnson, Richard Helm**, *Design patterns: elements of reusable object-oriented software*, Editura Addison Wesley, 1994, ISBN 978-0201633610, 416 pagini

[5] **Facebook**, *Facebook Login for iOS*, https://developers.facebook.com/docs/facebook-login/ios/v2.3, 10 iunie 2015

[6] **Firebase**, *Setting up Firebase in Xcode*, https://www.firebase.com/docs/ios/quickstart.html, 10 iunie 2015

[7] **Ikai Lan**, *Embed Youtube Videos in iOS Applications with the Youtube Helper Library*, https://developers.google.com/youtube/v3/guides/ios_youtube_helper, 10 iunie 2015

[8] **Julien Richard-Foy**, *Play Framework Essentials*, Editura Packt Publishing, 2014, ISBN 978-1783982400, 200 pagini

[9] **Matt Neuburg**, *iOS 8 Programming Fundamentals with Swift: Swift, Xcode and Cocoa Basics*, Editura O'Reilly Media, Sebastopol, CA, USA, 2015, ISBN 978-1491908907, 582 pagini

[10] **Matt Neuburg**, *Programming iOS 8: Dive Deep into Views, View Controllers and Frameworks*, Editura O'Reilly Media, Sebastopol, CA, USA, 2014, ISBN 978-1491908730, 1018 pagini

[11] **Neil Smyth**, **iOS 8 App Development Essentials – Second Editions: Learn to Develop iOS 8 Apps using Xcode and Swift 1.2**, Editura CreateSpace Independent Publishing Platform, 2015, ISBN 978-1511713337, 826 pagini

[12] **Patrick Mick , Stephen G. Kochan**, *Programming in Swift*, Editura Addison Wesley, Boston, USA, 2015, ISBN 978-0134037578, 550 pagini

[13] **Paul Chiusano, Runar Bjarnason**, *Functional Programming in Scala*, Editura Manning Publications, 2014, ISBN 978-1617290657, 320 pagini

[14] **Shiti Saxena**, *Mastering Play Framework for Scala*, Editura Packt Publishing, 2015, ISBN 978-1783983803, 274 pagini

[15] **Subbu Allamaraju**, *RESTful Web Services Cookbook*, Editura Yahoo Press, 2010, ISBN 978-0596801687, 316 pagini

[16] **Vandad Nahavandipoor**, *iOS 8 Swift Programming Cookbook: Solutions & Examples for iOS Apps*, Editura O'Reilly Media, Sebastopol, CA, USA, 2014, ISBN 978-1491908693, 902 pagini

Anexa 1

```
let kNLFNucleusTableViewCellDefaultHeight: CGFloat = 30.0

public protocol NLFTableRowAdapterProtocol {
    func tableView(tableView: UITableView, controller: UITableViewController, cellForRowAtIndexPath
indexPath: NSIndexPath, object: AnyObject) -> UITableViewCell
    func tableView(tableView: UITableView, controller: UITableViewController, heightForRowAtIndexPath
indexPath: NSIndexPath, object: AnyObject) -> CGFloat
}

public class NLFNucleusTableViewController: UITableViewController {
    var adaptersArray: [(adapter: NLFTableRowAdapterProtocol, classRef: AnyClass)] = []
    var backingObjects: [AnyObject]?

    public func use(adapter: NLFTableRowAdapterProtocol, classRef: AnyClass)
    {
        adaptersArray.append(adapter: adapter, classRef: classRef)
    }

    public override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath:
NSIndexPath) -> UITableViewCell {
        let objects = self.objects()
        if (indexPath.row < objects.count) {
            var object: AnyObject = objects[indexPath.row]
            for (adapter,classRef) in adaptersArray {
                if (object.isKindOfClass(classRef)) {
                    return adapter.tableView(tableView, controller: self, cellForRowAtIndexPath:
indexPath, object: object)
                }
            }
        }
        return UITableViewCell()
    }

    public override func tableView(tableView: UITableView, heightForRowAtIndexPath indexPath:
NSIndexPath) -> CGFloat {
        let objects = self.objects()
        if (indexPath.row < objects.count) {
            var object: AnyObject = objects[indexPath.row]
            for (adapter,classRef) in adaptersArray {
                if (object.isKindOfClass(classRef)) {
                    return adapter.tableView(tableView, controller: self, heightForRowAtIndexPath:
indexPath, object: object)
                }
            }
        }
        return kNLFNucleusTableViewCellDefaultHeight
    }

    public func addItem(item: AnyObject) {
        if (backingObjects == nil) {
            backingObjects = [item]
        } else {
            backingObjects?.append(item)
        }
    }

    public func objects() -> [AnyObject] {
        if (backingObjects != nil) {
            return backingObjects!
        }
        return []
    }

    public override func numberOfSectionsInTableView(tableView: UITableView) -> Int
    {
        return 1
    }

    public override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int
    {
        if (nil != self.backingObjects) {
            return self.backingObjects!.count
        }
        return 0
    }}
```

