Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Pattogató Bence

Személyes pénzügyi asszisztens alkalmazás fejlesztése

iOS és .Net platformokon

Konzulens

BUDAPEST, 2016

Table of Contents

[1 Bevezetés 4](#_Toc467530968)

[1.1 A probléma 4](#_Toc467530969)

[1.2 Feladatkiírás és annak értelmezése 5](#_Toc467530970)

[1.3 Előretekintés a dolgozatra 7](#_Toc467530971)

[2 A feladat 8](#_Toc467530972)

[2.1 Motiváció 8](#_Toc467530973)

[2.2 Funckionális specifikáció 8](#_Toc467530974)

[3 Létező megoldások feltérképezése 9](#_Toc467530975)

[3.1 Előre tekintés 9](#_Toc467530976)

[3.2 Megoldások bemutatása és értékelése 10](#_Toc467530977)

[3.2.1 Daily Account Manager (Day book) 10](#_Toc467530978)

[3.2.2 Foreceipt 11](#_Toc467530979)

[3.2.3 Tosh finance 11](#_Toc467530980)

[3.2.4 Money Mgr. 12](#_Toc467530981)

[3.2.5 Quick Money Recorder: Expense Budget Manager 13](#_Toc467530982)

[3.2.6 Buxfer 14](#_Toc467530983)

[3.2.7 TapToTrack 14](#_Toc467530984)

[3.2.8 Spender 15](#_Toc467530985)

[4 Felhasznált technológiák 17](#_Toc467530986)

[4.1 Kliens 17](#_Toc467530987)

[4.1.1 iOS 17](#_Toc467530988)

[4.1.2 iOS történeti hátter 17](#_Toc467530989)

[4.2 Szerver: .NET 17](#_Toc467530990)

[4.3 Frontend 17](#_Toc467530991)

[4.4 REST kommunikáció 17](#_Toc467530992)

[4.5 Tesztelés XCode-ban 17](#_Toc467530993)

[4.5.1 Automatizált tesztelés 17](#_Toc467530994)

[4.5.2 Tesztelés támogatása XCode-ban 18](#_Toc467530995)

[4.5.3 Az XCTest keretrendszer 18](#_Toc467530996)

[5 A rendszer tervezése 21](#_Toc467530997)

[5.1 A rendszer felépítése 21](#_Toc467530998)

[5.2 Adatmodellek 21](#_Toc467530999)

[5.3 Szerver architektúra 21](#_Toc467531000)

[5.4 iOS kliens architektúra 21](#_Toc467531001)

[5.4.1 Az alkalmazás felépítése 21](#_Toc467531002)

[5.4.2 Programozási minták 21](#_Toc467531003)

[5.4.3 MVVM – Model View ViewModel 25](#_Toc467531004)

[5.4.4 MVP – Model View Presenter 25](#_Toc467531005)

[5.4.5 Dependency injection általánosan 28](#_Toc467531006)

[5.4.6 Szálkezelés 28](#_Toc467531007)

[5.5 REST API tervezése 28](#_Toc467531008)

[5.6 iOS – Android közös tervezése lehetőségek 28](#_Toc467531009)

[5.7 Singleton 28](#_Toc467531010)

[Irodalomjegyzék 31](#_Toc467531011)

# Bevezetés

A diploma feladatom egy személyes pénzügyi asszisztens szoftverrendszerhez tartozó iOS alkalmazás és webkliens alkalmazás prototípusának fejlesztése valamint azt ezeket kiszolgáló backend alkalmazás tervezésében való részvétel. Az alkalmazás tőlem függetlenül más platformokra is implementálásra kerül, ezért nagyon körültekintően kellett megterveznünk minden olyan részét, ami a többi klienst is érinti.

A személyes pénzügyi asszisztens alkalmazás fejlesztésének ötlete saját és egy barátom példájához vezethető vissza. A felvetődő probléma minket érintett a mindennapokban és erre kínálok egy megoldást azoknak, akik szintén szemben találják magukat ezzel.

## A probléma

A probléma: magánéletem során többször ütköztem abba, hogy a pénzügyeimet nem tudtam megfelelően számon tartani, nem tudtam mire mennyit költök. Mikor kimutatást akartam csinálni, hogy milyen költségeken lehetne faragani, több hónapra visszamenőleg, csaknem kivitelezhetetlen feladatnak tűnt. Ekkor kezdtem el a piacon keresni, hogy milyen megoldásokat kínálnak erre mások.

A témakör összefoglaló neve PFM (Personal Financial Management), ez magába foglal mindenféle személyes pénzügyi kérdésekkel foglalkozó megoldást. Két alapvető ága van.

Az első megközelítés az utólagos elemzésre épít. A feldolgozás menete a következő: egy szoftverbe betápláljuk valamilyen úton a bankszámlakivonatainkat (fejlett rendszerek esetén egy egyszerű megoldás, hogy megadjuk e-bankunk hozzáférési adatait, az belép oda és letölti a számára szükséges kimutatásokat), az pedig különböző kimutatásokat készít ezekből, mindenféle meta-adat, algoritmus és heurisztika segítségével. Mivel a készpénzt teljesen automatizáltan nem tudják számon tartani, egyik gyengesége ennek a megközelítésnek, hogy csak az elektronikusan mozgatott pénzeket lehet utólagosan bármilyen módszerrel feldolgozni. Illetve felvet sok biztonság-kritikai kérdése, hogy az adott szoftver fejlesztőjében mennyire bíznak meg a felhasználók, hajlandók-e megadni ilyen szenzitív adatokat.

A másik megközelítés a manuális módszert helyezi a középpontba, azaz a felhasználó felelőssége, hogy a tranzakciókat egyesével bevigye a rendszerbe. Ezt megoldást már szinte minden létező platformra lefejlesztették, a teljesség igénye nélkül felsorolok néhányat:

* AndroMoney[[1]](#footnote-1) - Android alkalmazás
* Microsoft Excel template[[2]](#footnote-2)
* Money Control[[3]](#footnote-3) - iPhone, iPad, OS X alkalmazás (iPhone alkalmazásokról bővebben a 3. fejezetben írok)
* Money Tracker Free[[4]](#footnote-4) - Windows alkalmazás.

Ennek a módszernek nyilvánvaló hátránya, hogy néminemű idő- és energiaráfordítás szükséges a felhasználótól, hogy minden egyes költését rögzítse a rendszerben. Ellenben, ha egy szoftver annyira kényelmesen és kézreállóan működik, hogy ez kiadásonként pár másodpercen belül elvégezhető, akkor azok a felhasználók, akik számára fontos, hogy teljesen pontos képet kapjanak a pénzügyeik mikéntjéről úgy gondolom, hogy hajlandóak ráfordítani ezt az időt.

Én az utóbbi megoldás kidolgozása mellett döntöttem. Egyik érv, hogy az első megközelítés az egy nagyon komplex rendszer lenne, a másik, hogy engem személyesen különösen érdekel a mobil szoftverfejlesztés, korábban is már foglalkoztam sokat ezzel a témával. Az alkalmazás tervezése és megvalósítása során mindig azt tartottam szem előtt, hogy a felhasználónak hogyan lesz a legkönnyebb, a leggyorsabb és a legegyszerűbb adminisztrálnia egy tranzakciót.

