Szakdolgozat feladat

**Vihari Réka**

Mérnökinformatikus hallgató részére

Univerzális Rendezvény Applikáció iOS Platformra

Az okostelefonok elterjedésével egyre nagyobb szerepet töltenek be életünkben a mobilapplikációk. Ezeken az alkalmazásokon keresztül érjük el barátainkat, készítünk edzéstervet, de akár egy olyan mindennapi feladatot is, mint a főzés, ezek segítségével oldunk meg. Az élet számos területén megkönnyíti és lerövidíti az emberi munkát.

Míg pár évvel ezelőtt a mobilitást a laptopok jelentették, ma már a telefonjaink is képesek nagy mértékben helyettesíteni őket. A legtöbb számítógépen használt alkalmazás már az okostelefonunkon is elérhető. A mai rohanó világban pedig egyre nagyobb az igény a könnyen hordozható és kezelhető mobil készülékekre.

A hallgató feladata egy olyan alkalmazás megtervezése és implementálása iOS platformra, amely tetszőleges rendezvények szervezését és lebonyolítását könnyíti meg szolgáltatásaival, legyen szó akár egy esküvőről, bicikli túráról vagy egy sítáborról a hegyekben. A résztvevők az applikáción keresztül nyomon tudják követni a programokat, azok helyszínét, és egyéb, a rendezvényhez kapcsolódó információkhoz férhetnek hozzá.

A hallgató feladatának a következőkre kell kiterjednie:

* Mutassa be az iOS platformot fejlesztői szemszögből
* Ismertesse az alkalmazással szembeni funkcionális követelményeket
* Tervezze meg az alkalmazás architektúráját
* Készítse el a megtervezett iOS alkalmazás prototípusát

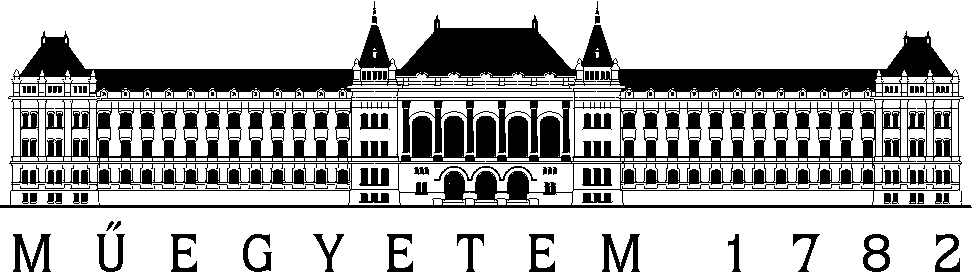
**Tanszéki konzulens:** Dr. Forstner Bertalan

Budapest, 2018. október 3.

Dr. Charaf Hassan

egyetemi tanár

tanszékvezető



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Vihari Réka

Univerzális Rendezvény Applikáció

iOS platformra

Konzulens

BUDAPEST, 2018

Tartalomjegyzék

[Szakdolgozat feladat 5](#_Toc530833373)

[Összefoglaló 10](#_Toc530833374)

[Abstract 11](#_Toc530833375)

[1. Bevezetés 12](#_Toc530833376)

[1.1 Mobilpiaci kutatás 13](#_Toc530833377)

[2. Az iOS platform bemutatása 16](#_Toc530833378)

[2.1 Az operációs rendszer fejlődése 16](#_Toc530833379)

[2.1.1 iOS 9 18](#_Toc530833380)

[2.1.2 iOS 10 19](#_Toc530833381)

[2.1.3 iOS 11 19](#_Toc530833382)

[2.1.4 iOS 12 20](#_Toc530833383)

[2.2. Swift 21](#_Toc530833384)

[2.3. Xcode 21](#_Toc530833385)

[2.4. Architektúrális minták 24](#_Toc530833386)

[2.4.1. MVC 24](#_Toc530833387)

[2.4.2. VIPER 26](#_Toc530833388)

[2.4.3. Viper vs MVC 27](#_Toc530833389)

[2.4.4. Konklúzió 27](#_Toc530833390)

[2.5. Verziókezelés 28](#_Toc530833391)

[3. Feladat ismertetése 30](#_Toc530833392)

[3.1. Specifikáció 30](#_Toc530833393)

[4. Technológiák ismertetése 31](#_Toc530833394)

[4.1. CocoaPods 31](#_Toc530833395)

[4.2. JHipster 33](#_Toc530833396)

[5. Tervezés 40](#_Toc530833397)

[5.1. Adatbázis 40](#_Toc530833398)

[5.2. Kommunikáció a szerverrel 42](#_Toc530833408)

[5.2.1 Authentikáció 45](#_Toc530833410)

[6. Megvalósítás 49](#_Toc530833411)

[6.1. Képernyők 49](#_Toc530833412)

[6.2. Funkciók 49](#_Toc530833413)

[6. Tesztelés 56](#_Toc530833414)

[7. Továbbfejlesztési lehetőségek 57](#_Toc530833415)

[8. Összefoglaló 58](#_Toc530833416)

[9. Irodalomjegyzék 59](#_Toc530833417)

Hallgatói nyilatkozat

Alulírott **Vihari Réka**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2018. 11. 22.