Anexa 2

```
public let kNLFNucleusStreamDidUpdate = "kNLFNucleusStreamDidUpdate"

let kDefaultBatchSize = 50

public class NLFNucleusStream: NSObject {
    var objects = Array<AnyObject>()
    var isFinished = false
    let apiRequest: NLFNucleusAPIRequest
    let jsonDecoder: NLFNucleusJSONDecoder
    var isLoading = false
    var isLoadingTop = false

    public init(apiRequest: NLFNucleusAPIRequest, jsonDecoder: NLFNucleusJSONDecoder) {
        self.apiRequest = apiRequest
        self.jsonDecoder = jsonDecoder
        super.init()
    }
    public func loadTop(batchSize: Int = kDefaultBatchSize) {
        if (self.isLoadingTop) {
            return
        }

        if (apiRequest.params == nil) {
            apiRequest.params = ["top_id":self.firstId()]
        } else {
            apiRequest.params!["top_id"] = String(self.firstId())
        }

        self.isLoading = true
        NLFNucleusAPI.request(apiRequest) {(data, response, error) in
            var resultArray = Array<AnyObject>()
            let dictJSON = NSJSONSerialization.JSONObjectWithData(data, options:
NSJSONReadingOptions.MutableContainers, error: nil)
            if let dictArray = dictJSON as? Array<NSDictionary> {
                for jsonDictionary in dictArray {
                    resultArray.append(self.jsonDecoder.decodeFromJSONDictionary(jsonDictionary))
                }
            }
            self.objects = (resultArray + self.objects)
            NSNotificationCenter.defaultCenter().postNotificationName(kNLFNucleusStreamDidUpdate, object:self)
            self.isLoadingTop = false
        }
    }
    public func loadMore(batchSize: Int = kDefaultBatchSize)
    {
        if (self.isFinished || self.isLoading) {
            return
        }

        if (apiRequest.params == nil) {
            apiRequest.params = ["skip":String(objects.count)]
        } else {
            apiRequest.params!["skip"] = String(objects.count)
        }

        self.isLoading = true
        NLFNucleusAPI.request(apiRequest) {(data, response, error) in
            var resultArray = Array<AnyObject>()
            let dictJSON = NSJSONSerialization.JSONObjectWithData(data, options:
NSJSONReadingOptions.MutableContainers, error: nil)
            if let dictArray = dictJSON as? Array<NSDictionary> {
                for jsonDictionary in dictArray {
                    resultArray.append(self.jsonDecoder.decodeFromJSONDictionary(jsonDictionary))
                }
            }
            if (resultArray.count == 0) {
                self.isFinished = true
            } else {
                self.objects += resultArray
                NSNotificationCenter.defaultCenter().postNotificationName(kNLFNucleusStreamDidUpdate, object:self)
            }
            self.isLoading = false
        }
    }

    func firstId() -> String {
        if (count(self.objects) > 0) {
            return (self.objects.first! as! NLFNucleusStreamableOject).getId()
        }
        return ""
    }
}
public protocol NLFNucleusJSONDecoder {
    func decodeFromJSONDictionary(jsonDictionary: NSDictionary) -> NLFNucleusStreamableOject
}

public protocol NLFNucleusStreamableOject : AnyObject {
    func getId() -> String
```