## Feladatkiírás és annak értelmezése

Szemelyes-penzugyi-asszisztens-alkalmazas-Feladatkiiras-1-2

A feladatkiírásból is látszik, hogy a rendszer megvalósítása az nem csak az én feladatom volt, hanem egy másik hallgatóval közösen kaptuk a fenti feladatot. A munka négy nagy részre osztható, ezek a következők:

* Szerveroldali alkalmazás
* iOS mobilalkalmazás
* Android mobilalkalmazás
* Front end webalkalmazás

A felsoroltak közül az én feladatom az iOS mobil-, és frontend webalkalmazás prototípusának megvalósítása, illetve az ezeket backend szoftver tervezése. Az iOS alkalmazáshoz tartozik egy azt kiegészítő, Apple Watch alkalmazás is, ami segíti a felhasználót, hogy még könnyebben és egyszerűbben használhassa a szolgáltatást.

## Előretekintés a dolgozatra

A dolgozat felépítése, a következő: .... ezt a végén egyszerűbb lesz megírni

# A feladat

## Motiváció

## Funckionális specifikáció

Ez is majd később jön ide, van már doksi, amit csak át kell alakítani

# Létező megoldások feltérképezése

## Előre tekintés

Feladatom volt, hogy ismerjem meg és mutassam be a témakörben a már létező megoldásokat. iOS alkalmazásokat kerestem az App Store-ban, a következő keresőszavakra:

* PFM,
* financial manager,
* budget tracker,
* budget manager,
* personal finance,
* expense tracker
* expense manager.

Mivel az alkalmazás ingyenesen lesz publikálva az Apple App Store-ban, ezért az ebbe a kategóriába tartozókkal hasonlítottam össze, hogy releváns eredményeket tudjak mutatni. Az alkalmazások mindegyikét letöltöttem, átfogóan áttekintettem, működését kipróbáltam és értékeltem az alábbi szempontok szerint. A mellettük szereplő szám azt mutatja, hogy milyen súllyal számítanak bele a végső eredménybe (1-3-as skálán) amit lentebb értékelek majd, az értékkészlet 0 (legrosszabb) – 5 (legjobb):

* Mennyire intuitív az alkalmazás (3)
  + A legszubjektívebb, egyben legfontosabb mérőszám, magamon kívül 3 különböző emberrel próbáltattam ki az alkalmazást, korosztályban 18 alatt, 18-40, 40+, néztem hogy használják, milyen gyorsan jönnek rá, hogy hogy lehet egy tranzakciót felvenni.
* Mennyire gyorsan lehet felvenni egy tranzakciót (3)
  + A pontokat a következő skála szerint adtam:
    - < 5 másodperc – 3 pont
    - 5 – 10 másodperc – 2 pont
    - > 10 másodperc – 1 pont
  + A méréshez a következő tranzakciókat használtam
    - 20.000 Ft autó szervíz
    - 1500 Ft ebéd
* Mennyire átláthatók a múltbeli tranzakciók (2)
* Milyen az első benyomás 2-3 perc használat után (2)
* Design (2)
  + Mennyire felel meg az Apple guideline-oknak
  + Mennyire megnyerő külsőre
* Kötelező-e a regisztráció? (1) – véleményem szerint, ha kötelező a regisztráció az első belépésnél, az negatív felhasználói élmény. A pontozásnál úgy értékeltem, hogy mínusz pont, ha kötelező. Ha opcionális, akkor az ötletességet vettem figyelembe, hogy utólag figyelmezteti-e a felhasználót, könnyű-e elérni, stb.

## Megoldások bemutatása és értékelése

A következő listában felsorolom azokat az alkalmazásokat, amelyeket relevánsnak találtam. Ebben a listában röviden szövegesen értékelem és néhány képernyőképet mellékelek az alkalmazásokról, majd utána egy táblázatban összesítem az értékelési szempontok szerint adott pontokat.

### Daily Account Manager (Day book)[[5]](#footnote-5)

Az alkalmazás induláskor iCloud belépést kért, ott tárolja az adatokat. Felhasználói felülete átlagos, tab bar-ral működik a navigáció a képernyők között, mondhatni jól használhatónak. Mind bevételt mind kiadást lehet felvenni, a beviteli folyamat lehetne egyszerűbb. Összességében elégedett voltam, korrekt alkalmazás.

### Foreceipt[[6]](#footnote-6)

Az alkalmazás indulásakor Google account-tal kellett bejelentkezni, nem is lehetett más opciót választani, ami nem felhasználóbarát. Az alkalmazást első kézbevétel után egy kicsit nehézkes volt használni, nem láttam át rögtön, hogy melyik funkciót hogy érem el. A kezelés szintén tab bar-ral működik, az volt a benyomásom, hogy nem annyira könnyen kezelhető, a felület platfromidegen néhány helyen és sok időbe telik egy tranzakciófelvétel.

### Tosh finance[[7]](#footnote-7)



Indulás után kötelező a bejelentkezés, amit Facebook/Google és hagyományos felhasználónév-jelszó párossal is elvégezhetünk. Egy kiadás felvétele elég egyszerű, gyorsan megvalósítható. Számomra negatívum volt, hogy a szám billentyűzet túl színes és a beépítetthez képest fejjel lefelé van, ami elsőre furcsa volt. Az alkalmazás szép, könnyen kezelhető, extra benne, hogy felvehetünk külön számlákat és keretet, hogy még mennyi költhetünk az adott számláról, továbbá lehet különböző formátumokban exportálni.

### Money Mgr.[[8]](#footnote-8)

Felhasználó kezelés nincs az alkalmazásban, iCloud fiókba lehet backup-ot készíteni vagy e-mailben elküldeni magamnak, ezt nem tartom a legkönnyebb útnak. A design iOS 5-6-ra emlékeztet, a felhasználói élmény számomra eddig itt a legalacsonyabb a nehezen megnyomható gombok, kicsi, rosszul áttekinthető UI elemek és a 3-4 évvel ezelőtti sztenderekhez igazodó design miatt. A tranzakció felvétel sebessége átlagos, lehetne egyszerűbb. Összességében a nagyon elmaradt design és kezelési nehézségek miatt nem tetszett az alkalmazás, nem használnám.

### Quick Money Recorder: Expense Budget Manager[[9]](#footnote-9)

Az alkalmazás viszonylag jó helyen szerepelt az App Store-ban legnagyobb meglepetésemre, mert megnyitva az alkalmazást egy nagyon nem felhasználóbarát, design-ban körülbelül 5 évvel ezelőtti irányelvekhez igazodó képernyő fogadott. Az ikonok, a színek, a gombok formája és elhelyezkedése sem stimmelt, az első sikeres tranzakciófelvétel 2-3 percbe telt.