...…………………………………………….

Vihari Réka

page4image1684976

Összefoglaló

Az okostelefonok elterjedésével egyre nagyobb szerepet töltenek be életünkben a mobilapplikációk.

Ezeken az alkalmazásokon keresztül érjük el barátainkat, készítünk edzéstervet, de akár egy olyan mindennapi feladatot is, mint a főzés, ezek segítségével oldunk meg. Az élet számos területén megkönnyíti és lerövidíti az emberi munkát. Beépültek magán- és szakmai életünkbe egyaránt.

Az Apple Inc. 2007-ben jelent meg a piacon első iPhone készülékével, amely forradalmi újítás volt a kor telefonjaihoz képest. Az érintőképernyőben lévő lehetőségeket látva tervezték meg az első okostelefonjukat, ami azonnal hatalmas népszerűségre tett szert. Ennek oka a könnyű kezelhetőség és a felhasználói élmény volt, amelyet az új funkciók nyújtottak. Ma a legújabb piacon lévő iPhone az Xs (2018), amely már arcfelismerő funkcióval és vezeték nélküli töltéssel rendelkezik.

A telefonok mindennapos használatával megnövekedett az igény a mobilapplikációkra is. Általában a megrendelők cégek, akik a munkafolyamatuk egy részét szeretnék modernizálni, illetve megkönnyíteni. A mai felhőszolgáltatások révén, telefonunkról bármikor hozzáférhetünk a szükséges adatokhoz. A vállalati dolgozók kommunikálhatnak egymás között is applikációkon keresztül, de akár az ügyfelekkel is. Másrészt pedig egyre többen kapnak kedvet a Startup-ok által készített sikeres mobilapplikációk révén ötleteik megvalósítására. Így sokszor találkozhatunk olyan megrendelőkkel is, akik saját elképzeléseiket szeretnék látni egy újonnan készült alkalmazásban.

Szakdolgozatom célja egy olyan alkalmazás elkészítése iOS platformra, amely tetszőleges rendezvények szervezését és lebonyolítását könnyíti meg szolgáltatásaival, legyen szó akár egy szakmai konferenciáról, bicikli túráról vagy egy sítáborról a hegyekben. A résztvevők az applikáción keresztül nyomon tudják követni a programokat, azok helyszínét, és egyéb, a rendezvényhez kapcsolódó információkhoz férhetnek hozzá.

Dolgozatomban ismertetem az iOS platformra való fejlesztés sajátosságait. Bemutatom az alkalmazás tervezését és implementációját, kitérve a felhasználói felület felépítésére és az alkalmazás architektúrájára is.

Abstract

same in english

# Bevezetés

Egyre nagyobb szerepet töltenek be életünkben a mobilalkalmazások. Gyakorlatilag már mindenhova visszük magunkkal telefonunkat. Ez az új szokás adta a gyökerét mobilapplikációs széleskörű elterjedséhez. Az eszközök folyamatos fejlődésével és az általuk nyújtott szolgáltatások bővülésével egyre nagyobb a kereslet ezen a piacon. A mai rohanó világban nagy szerepet játszik, hogy helytől függetlenül el tudjuk érni adatainkat, beleértve a naptárunkat, fényképeinket és akár munkahelyi feladatainkat is. Nagy segítséget nyújtanak a könnyű kezelhetőség és hozzáférhetőség révén.

Manapság már egy nyaralást is megtervezhetünk applikációkon keresztül, a repülőjegy vásárlástól a programok kiválasztásáig. Ezen belül lenne a célja a megtervezett alkalmazásomnak az előre megszervezett események könnyebb lebonyolítása. A megszokott, korábbi megoldás úgy működik egy szervezett programnál, hogy az esemény első napján kiosztanak egy programfüzetet és az alapján kell az időpontokban a megfelelő helyen lenni. Sajnos ez a gyakorlatban általában úgy működik, hogy már aznap elveszítjük ezt a füzetet és igyekszünk emlékeink alapján tájékozódni. Ennek megváltoztatásaképp szeretném ezt a rendszert egy applikációba integrálni, kibővített funkciókkal és már a valós idejű programváltozásokat is kezelve. A telefonunk az események alatt is mindig velünk van, hiszen ezzel ellenőrizzük levelezésünket, készítünk fényképeket és tartjuk a kapcsolatot a többi résztvevővel. Így kézenfekvő megoldásnak tűnt ezt a kommunikációt és lebonyolítást egy helyen összefűzni.