Anexa 3

```
//
//  AbroadAPI.swift
//  Abroad
//
//  Created by Florian Marcu on 4/11/15.
//  Copyright (c) 2015 Florian Marcu. All rights reserved.
//

import NucleusFramework

class AbroadAPI: NSObject {
    class func requestLikes(userID: String, completionHandler: ((Array<AbroadPost>) -> Void)?)
    {
        let params = ["user_id":userID]
        let request = NLFNucleusAPIRequest(params:params, path:"likes.php")
        NLFNucleusAPI.request(request) {(data, response, error) in
            if completionHandler != nil {
                var songsList = Array<AbroadPost>()
                var likesArray = NSJSONSerialization.JSONObjectWithData(data, options:
NSJSONReadingOptions.MutableContainers, error: nil) as! Array<NSDictionary>
                for jsonDictionary in likesArray {
                    songsList.append(AbroadPost(jsonDictionary: jsonDictionary))
                }
                completionHandler!(songsList)
            }
        }
    }

    class func updateUser(params: Dictionary <String, String>?, completionHandler: ((AbroadUser) -> Void)?) {
        let request = NLFNucleusAPIRequest(params: params, path:"user/update")
        NLFNucleusAPI.request(request) {(data, response, error) in
            if (completionHandler != nil) {
                var errorTmp: NSError?
                var jsonDictionary = NSJSONSerialization.JSONObjectWithData(data, options:
NSJSONReadingOptions.MutableContainers, error: &errorTmp) as? NSDictionary
                if (jsonDictionary != nil) {
                    completionHandler!(AbroadUser(jsonDictionary: jsonDictionary!))
                }
            }
        }
    }

    class func createNewsFeedStream(user: AbroadUser) -> NLFNucleusStream {
        let request = NLFNucleusAPIRequest(params:["user_id":user.userID], path:"newsfeed")
        return NLFNucleusStream(apiRequest: request, jsonDecoder: AbroadPostJSONDecoder())
    }

    class func createMessagingUsersStream(user: AbroadUser) -> NLFNucleusStream {
        let request = NLFNucleusAPIRequest(params:["user_id":user.userID], path:"users/messagingUsers")
        return NLFNucleusStream(apiRequest: request, jsonDecoder: AbroadUserJSONDecoder())
    }

    class func createProfileStream(user: AbroadUser) -> NLFNucleusStream {
        let request = NLFNucleusAPIRequest(params:["user_id":user.userID], path:"profile")
        return NLFNucleusStream(apiRequest: request, jsonDecoder: AbroadPostJSONDecoder())
    }

    class func createCommentsStream(status: AbroadPost) -> NLFNucleusStream {
        let request = NLFNucleusAPIRequest(params:["status_id":status.statusID], path:"comments")
        return NLFNucleusStream(apiRequest: request, jsonDecoder: AbroadCommentJSONDecoder())
    }

    class func addStatus(userID: String, textValue: String, completionHandler: ((AbroadPost) -> Void)?) {
        let params = ["user_id":userID, "status":textValue]
        let request = NLFNucleusAPIRequest(params:params, path:"addStatus")
        NLFNucleusAPI.request(request) {(data, response, error) in
            if completionHandler != nil {
                var abroadPost = NSJSONSerialization.JSONObjectWithData(data, options:
NSJSONReadingOptions.MutableContainers, error: nil) as! AbroadPost
                completionHandler!(abroadPost)
            }
        }
    }

    class func addComment(userID: String, statusID: String, textValue: String, completionHandler: ((AbroadPost) ->
Void)?) {
        let params = ["user_id":userID, "status_id":statusID, "comment":textValue]
        let request = NLFNucleusAPIRequest(params:params, path:"addComment")
        NLFNucleusAPI.request(request) {(data, response, error) in
            if completionHandler != nil {
                var abroadPost = NSJSONSerialization.JSONObjectWithData(data, options:
NSJSONReadingOptions.MutableContainers, error: nil) as! AbroadPost
                completionHandler!(abroadPost)
            }
        }
    }
}
```

Anexa 4

```swift
//
//  AbroadUser.swift
//  Abroad
//
//  Created by Florian Marcu on 4/11/15.
//  Copyright (c) 2015 Florian Marcu. All rights reserved.
//

import NucleusFramework

class AbroadUser: NSObject, NLFNucleusStreamableOject {
    var userID: String = ""
    var profileImageURL: String = ""
    var firstName: String = ""
    var lastName: String = ""
    var email: String = ""
    var longitude: String?
    var latitude: String?

    init(facebookID: String) {
        userID = AbroadUser.abroadID(facebookID)
    }

    init(jsonDictionary: NSDictionary) {
        if (jsonDictionary["user_id"] != nil) {
            userID = jsonDictionary["user_id"] as! String
        }
        if (jsonDictionary["profile_picture"] != nil) {
            profileImageURL = jsonDictionary["profile_picture"] as! String
        }
        if (jsonDictionary["firstname"] != nil) {
            firstName = jsonDictionary["firstname"] as! String
        }
        if (jsonDictionary["lastname"] != nil) {
            lastName = jsonDictionary["lastname"] as! String
        }
        if (jsonDictionary["latitude"] != nil) {
            latitude = jsonDictionary["latitude"] as? String
        }
        if (jsonDictionary["longitude"] != nil) {
            longitude = jsonDictionary["longitude"] as? String
        }
    }

    func fullName() -> String
    {
        return firstName + " " + lastName
    }

    func getId() -> String {
        return self.userID
    }
}
```