Azért is mutatom itt be, mert ez egy jó példa arra, hogy milyen egy alkalmazás, ami a felhasználói igényeket egyáltalán nem elégíti ki. A felhasználó-, és adatkezelés prémium funkció, amit láttam belőle az, hogy van több opció is, amik között szerepel Dropbox, Google Drive és iCloud is.

### Buxfer[[10]](#footnote-10)



Induláskor felajánlja az alkalmazás, hogy hozzak létre felhasználói fiókot, akár közösségi profil alapján, de van egy próba lehetőség, hogy beletekintsek az appba, ez nagyon szimpatikus ötlet. Így egy előre adatokkal felöltött képernyő jelenik meg, ahol ki lehet próbálni a funkciókat. Hasznos.

Az kezdeti jó élmény után a tranzakció felvétele sajnos nem hozta a vártakat. A design itt már nekem túl egyszerű volt, ennek ellenére a felvétel kicsit hosszas és további negatívum, hogy az adatokat nem cache-elik, ebből kifolyólag a gombokra tapintva többször is várni kellett, míg letölti azokat, ami nagyon csökkenti a felhasználói élményt.

Bal oldalról elérhető egy menü, amiben nagyon sok opció található, talán egy kicsit túl sok is ha a fenti specifikációt lefedő alkalmazáshoz hasonlítom. Összességében az alkalmazás közepes a többihez képest, kicsit túl bonyolultnak érzem és szerintem több funkciót tettek bele mint amennyire szüksége van egy felhasználónak.

### TapToTrack[[11]](#footnote-11)

Az alkalmazás rögtön a kiadás összesítő nézeten fogad, ahonnan egy-egy, a navigációs sávban elhelyezkedő gombbal lehet átváltani a történeti/chart nézetre illetve a beállításokhoz menni. A tranzakció felvétele kényelmes és gyors, egy apróság kivételével, a leírás mező tapinthatóságát illetően. Összességében letisztult, egyszerű és könnyen használható a felvétel része.

A múltbeli tranzakciók megtekintése véleményem szerint átlagos, a többi alkalmazáshoz képest. Felhasználókezelés tekintetében csak a saját, felhasználónév/jelszavas változatot támogatják.

### Spender[[12]](#footnote-12)

Az alkalmazás nagyon letisztult, modern dizájnnal rendelkezik. Rögtön a tranzakció rögzítő képernyő fogad induláskor, a kategória kiválasztása is egyszerű és konfigurálható. Az időpont beállítása is kézreáll. Van prémium szolgáltatás, ami extra funkciókat engedélyez, mint például saját kategória felvétele, büdzsé beállítása, különböző chart-ok generálása, de ezekre nem ereztem szükséget, e nélkül is jól használható. Minimális negatívum, hogy a már felvett tranzakciók visszanézése nekem nem volt elsőre könnyen értelmezhető.

Összességében nagyon egyszerű, letisztult és jól használható alkalmazás, ezt találtam mind közül a legjobban használhatónak.

# Felhasznált technológiák

## Kliens

### iOS

Az iOS alkalmazás fejlesztését választottam feladatomul, több ok miatt is. Három éve foglalkozom iOS szoftverfejlesztéssel, előtte egy évig Android mobil alkalmazásokkal foglalkoztam szabadidőmben / egyetemen. Személyes véleményem, hogy az Apple által képviselt szoftverfejlesztési elvek jóval letisztultabbá, könnyebben átláthatóvá teszik a munkát az iOS rendszerrel, mint legnagyobb vetélytársánál.

### iOS történeti hátter [3]

Az iOS az iPhone, iPod touch, iPad termékek operációs rendszere. Ugyan azokra az alapokra épül mint a macOS, a UI miatt a kettő nem kompatibilis egymással. A Unix alapú Darwin-ra épülő XNU hibrid kernel adja az OS alapját [4], ami magába foglal monolitikus-, és mikro-kerneleket is. Mindkét technológia jó tulajdonságait ötvözi, úgy mint a mikro-kernel üzenet továbbítási képességei, ami a modularitást növeli, vagy a monolitikus-kernel sebességbeli fölényét, néhány kritikus feladat végrehajtásánál. Az XNU ARM, IA-32 illetve x86-64 processzorokon fut, egy és több processzort (SMP - Symmetric multiprocessing) is támogatva.

Az iOS egy alapvetően zárt rendszer, azon futó szoftvereket csak erős megkötések mellett lehet fejleszteni, többek között, hogy csak iOS SDK-val macOS alatt lehetséges. Ezek terjesztése többféle képpen lehetséges, szabadon felhasználhatóan (bárki által, korlátlan eszközre telepíthetően) csak az Apple App Store-on keresztül. Egy másik módja a terjesztésnek (Ad-Hoc), hogy előre meghatározott eszközökre (UDID azonosító regisztrálásával, maximum 100 db) telepíthetők az alkalmazások csoportok, szervezetek számára. A harmadik pedig cégen belüli használatra az Enterprise, amihez nincs konkrét szám kötve, de gyanúsan sok telepítés után az Apple letilthatja. Az App Store-ba csak azt követően kerülhet alkalmazás, hogy az egy felülvizsgálaton esik át. Itt meg kell felelni bizonyos kritériumoknak [[13]](#footnote-13), majd ezt követően (állítólag) emberi tesztelésen is átmegy, ez általában 2-10 napot vesz igénybe és ha nem utasítják vissza, akkor onnantól elérhető publikusan.

A készülékeket illetően, 2007-ben jelent meg az iPhone 1.0 majd fél évvel később tették nyilvánosan elérhetővé az iOS SDK-t, amivel 3rd party alkalmazások fejlesztése lehetővé vált. Az App Store-t 2008 márciusában nyitották meg a publikum számára. 2010 volt a következő mérföldkő, az iPad megjelenése, innentől iOS a rendszer neve, addig iPhone OS volt hivatalosan. 2015-ben jelent meg az Apple Watch (Watch OS) és az Apple TV új operációs rendszerrel a TVOS-sel, amikre szintén fejleszthetők 3rd party alkalmazások.

Az Apple által támogatott fejlesztői környezet az XCode, amiben Objective-C vagy Swift nyelven lehet fejleszteni. (C-ben és C++-ban írt könyvtár, kódrészlet is felhasználható). Lehetőség van más eszközzel is fejleszteni alkalmazást, erre több opció van. Egyik, hogy egy másik nyelven írt kódot fordítunk le bináris iOS kódra valamilyen szoftver segítségével, erre a ma talán legismertebb környezet a Xamarin, ami C# kódot fordít (akár Android és Windows Phone-on is futtatható állományra). Egy másik megközelítés, hogy egy natív iOS alkalmazásba egy weboldalt töltünk be, amiben HTML/CSS/JavaScript vagy más WebKit által támogatott technológia jeleníti meg a tartalmat. Egy másik lehetőséget kínál a Unity[6], játékok fejlesztésére. Ez egy cross platform játék engine, ami iOS-en OpenGL ES API[[14]](#footnote-14) használatával fut.