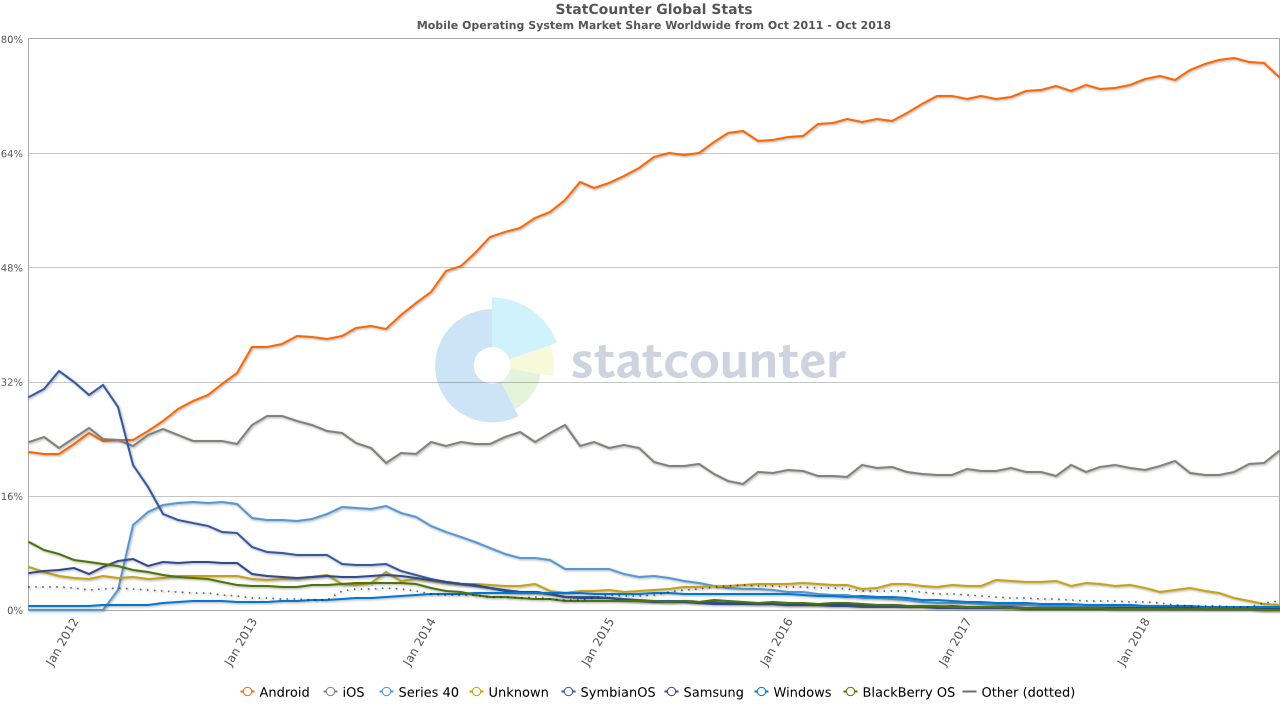
A következő fejezetekben egy általam tervezett alkalmazást mutatok be, amely segíti a kapcsolattartást a résztvevők és a szervezők között. A nyáron lehetőségem volt a BME Automatizálási és Informatikai tanszéken részt venni egy olyan iOS alkalmazás fejlesztésében, amely egy testreszabható, többfajta igényt kielégítő applikáció Itt fogalmazódott meg bennem az ötlet, hogy megtervezzek egy hasonló alkalmazást esemény specifikusan.

Dolgozatomban ismertetem az iOS platformot és a fejlesztői környezetet. Illetve, kitérek az alkalmazások felépítésének fejlődésére is. Továbbá, bemutatom a vállalatoknál gyakran használt verziókezelést is, amely egy több tagú fejlesztői csapatban segíti a hatékony alkalmazás készítést.

A továbbiakban az alkalmazás egyes funkcióit mutatom be, illetve, a hozzá tartozó backend-et is, amellyel válik az alkalmazás univerzálissá. A tervezési és implementálási folyamatok bemutatása után kitérek az alkalmazás tesztelési lehetőségeire is.

## Mobilpiaci kutatás

A fejezet célja a piacon lévő mobil operációs rendszerek bemutatása. Telefon vásárlásnál több operációs rendszer közül választhatunk, melyeknek megvan a maga előnyük és hátrányuk. Igazodhat a felhasználók funkcionális követelményű igényeihez, illetve akár a pénztárcájukhoz is. Az alábbi diagramon látható, hogy az okostelefonok piacán két fő meghatározó szereplő van:



1.1. ábra Mobil operációs rendszer eloszlása

Ez a kettő nem más, mint az Android és az iOS mobil operációs rendszer. Napjainkban a további piaci szereplők a BlackBerry és egyéb kisebb cégek. A Windows Phone fejlesztését a Microsoft mára megszüntette. A kimutatás 2011-től napjainkig mutatja a százalékos eloszlását a rendszerek használatának. Látható, hogy 2011-ben még hasonló elterjedésű volt a két cég, de az Android platform a többi készülékgyártó támogatásának növelésével (LG, HTC, Motorola, Samsung) terjeszkedni kezdett. Így a legolcsóbb okostelefonoktól a drágább készülékekig elérhető vált mindenki számára az Android operációs rendszere. Ehhez képest, az iOS továbbra is csak Apple készülékekre volt elérhető, így az eszközök magas árának köszönhetően az iOS nem növekedett tovább, inkább konstans résztvevője lett a piacnak.

Másrészt, az Android fejlesztője a Google, mely nyílt forráskódúvá tette a szoftvert, ezáltal szabad utat adva a fejlesztőknek. Ehhez képest az Apple iOS rendszere továbbra is zárt forráskódú maradt. Mégis az iOS használók aránya nem csökkent, mivel sok előnyét tudja felmutatni a társával szemben. Ezek között van a teljesítmény, melyet az alábbi diagramon szemléltetek is:



1.2. ábra iOS készülékek gyorsasága Android készülékekhez képest

Az ábra alapján látszik, hogy míg egy videó szerkesztése iPhone X készülékkel 0:42 másodpercet vesz igénybe, addig Android operációs rendszerrel ellátott társainak ez akár több mint a kétszeresébe is telhet. A készülékek nem azonos CPU-val és memóriával rendelkeznek, de látható, hogy jelenleg nincs olyan Andoroid-os készülék, mely hasonló gyorsaságot tudna nyújtani. Nem csak a videó szerkesztésben mutatkozik meg a gyorsaság, hanem a mindennapi alkalmazások használatában is észrevehető. Ha több percet kell várnunk egy alkalmazás betöltéséhez, az nagy mértékben ronthatja a felhasználói élményt.

Továbbá, a csak cégen belüli platform támogatottságnak köszönhetően jobb a hardver és szoftver integrációja. A nemrég bevezetett 3D Touch funkciója említendő meg ez esetben, mely érzékeli a képernyő nyomásának erejét, ezzel is új funkciókat adva az operációs rendszernek. Illetve, ezzel is növelne az egyszerűbb használhatóságot az Android rendszerekhez képest.

Az Apple Store-ban megtalálható alkalmazások kiemelkedőek a Google Play Áruházban megtalálható Android készülékeken elérhető applikációktól. Ez az Apple szigorú követelményeinek is köszönhető, az applikációk publikálásával szemben. Továbbá, az Android nagy eszköztámogatottsága és a különböző gyártók révén, nehéz olyan alkalmazásokat készíteni, melyek minden készüléken a megfelelő teljesítményt és felhasználói élményt nyújtják. Viszont iOS-re fejlesztés során lehetőségünk van minden eszközre és verzióra tesztelni, ezzel ellenőrizve alkalmazásunk megfelelő működését.

Ezen felül, érdemes még megemlíteni az iPhone együttműködését az Apple által forgalmazott Mac gépekkel. Akár telefonhívást vagy üzenet érkezik a telefonunkra, azonnal megjeleníti laptopunkon is. Továbbá, ha mobilunkon böngésszük az Internetet, akkor számítógépünkön megjelenik az opció, hogy ott folytassuk tovább a keresést.

Az iCloud az Apple által fejlesztett felhő tároló, melyből a készülék vásárlásánál 5 GB tárhelyet kapunk ingyenesen. Ezen felül pedig kedvezményes áron juthatunk további tárhelyhez, melyet családtagjainkkal is megoszthatunk. Telefonunkon beállíthatjuk az automatikus szinkronizációt, így nem kell foglalkozni adataink manuális időszerű feltöltésével.

Ezen előnyök alapján érthető, hogy akik eddig iPhone készüléket használnak, kevésbé hajlamosak váltani Android készülékre.

# 2. Az iOS platform bemutatása

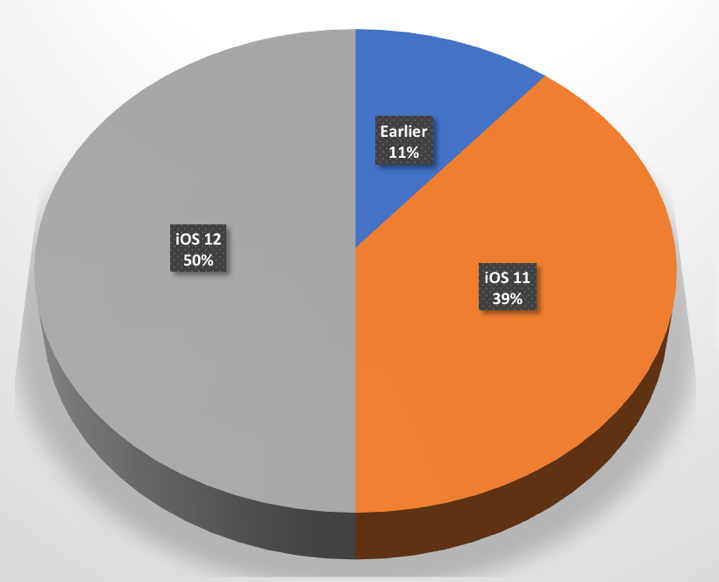
## Az operációs rendszer fejlődése

Az Apple operációs rendszere az iOS, mely iPhone, iPad és iPod eszközökön működik. Mint minden operációs rendszer, ez is tartalmaz beépített alkalmazásokat (például: Üzenetek, Telefon, Internetböngésző, stb.), de a 2. verziója óta lehetőség van fejleszteni is rá külső applikációkat.