A legtöbben XCode-ot használnak alkalmazás fejlesztésre, mivel ez az Apple saját terméke, ezért mind közül (iOS specifikusan) ez a legjobban támogatott, mindig up to date. Az XCode egy iOS szimulátort biztosít a fejlesztőnek, amivel iPhone, iPad és Apple Watch szimulálható. Valójában a kód MacOS-re fordul, ezért a teljesítménye egy hagyományos emulátorhoz[[15]](#footnote-15) képest nagyon jó.

Az iOS alkalmazások egy-egy sandbox környezetben futnak. Ez sok limitációval jár, mint háttértár elérésének, alkalmazások közötti kommunikáció korlátozása, limitált erőforrásokkal rendelkezik. Ezekért (és még sok más limitációért) cserébe a rendszer által teljesen ellenőrizetten futnak az alkalmazások, ami stabilitást és biztonságot jelent rendszer szinten. Garbage collection[[16]](#footnote-16) nincs, helyette az ARC (Automatic Reference Counting) tartja számon és szabadítja fel az objektumokat a memóriából.

Az alkalmazások framework-ökön keresztül érik el az iOS funkcióit, egy framework DLL, hozzá tartozó erőforrások és header-ök-ből áll. Az iOS számos ilyen framework-ön keresztül érhető el, például a Foundation (alapvető segédosztályok), UIKit (UI, alkalmazás életciklus, gesztusok, stb.) vagy a AVFoundation (audió és videó kezelés).

### Swift

####<Ezt folytassam?>####

Ahogy fentebb említettem, az Apple által támogatott egyik programozási nyelv a Swift. Ezt 2014-ben mutatták be, az Apple saját fejlesztésű nyelve, 2015 decemberétől open source, Apache Licence-szel elérhető bárki számára.

A swift multiparadigmás nyelv, támogatja az objektum orientált, funkcionális, struktúrált és imperatív megközelítésű programozást is. [8] A swift fordítási időben típusozott nyelv,

Jelenlegi verziója a Swift 3.0, nagy ütemben folyik a fejlesztése (immáron GitHub-on).

## Szerver: .NET

## Frontend

## REST kommunikáció

## Tesztelés XCode-ban

A tesztelés fontossága a szoftverfejlesztésben megkérdőjelezhetetlen. A funkcionálisan jól működő, hibamentes és skálázható kód jelzáloga tud lenni ha megfelelő lefedettséggel tesztelik a kódot. A tesztelés legalapvetőbb és minden fejlesztő által végzett formája, hogy lefuttatja a kódot és kipróbálja, hogy megfelelően működik-e. Azonban itt kérdés merül fel, hogy a megfelelő működés mit is jelent valójában. Heurisztikusan kipróbálja a fejlesztő két-három kézenfekvő, általa várt bementre, majd elkönyveli, hogy működik. Ahhoz, hogy garantálni tudja, hogy biztosan jól működik a kód, ez kevés. Átfogó tesztelés során véletlenszerű, sok fajta, akár hibás bemenetekre is szükség van, hogy minden lehetséges opcióra bebizonyítsa, hogy helyesen fut le a kód, az elvárt eredményt adja a bemenet függvényében.

### Automatizált tesztelés

Ennél vannak ma már sokkal szofisztikáltabb és hatékonyabb módszerek is. Az automatizált tesztelés nagy előnye, nem szükséges emberi beavatkozás, rábízhatjuk egy szoftverre, hogy tesztelje le helyettünk az kérdéses programkódot. Emiatt persze az igénybe vett idő is töredékére csökken, mint a kézi tesztelésnél, amellett, hogy kizárja a tesztelés közbeni hibavétés esélyét. Automatizált tesztek futtatásával tudja a fejlesztő bizonyítani, hogy a megírt kód megfelelően működik, illetve a kód inkrementális fejlesztés során nem rontott el semmilyen olyan funkcionalitást, amely korábban már működött.

### Tesztelés támogatása XCode-ban

Az XCode alapvetően három különböző tesztelési metodikát támogat.

* Funkcionális teszt, az üzleti logika és a kód funkcionális működésének ellenőrzésére,
* Teljesítmény teszt, a végrehajtási idő mérésére
* Felhasználói felület teszt, a megjelenő felületi elemek és azok közötti interakciók helyességének ellenőrzésére.

A funkcionális és performancia tesztek tulajdonképpen függvények egy-egy működésének ellenőrzésére. Minden teszteset felállít egy tesztkörnyezetet (setUp), végrehajtja a megírt teszteket majd felszabadítja a környezetet. A felhasználói feleület tesztek a program nagyon magas szintű tesztelését teszik lehetővé, olyan módon, hogy szimulálják, mintha a felhasználó használná azt. Ezen módszer tulajdonképpen egyfajta integrációs teszt is, mert egyben teszteli a program részeit, moduljait, hogy azok egy egységként helyesen és hibamentesen tudnak-e működni.

A funkcionális tesztelés legelterjedtebb módja a Unit tesztelés. Egy kód unit az a legkisebb tesztelhető kód-egység, ami magában megvalósít egy logikailag ellenőrizhető funkciót. Leggyakoribb használati esetük (mint már említettem), hogy egy időpillanatban az összes addigi funkcióra vannak már írva unit tesztek, mindegyik sikeresen lefut és amikor ezután kódot írunk, akkor ellenőrizzük, hogy továbbra is minden unit teszt sikeres-e, azaz új kód írásával nem rontottunk-e el egy már meglévő működő részt. A tesztesetek eredményét egy előre definiálttal hasonlítja össze, ez dönti el, hogy az sikeres volt-e vagy sem.

### Az XCTest keretrendszer

Az XCode tesztelési keretrendszere a XCTest. Amikor létrehoz a fejlesztő egy új XCode projektet, az rögtön generál hozzá egy teszt targetet is, amit külön tudunk majd futtatni tesztelési célból, így nem kell az alkalmazás minden részét fordítani illetve futtatni a tesztek ellenőrzéséhez.

Egy teszteset létrehozásához egy osztályra van szüksége a fejlesztőnek, ami az XCTestCase ősosztályból származik. Ebben az osztályban minden olyan függvény ami a „test” előtaggal kezdődik, le lesz futtatva és kiértékelve egy unit teszt-ként. Ebben a leszármazott osztályban továbbá opcionálisan felül lehet írni a következő két metódust, ami minden egyes teszt metódus előtt és után meg fog hívódni.

* setUp() – Ez a metódus hívódik minden „test” előtaggal keződő függvény előtt. Itt kell beállítani a tesztkörnyezetet amennyiben szükség van rá (változókat inicializálni, memóriabeli adatbázist létrehozni, stb.).
* tearDown() – Ez a metódus hívódik minden teszt metódus után. Ebben az ellenkezőjét kell elvégezni, felszabadítani azt a környezetet amit a setUp függvényben létrehozunk, illetve a környezetet olyan állapotba állítani, hogy semlegesítsük az előző teszteset futása során bekövetkezett változásokat.