Az Apple korábbi elnöke Steve Jobs, a zárt rendszer filozófiáját követve, minden termékkel kapcsolatos folyamatot felügyelet alatt akart tartani. Az elképzelése az volt, hogy az általuk kifejlesztett hardver és szoftver tökéletes és külső fejlesztők bevonása nem szükséges a fejlesztéshez. Így az iPhone-ok megjelenésekor nem is volt lehetőség rájuk külső alkalmazásokat készíteni nem Apple fejlesztőként. A cég később belátta, hogy enélkül a fejlődésre lassú ütemben van csak lehetőség, így engedélyezte a fejlesztést, de a mai napig szigorú követelményeknek kell megfelelnie egy alkalmazásnak, hogy bekerülhessen az Apple Store-ba. Ezen szempontok közé tartozik a biztonság, a teljesítmény, a dizájn és még sok kisebb elvárás. Illetve, csak megvásárolt fejlesztői felhasználóval van lehetőség kezdeményezni alkalmazásunk feltöltését. Illetve, az operációs rendszer továbbra is zárt forráskódú maradt.

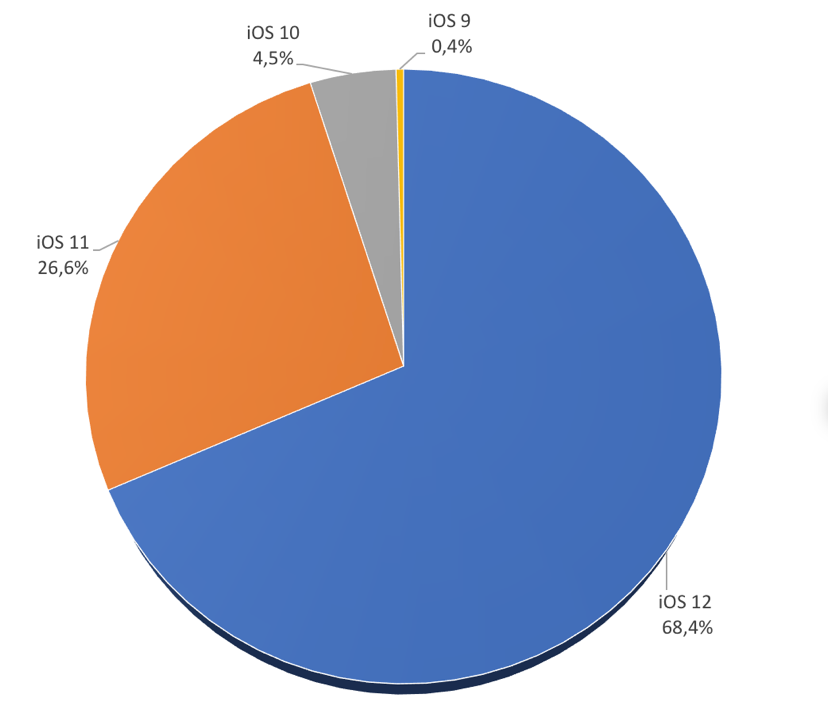
A legelső iOS verzió 2007 január 9-én jelent meg. Ma már a 12. verziónál tartunk és az évek során sok újdonsággal bővültek a funkciók. Az új alkalmazások fejlesztési lehetőségei korlátozva vannak legalább iOS 9 platformra, ennél alacsonyabb verzióra nem tudunk fejleszteni. Az Apple Inc. ezzel is ösztönözi a felhasználókat a nagyobb verzióra váltásra., mivel aki nem frissít, annak nem érhetőek el a készülékére az új alkalmazások. Ezzel a lépéssel kizárták az iPhone 4 és azelőtti készülékeket, mert a készülékre elérhető legmagasabb iOS verzió a 7.1.2, így ezekre új applikációkat már nem lehet fejleszteni.

Az App Store által felmért (2018.10.18.) diagramon látható, hogy az Apple által forgalmazott készülékek, milyen arányban tértek át az új verzióra:



**2.0.1. ábra iOS verzók megoszlása minden Apple készüléken**

Az alábbi diagramon látható az iPhone felhasználók egy 2017.októberében kiadott applikáció (iOS 9 és feletti támogatással) alapján felmért felhasználók verzióhasználata:

****

**2.0.2. ábra iOS verziók megoszlása, egy iOS 9+ applikáció felhasználóinál**

Ebből látható, hogy a felhasználók nagy része már a legújabb verziót használja. Ez annak is köszönhető, hogy minden egyes verzióval javítja az Apple a felhasználói élményt és a kibővített funkciók és hibajavítások ösztönzik a felhasználókat a frissítésre. Itt is meg kell jegyezni, hogy vannak bizonyos készülékek (iPhone 4s, 5, 5c, 5s), amelyekre már ez a legújabb verziófrissítés nem elérhető.

Az alábbi fejezetekben azon verziók újításait szeretném bemutatni, melyeket a megtervezett alkalmazásom támogat.

Említésképpen az előző verziók fontosabb szolgáltatásai:

|  |  |
| --- | --- |
| iPhone OS 1.0 | Telepített, fix alkalmazások (Telefon, Üzenetek, Mail, Naptár) |
| iPhone OS 2.0 | App Store megjelenése, Térkép aplikáció fejlesztése |
| iPhone OS 3.0 | Copy-Paste, Videófelvétel, Push értesítések |
| iPhone OS 4.0 | FaceTime, Airplay, Airprint, Személyes hotspot |
| iPhone OS 5.0 | iCloud, iMessage, iTunes Wi-Fi Sync, Notification Center |
| iPhone OS 6.0 | Apple Térképek, Passbook (most Wallet), Facebook integrálás |
| iPhone OS 7.0 | AirDrop, iTunes Radio, Control Center |
| iPhone OS 8.0 | Apple Music, Apple Play, Family Sharing |

### iOS 9

Az Apple Inc. által fejlesztett iOS 9 mobil operációs rendszert 2015 szeptemberben adták ki. Új applikációk jelentek meg a verzióval és a meglévőket is fejlesztették. A Safari ebben a verzióban már lehetőséget adott a felhasználóknak bizonyos kéretlen tartalmak blokkolására és beállíthatták, hogy milyen mértékben engedélyezik a preferenciáik nyomon követését.

A fejlesztőknek nagy segítséget nyújtott a UIStackView bevezetése, mely megkönnyítette a képernyőn lévő elemek megfelelő elhelyezését. Az alkalmazás fejlesztése közben kijelölhetünk elemeket (szövegek, gombok, stb,), amiket beágyazunk egy UIStackView-ba, melyből választhatunk vízszintes és függőleges rendezést. Ezáltal automatikusan berendeződnek az elemek egymás mellé vagy alá.

Az iPhone 6S és iPhone 6S plus készülékekre bevezették a 3D touch-ot, amellyel a készülék már a képernyő nyomásának erősségét is képes érzékelni, ezáltal új funkciókat adva az érintőképernyő használatának.

Bevezették az alacsony töltöttségű módot, amely egy felugró üzenetet jelenít meg a képernyőn, ha telefonunk 20%-ra merül, megkérdezve, hogy használni kívánjuk-e a funkciót. Ha elfogadjuk, a készülékünk kikapcsolja a háttérben lévő alkalmazások frissítését és tétlenség esetén 30 másodpercen belül automatikusan zárolja telefonunkat. Ezzel a funkcióval akár 1 óra üzemidőt is nyerhetünk.

### iOS 10

Az új verzió 2016 szeptemberében jelent meg. A 10-es verzióra frissítők találkozhattak egy újfajta lezárt állapotú képernyővel (Lock Screen), aminél feloldás nélkül láthatjuk fontosabb alkalmazásainktól érkezett értesítéseinket. A kijelzőnk fentről lefele történő húzásával érhetjük el az összes érkezett értesítésünket.

Bevezették a Control Centert, melyet a képernyő alulról felhúzásával érhetünk el. Itt fontosabb beállításokat végezhetünk el. Ezek közé tartozik a WiFi vagy Bluetooth keresés engedélyezése, a zseblámpa bekapcsolása, a képernyő elforgatási lehetőségének zárolása vagy akár a zenelejátszás irányítása.

A Térkép alkalmazásba beleépítették, hogy jegyezze meg a legutóbbi keresett helyeket és megnyitáskor ajánlja fel azt a célpontot, ahova legtöbbször terveztünk útvonalat.

A Fotó alkalmazást szűrőkkel bővítették, melyek arcfelismeréssel külön albumokba rendezik a képeket a felismert személyekről és földrajzi hely szerinti csoportosítást is alkalmaznak.

Az Üzenetek alkalmazás animációkkal bővült, melyekkel a felhasználók a leírt szöveget animálva vagy akár teljes képernyős animációt is küldhetnek a címzettnek.

Fejlesztők számára az alaklamzásokon belüli üzenet küldséhez az Apple létrehozta az MSMessageAppViewController osztály, mely szabadon felhasználható. Interaktív üzenetek készíthetőek vele, illetve a shouldExpire metódus használatával beállítható, hogy egy üzenet automatikusan törlődjön egy meghatározott idő után.

### iOS 11

Az iOS 11-es verzióját 2017 szeptemberben vált letölthetővé. Ebben a verzióban már az alap felhasználó felületek is új dizájnt kaptak. A Számológép, a Telefon alkalmazás, a 10-es verzióban megjelent Lock Screen és a Control Center teljesen újra lett tervezve. A már meglévő Notification Center beleolvadt a Lock Screen-be így már nem csak az egyes alkalmazások eseméyeit érhettük el itt, hanem naptárunk teendőit, térkép úticél javaslatokat és a kereső mezővel alkalmazásaink gyorsabb elérése vált lehetővé.