A tesztek futásának ellenőrzésére az XCTAssert metódus család áll rendelkezésre a XCTest keretrendszerben. Lényege minden metódusnak ezek közül ugyanaz, állítunk egy logikailag igazra vagy hamisra kiértékelhető feltételt, aminek teljesülésétől függ a teszteset sikeressége. A teljesség igénye nélkül néhány példa:

* Sikeres a teszt, ha a (String) name változó nem üres

XCTAssert(!name.isEmpty)

* Sikeres a teszt, ha a person objektum létezik (nem nil)

XCTAssertNotNil(person)

* Ellenőrizhetjük két objektum egyenlőségét (ilyenkor a tömbben tárolt elemeknek meg kell valósítania az Equatable protocol-t, amivel definiálhatjuk, hogy mikor egyenlő két elem) A példában a Person osztály két példánya akkor egyenlő, ha megegyezik az idNumber tulajdonságuk. Mindkét assertation sikeresnek értékelődik ki.

struct Person: Equatable {

var name: String

var motherName: String

var idNumber: String

var age: Int

}

func ==(lhs: Person, rhs: Person) -> Bool {

return lhs.idNumber == rhs.idNumber

}

let steve = Person(name: "Steve", motherName: "Mom", idNumber: "abc123", age: 16)

let patrick = Person(name: "Patrick", motherName: "Mother", idNumber: "abc456", age: 23)

let oldSteve = Person(name: "Old Steve", motherName: "Mom", idNumber: "abc123", age: 46)

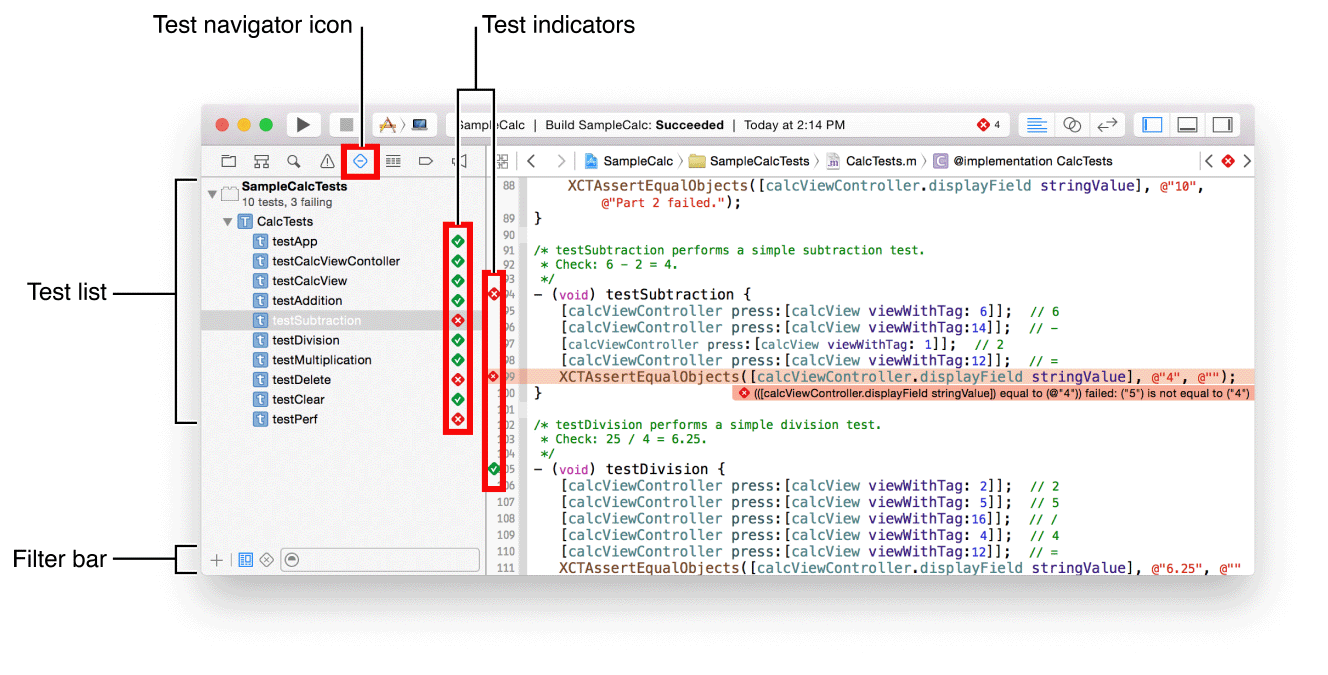
// Sikeres lefutás

XCTAssertNotEqual(steve, patrick)

// Sikeres lefutás

XCTAssertEqual(steve, oldSteve)

A tesztelési felület a következő ábrán látható:



A zölddel jelölt tesztek sikeresen futottak le, míg a pirossal jelöltek sikertelenül. Egyesével is elindíthatók a tesztek, ha a zöld/piros ikonra kattint a fejlesztő.

# A rendszer tervezése

## A rendszer felépítése

## Adatmodellek

## Szerver architektúra

## iOS kliens architektúra

### Az alkalmazás felépítése

Az iOS mobil alkalmazás tervezésénél figyelembe vettem az Apple által leírt kezelői felület irányelveket[[17]](#footnote-17). Ezek vonatkoznak gomb méretektől elkezdve színeken át, betűméretekre, helykihasználásra és sok minden másra, ami mind azt a célt szolgálja, hogy a felhasználók a számukra a legkényelmesebb és legkézenfekvőbb módon érjék el a funkciókat.

Az alkalmazásban három fő nézet van, az új kiadás, a kimutatások és a beállítások. A fő nézetek egymás mellett helyezkednek el egy UIPageViewController-ben, amik között akár swipe gesztúrával, akár gombnyomással is válthat a felhasználó. Az alkalmazás a középső nézettel indul, hogy a leggyakoribb use-case induláskor rögtön elérhető legyen, az adatbevitel. A fő nézet helyezkedik el középen, tőle balra a kimutatások és jobbra pedig a beállítások képernyő. További három képernyő érhető el az alkalmazásban, ezek a következők:

• Diagram részletes nézet – kimutatások nézetből

• Kategóriák – új kiadások nézetből

• Korábbi kiadások – új kiadás felvétele nézet

### Programozási minták

Az architektúrális tervezési minták használata a programozásban elengedhetetlen. Nagyon sok ilyen mintát használ egy mai szoftverfejlesztő, akár annak pontos ismerete nélkül, intuitívan is. Egy tervezési minta tulajdonképpen egy tervezési sablon, ami egy már ismert, általános probléma megoldására szolgál egy bizonyos kontextusban.

A programozási minták használata nem csak könnyebbé teszi a szoftverfejlesztők feladatát, hanem egyfajta útmutatás is adnak, hogy egy ismert problémát miként tudunk a legkönnyebben és leghatékonyabban megoldani. Mobil szoftverfejlesztés témakörre korlátozódva kijelenthető, hogy egy jó architektúra a következő előnyökkel ruházza fel a rendszert:

* jól elosztott felelősségek az entitások között, szigorú szabályokkal,
* könnyű és teljes körű tesztelhetőség,
* könnyű és tiszta fejlesztési folyamat valamint jó karbantarthatóság,
* egyezményes mintha használatával könnyen átadható kód más fejlesztőnek.

iOS fejlesztés témakörben egy ismert és sokat vitatott problémakör, hogy az alábbi három feladat feladat kezeléséért milyen entitások felelősek :

* a tárolt adatokhoz való hozzáférés,
* a felhasználói felület létrehozása,
* az adatok összekötése a felhasználói felülettel:
  + adatok megjelenítése a felületen,
  + a felületen generált / módosított adatok eljuttatása a tároló réteghez.