A fejlesztők számára az ARKit megjelenése volt a legfőbb funkció, mely segítségével a készülék feltérképezheti a környezetet, melyet a fejlesztők manipulálhatnak és virtuális objektumokat helyezhetnek rá. Az ARKit integrálható 2D és 3D játékokhoz, melyet így kiterjesztett valósággal ruházhatunk fel.

Felhasználói oldalról új funkcióként létrejött a Drag and Drop lehetőség, amely lehetővé teszi, hogy egyik alkalmazásból a másikba helyezzünk át fájlokat.

A billentyűzetbe beépített nyíl segítségével kisebb méretűre változtathatóvá vált, ezzel is biztosítva a növekvő képernyőméret esetén is az egy kézzel kezelhetőséget.

Az App Store is nagy változáson ment keresztül, a teljes dizájnt feljavították és külön szekciókra lettek osztva az alkalmazások a játékoktól.

### iOS 12

Az iOS legújabb verziója, melyet 2018 szeptemberében adtak ki. Az új fejlesztések közé tartozik a csoportos FaceTime lehetősége, 70 új beépített emoji a billentyűzetbe és eSIM támogatás.

Fejlesztői szemszögből újdonságként szerepelt a Siri használata, mely képes a fejlesztett alkalmazáshoz hivatkozásokat megtanulni. Előre meg lehet határozni számára, hogy melyek azok a tevékenységek, amiket a felhasználó használni fog az alkalmazásban. Az alkalmazás ezeket tudja specifikálni hivatkozásokként a Siri-nek. Továbbá, újdonságnak számít még, hogy az alkalmazásoknál engedélyezni lehet az Apple Pencil (ceruza) használatát.

Felhasználói szemszögből újdonság a beállításokból elérhető új Képernyőidő menü, melyben követhetjük, hogy bizonyos alkalmazásokat mennyi ideig használunk az adott napon vagy az utolsó 7 napban. Ennek köszönhetően jobban láthatjuk, hogy mennyi időt töltünk el hasznosan az okostelefonunk használatával vagy éppenséggel mennyi időt töltünk el a közösségi hálók böngészésével. Erre külön kategóriák is el vannak különítve a menüben olvasás, tanulás és közösségi hálók néven.

A zárolt képernyőn (Lock Screen) felugró értesítéseket alkalmazásokra csoportosítva láthatjuk, melynek tetején verem szerűen a legfelső üzenetet szerepel, de rákattintással akár ki is bonthatjuk a zárolt képernyőn belül, hogy az összes értesítést lássuk.

Az iPhone X arcfelismerő rendszeréhez érkezett újításban, már nem csak arcmimikánkat ismeri fel a telefon, hanem a nyelvünket is. Ezáltal a nyelv mozgását is rátudja vetíteni az animojikra.

### Swift

Az Apple forradalmi újítása, melyet Tim Cook – az Apple vezérigazgatója – mutatott be 2014 nyarán az iOS 8 operációs rendszerrel együtt. Az Apple eddigi fejlesztői nyelvét, az Objective-C-t váltotta fel. A teljesen új programozási nyelv ötvözte a C és az Objective-C legjobb funkcióit a C megkötései nélkül: "Objective-C, a C nélkül".

A nyelvet az Apple hozta létre iOS, OS X, watchOS és tvOS platformokra való fejlesztéshez.

A nyelv először zárt forráskóddal indult, de később belátta az Apple, hogy a nyílt forráskód által, külsős fejlesztők segítségének bevonásával gyorsabban haladhat a nyelv fejlődése. Ennek támogatásához az Apple fejlesztői fórumokat is létrehozott, ahol fogadják a nyelv fejlesztéséhez szükséges ötleteket. Ennek köszönhetően 2017-ben már a Swift 4.0 verzióját mutathatták be.

A nyelv az objektum-orientáltság mellett támogatja a strukturált, a procedurális-imperatív és a funkcionális stílusú programozást is. Legjelentősebb előnyei a biztonságos és kényelmes fejlesztés, illetve a jó futásidejű teljesítményre törekvés.

A dolgozatomban a Swift nyelvet használom, melynek egyszerűsítetett szintaxisa és nyelvtana van az Objective-C-hez képest. A Swift nyelv könnyen olvashat és írható. Továbbá, ugyanahhoz a funkcióhoz Swift nyelven kevesebb kódot kell írnunk, mint Objective-C-ben. Ezáltal kevesebb időt is vesz igénybe egy Swift nyelven írt alkalmazás build-elése, mint ha Objective-C-ben írtuk volna.