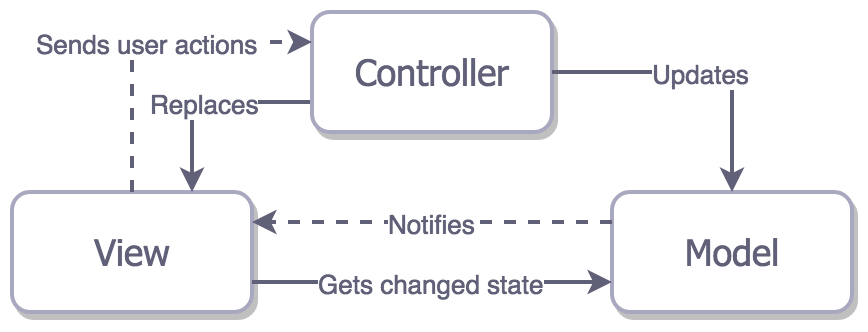
Az az alapvető megoldás, hogy egy entitás felelőssége legyen megoldani mind a három problémát azt okozná, hogy a kód elburjánzana, nehezen karbantartható és újrafelhasználható lenne. Emellett a hibakeresést és nagyon megnehezíteni az összeolvadt és el nem különített felelősségű objektumokban. Egyértelmű, hogy a felelősségeket szét kell választani, erre a kérdésre kerestem én a lehető legjobb választ.

#### Az Apple megoldása - MVC

Az Apple javasol a fentebb felvetett problémára egy megoldást, az MVC (Model View Controller) egy általuk módosított változatát.

Az eredeti MVC architektúrális minta a rendszer entitásait és fentebb felsorolt felelősségeket három csoportba osztja:

* Model: az adatok eléréséért és kezeléséért felel
* View: a megjelenítési réteg kezelése a felelőssége
* Controller: mediátor a View és a Model között, rajta keresztül éri el a View a Model által kezelt adatokat és az ő felelőssége frissíteni a Model-t ha a View manipulálja az adatot és fordítva, frissíteni a View-t, ha a Model-ben tároltak változtak.

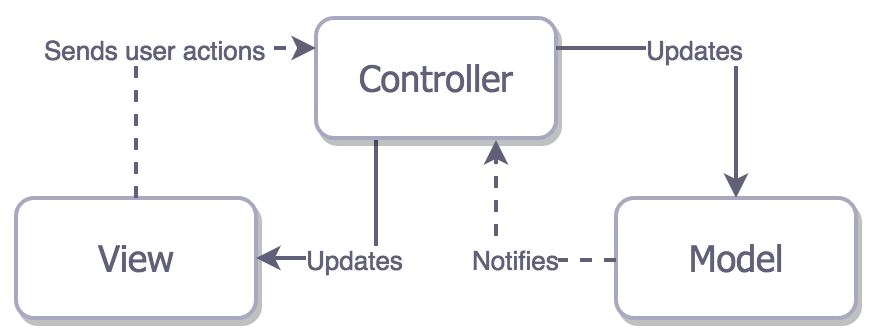


Ebben a mintában a View igazából állapotmentes, tulajdonképpen a Controller létrehozza amikor a Model megváltozott. Ez azt eredményezi, hogy mind a három entitásnak ismernie kell a másik kettőt, ami nagyon nehézzé teszi az újrafelhasználhatóságot fejlesztéskor. Ezért hozta létre az Apple ennek a mintának egy általuk módosított, ebben a környezetben jobban használható változatát.

#### Az Apple féle MVC – teoretikusan

Az Apple által módosított változatban az entitások ugyanazok maradnak, a közöttük történő kommunikáció változik.

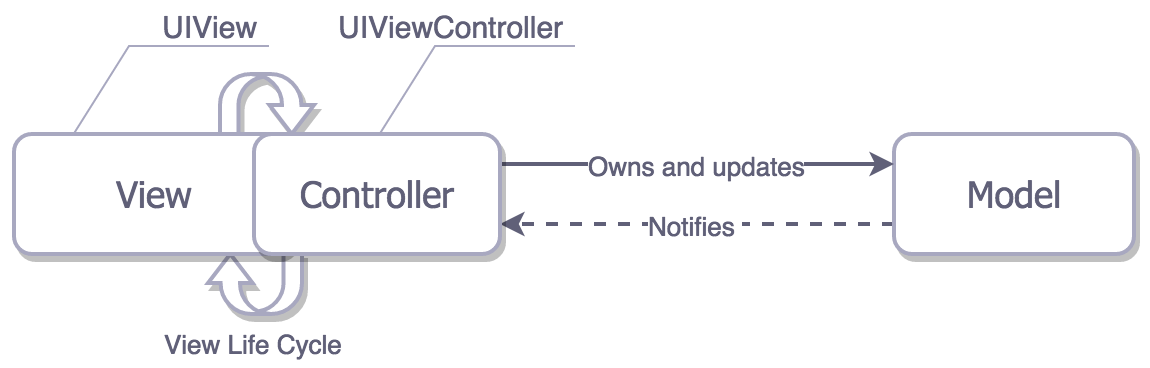
A Controller egy mediátorként viselkedik a View és a Model között, úgy, hogy azoknak ne kelljen ismerniük egymást. Itt a View és a Model újrafelhasználható marad, csak a Controller-lesz az az entitás amit az adott üzleti logika szerint kell módosítanunk és illeszteni a View és Model között. Ez egy elfogadható kompromisszum.



Elméletben ez egy teljesen jól használható és kigondolt minta. A valóság azonban egy kicsit más, amikor valós iOS szoftverfejlesztésre és az Apple-féle MVC használatára kerül sor.

#### Az Apple-féle MVC – a gyakorlatban

A Cocoa MVC rövidítést sokan tréfásan Massive View Controller-ként oldják fel, ez nem véletlen. Az Apple-féle MVC arra ösztönözi a fejlesztőt, hogy a Model-t leválasztva a View és Controller entitásokat összemossa a ViewController-ekben azzal, hogy a View-k életciklusa nagyon nagy mértékben van belevonva a ViewControllerbe.



A View-k összes feladata a gyakorlatban kis túlzással arra korlátozódik, hogy eseményeket továbbítsanak a Controller-jük felé. A Controller feladat lesz, hogy a View-t adattal töltse fel, kezelje az eseményeket, hálózati hívásokat menedzseljen, egyszóval majdnem minden.

Az igazi probléma akkor jelentkezik, amikor tesztelésre kerül sor és ott jövünk rá, hogy csak a Model-t tudjuk igazából tesztelni, mert az üzleti logika teszteléséhez a View és a Controller összeolvadása miatt szükségünk van a UI elemek renderelésére, az életciklusuk követésére, stb.

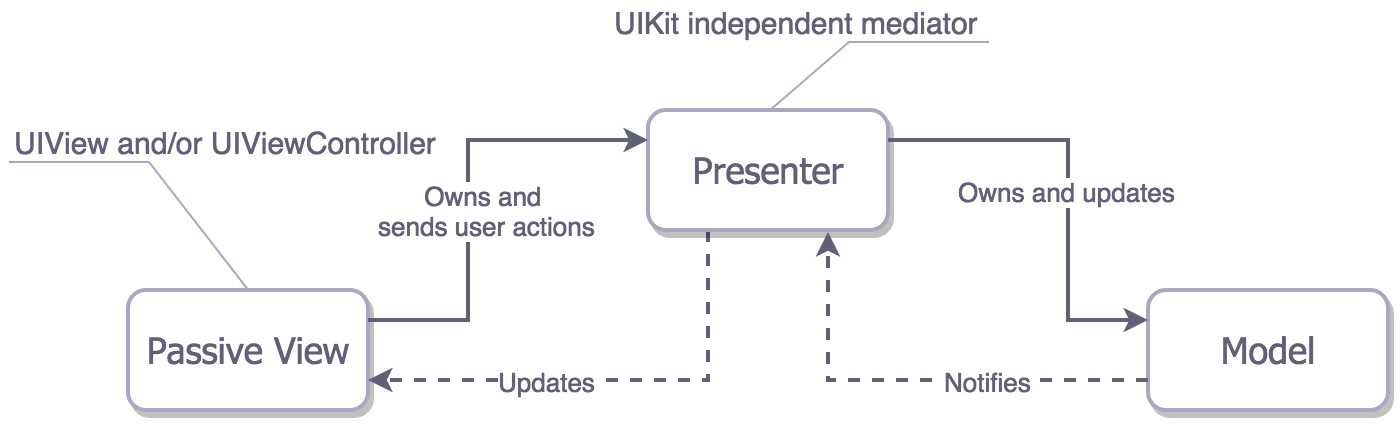
Összefoglalva, az Apple által továbbfejlesztett MVC a valóságban elég nehezen tesztelhető kódot eredményez és a felelősségek sem tisztázottak, sok helyen megszegik az elméletben kiosztott felelősségeket, ámbár meredek tanulási görbét mutat és kisebb alkalmazásoknál az implementációt nagy mértékben fel tudja gyorsítani.

Azonban ha nagyobb volumenű alkalmazást tervezünk, amiben fontos a tesztelhetőség, a karbantarthatóság és a felelősségek elkülönítése, akkor egy másik architektúrális mintára van szükség. A továbbiakban bemutatok kettőt ezek közül az MVVM és a MVP mintákat majd részletesen bemutatom, hogy melyik mellet döntöttem és miért.

### MVVM – Model View ViewModel

<coming soon>

### MVP – Model View Presenter



Az MVP architektúra a fenti ábrán látható. Nagyon hasonlít arra a mintára, amit az Apple MVC-nek nevez és fentebb olvashattak róla. Viszont van egy nagyon fontos különbség a kettő között, míg az előbbinél a View és a Controller életciklusa és sok más tulajdonsága szinte összeolvadt, itt a Presenter entitás semmit nem tud a View életciklusáról és annak semmilyen működéséről, csakis és kizárólag azért felel, hogy szóljon a View-nak illetve a Model-nek az egymást érintő változásokról.

A fenti mintával szemben, itt a UIViewControllerek a View szerepét fogják betölteni és nem pedig a Controller-ét, ami magába olvassza a View-t is. Ez a része az az MVP-nek, ami a nagyon jó tesztelhetőséget biztosítja, mivel minden PassiveView – Presenter – Model esemény- / adatáramlás interfészeken keresztül történik, így bármelyik könnyedén lecserélhető illetve mock[[18]](#footnote-18)-olható. Ennek az ára a fejlesztési idő (kezdetben) jelentős növekedése, mivel mind az adat, mind az események kötését nekünk kell megírnunk, úgy, hogy – mint írtam – minden entitás interfészeken keresztül kommunikáljon a szomszédos rétegekkel.

Egy nyitott kérdés marad az MVP-vel kapcsolatban, mégpedig, hogy kinek a felelőssége lesz létrehozni és összerakni a három egymástól teljesen elszeparált entitást. Erre egy bevett módszer, hogy létrehozunk egy alkalmazás szintű, statikus Router osztályt, ami az entitás-hármasok összerakásáért és a nézetről-nézetre való navigálásért fog felelni. Illetve ahogy a dolgozatban később olvashatnak róla én ezt kombináltam egy másik megközelítéssel, a dependency injection-nel, ami a kód nagyon jó újra felhasználhatóságát és karbantarthatóságát garantálja.

Összességében elmondható a leírt MVP-ről, hogy megfelel azoknak az elvárásoknak amiket az elején támasztottunk egy jó architektúrával szemben. Nagyon könnyen és teljeskörűen tesztelhetőköszönhetően annak, hogy az üzleti logika teljesen le van választva a megjelenítési rétegről. Továbbá a későbbi karbantarthatóságot és rugalmasságot azzal biztosítja, hogy mivel el vannak választva egymástól az entitások - csak interfészeken keresztül ismerik a másikat - így egy konkrét implementációja az adott Model/View/Presenter interfésznek teljes szabadságot ad a fejlesztőnek a változtathatóságban anélkül, hogy a többi szereplőn változtatni kellene.

### Dependency injection általánosan

A dependency injection egy programozási minta, ami a függőség megfordítás elvén (IoC – Inversion of Control)[[19]](#footnote-19) alapszik. Ennek a mintának a megfelelő használatával lazán csatolt komponensekre tudjuk felbontani az alkalmazást, ami nagyban hozzájárul a kód tesztelhetőségének és karbantarthatóságának növeléséhez. A minta lényege, hogy az objektumok létrehozását teljesen leválasztjuk attól az osztálytól, ahol azt használni akarjuk. Az adott helyen csak annyit írunk le, hogy ott szükség van egy X interfésznek megfelelő entitásra, hogy ki és hogyan hozza azt létre, már nem az azt használó objektum felelőssége.

#### iOS specifikusan

### Szálkezelés

## REST API tervezése

## iOS – Android közös tervezése lehetőségek

## Singleton

A singleton minta annak a problémának a megoldására szolgál, amikor egy objektumról biztosítani akarjuk, hogy pontosan egy példány fog létezni belőle. A minta használatával elővigyázatosnak kell lenni, mivel saját magáról tárol egy referenciát, így soha nem fog tudni felszabadulni, ezért alaposan megfontolandó, hogy mikor van szükség a használatára. Az alkalmazásunkban az ApplicationRouter osztály singleton, mivel ebből pontosan egy példánynak szabad léteznie. Ez az osztály felel az alkalmazás képernyői közötti navigációért és az MVP minta szerinti entitás hármasok összállításáért.

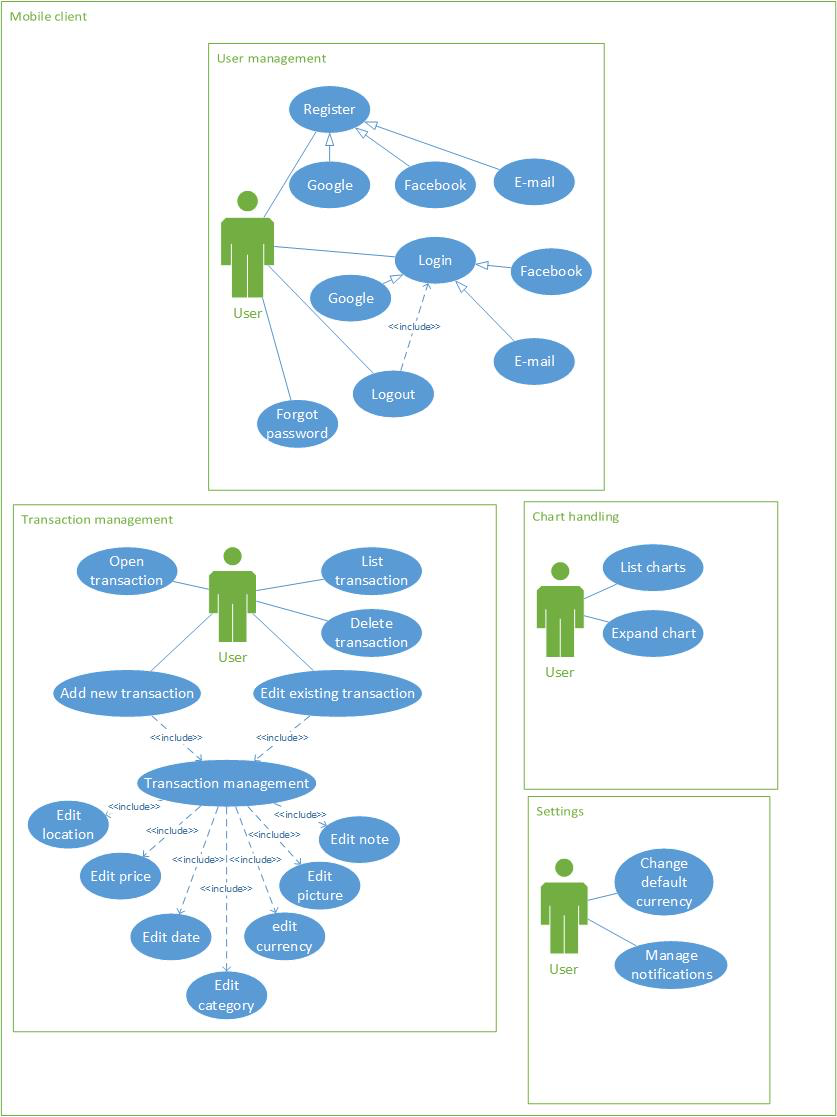
class ApplicationRouter {

static let sharedInstance = ApplicationRouter()

private init() {

// Egyszeri inicializáló kód

}

Függelék

Irodalomjegyzék

1. iOS Architecture patterns: <https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52#.mkod5pl0e>
2. <https://developer.apple.com/library/ios/documentation/ToolsLanguages/Conceptual/Xcode_Overview/UnitTesting.html>
3. <https://www.aut.bme.hu/Course/ios> - előadásokon vetített diák anyaga
4. <https://en.wikipedia.org/wiki/XNU>
5. <https://hu.wikipedia.org/wiki/Alkalmazásprogramozási_felület>
6. <https://docs.unity3d.com>
7. <https://hu.wikipedia.org/wiki/Szemétgyűjtés>
8. <https://swift.org/about/#swiftorg-and-open-source>

1. <https://play.google.com/store/apps/details?id=com.kpmoney.android> [↑](#footnote-ref-1)
2. <https://templates.office.com/en-us/Personal-money-tracker-TM02780244> [↑](#footnote-ref-2)
3. <https://itunes.apple.com/hu/app/moneycontrol-income-expense/id634892482?mt=12> [↑](#footnote-ref-3)
4. <https://www.microsoft.com/hu-hu/store/p/money-tracker-free/9wzdncrdklch> [↑](#footnote-ref-4)
5. https://itunes.apple.com/hu/app/daily-account-manager-income/id1168406615?mt=8 [↑](#footnote-ref-5)
6. https://itunes.apple.com/hu/app/foreceipt-receipt-manager/id721048897?mt=8 [↑](#footnote-ref-6)
7. https://itunes.apple.com/hu/app/toshl-finance-expense-tracker/id921590251?mt=8 [↑](#footnote-ref-7)
8. https://itunes.apple.com/hu/app/money-manager-pro-financial/id465646699?mt=8 [↑](#footnote-ref-8)
9. https://itunes.apple.com/hu/app/quick-money-recorder-expense/id554067740?mt=8 [↑](#footnote-ref-9)
10. https://itunes.apple.com/hu/app/buxfer-expense-tracker-budget/id312260980?mt=8 [↑](#footnote-ref-10)
11. https://itunes.apple.com/hu/app/live-expenses-expense-tracker/id486195690?mt=8 [↑](#footnote-ref-11)
12. https://itunes.apple.com/hu/app/spender-money-management/id637562972?mt=8 [↑](#footnote-ref-12)
13. https://developer.apple.com/app-store/review/guidelines/ [↑](#footnote-ref-13)
14. API (Application Programming Interface): egy program vagy rendszerprogram azon eljárásainak (szolgáltatásainak) és azok használatának dokumentációja, amelyet más programok felhasználhatnak. Egy nyilvános API segítségével lehetséges egy programrendszer szolgáltatásait használni anélkül, hogy annak belső működését ismerni kellene. [5] [↑](#footnote-ref-14)
15. A szimulátor egy eszköz viselkedését próbálja leutánozni minél jobban, míg egy emulátor az eszköz belső működésének pontos másolatára hivatott. [↑](#footnote-ref-15)
16. [7] A számítástechnikában a szemétgyűjtés (angolul garbage collection) egy formája a biztonságos memóriakezelésnek. A szemétgyűjtő (angolul garbage collector vagy röviden GC) megkísérli eltávolítani a memóriából azokat az objektumokat, amelyeket az alkalmazás már nem használ. [↑](#footnote-ref-16)
17. <https://developer.apple.com/ios/human-interface-guidelines/overview/design-principles/> [↑](#footnote-ref-17)
18. Mock-olásnak nevezzük, amikor tesztelés céljából, vagy egy korábbi fejlesztési állapotban a valós adatokat/funkcionalitást - az azt használó/ismerő entitás számára átlátszóan - egyszerűen leírható példa (dummy) adat/funkciókra cseréljük [↑](#footnote-ref-18)
19. A kód interfészektől függjön és ne megvalósításoktól [↑](#footnote-ref-19)