## Xcode

Az Xcode az Apple által fejlesztett integrált fejlesztői környezet (IDE – Integrated Development Environment), melynek segítségével alkalmazásokat készíthetünk az Apple összes platformjára. Ingyenes letölthető az Apple Store-ból, ezzel viszont következik az a tulajdonsága is, hogy csak OS X-et futtató gépekre tölthető le.

A fejlesztői környezet a teljes munkafolyamatban segítséget nyújt, az alkalmazás készítésétől kezdve tesztelésig és az optimalizálásig. Továbbá, az alkalmazásunk AppStore-ba feltöltési folyamata is innen indul.

Az iPhone-ra történő fejlesztésnél nagy segítséget nyújt a beépített szimulátor, amellyel az összes típusú iPhone-on kipróbálhatjuk alkalmazásunk működését és tesztelhetjük a felhasználói felület megfelelő megjelenítését. Ez azért is nagy segítség, mert nincs szükség arra, hogy megvegyünk minden készüléket a teszteléshez, hanem a szimulátoron keresztül hozzá férhetünk mindegyikhez. Ez alapvetően hasznos az alkalmazásunk működésének tesztelésére, de a számítógépet használja mint fő processzort, így teljesítménykülönbségek fordulhatnak elő, mert lehetséges, hogy amely program a szimulátoron lefutott, valós készüléken egyáltalán nem fut le, vagy nagyon lassan. Viszont, olyan szempontból előnyös ez a tulajdonsága, hogy az alkalmazást ezáltal gyorsabban vizsgálhatjuk meg. Illetve, nem csak a kijelző méretét választhatjuk ki, hanem az iOS verziókat is, ez fontos lehet olyan alkalmazásoknál, melyek több iOS verziót is támogatnak (mint például az általam készített, amely iOS 9 – 12-es verziót támogat). Tesztelésnél itt előfordulhatnak kisebb eltérések, melyek akár az alkalmazás rovására is mehetnek, így célszerű ezeket is végignézni. Valós eszköz és szimulátor használata közben is lehetőségünk van debuggolás-ra törés pontok berakásával, így tesztelhetjük, hogy egyes gombnyomásokra meghívódik-e a megfelelő függvény az előre eltervezett értékekkel.

A legfontosabb eszköz, ahol a kódírás kezdődik a forráskód-szerkesztő. Támogatott nyelvei közé tartozik az Objective-C, a Swift, C, C++, Java, Python, Ruby és egyebek. Ezáltal nem csak OS X-re fejlesztéshez használható, hanem a könnyen kezelhetőség miatt a fejlesztők gyakran használják, akár nem Apple termékre fejlesztéshez is. Beépített funkcióival gyorsabb fejlesztést tesz lehetővé. Ezek közé tartozik a szintaxis színezés, a beszédes figyelmeztető és hiba üzenetek. Az utóbbi esetében nem is engedi kódunkat lefordítani. Továbbá előnyei még, a gyorsan és könnyen használható kód kiegészítés és navigáció a kódon belüli hivatkozások között.

A hozzá tartozó Assistant Editorral ketté oszthatjuk képernyőnket. Ennek használatával az Xcode automatikusan megjeleníti fájlunk mellett a hozzá tartozó legrelevánsabb fájlt, de akár mi is kiválaszthatjuk a fájlokat. Alkalmazás készítés közben ez nagyon hasznos lehet, én is többször használtam a fejlesztés során. Az egyes felhasználói felületek vizuális megjelenése mellé berakható a hozzá tartozó Swift kód fájl, így egyes objektumokat könnyebben szerkeszthetünk. Debug módban is használhatjuk, a program képernyőjének három részre osztásával. Itt helyezkedik el a debug navigátor, ahol láthatjuk a futó alkalmazás aktuális állapotát (CPU használat, memória, stb.). A másik két rész a forráskódból és a debug területből áll. A legutóbbinál vizsgálhatjuk meg egyes változókra és objektumokra lebontva az alkalmazásunkat. Itt található a snapshot biztonsági funkció is, amellyel egy pillanatképet készíthetünk az alkalmazásunk aktuális állapotáról. Ezt akkor érdemes használni, ha jelentős változásokat akarunk véghez vinni az alkalmazásunkban. Ha esetleg bármi nem úgy alakul, ahogy elterveztük, akkor erre az állapotra bármikor visszatérhetünk.

Másik jelentős eszköze az Xcode-nak az Interface Builder, amely kezelői felület tervezéséhez nyújt segítséget. Ezzel akár egy primitív alkalmazást is elkészíthetünk, egy sor kód írása nélkül. Két fajta fájl készíthető vele, az egyik a XIB fájl (saját sémájú XML fájl) a másik a Storyboard (egy alkalmazás kezelői felületének vizuális reprezentációja). A Storyboard-ban lehetőségünk van egyes képernyőket megtervezni. Beépített elemeket adhatunk hozzájuk, mint például táblázat nézet, kép nézet, szöveg vagy akár gombokat. Ezek a UIView leszármazottai és a forráskódban Outlet-ekként hivatkozhatunk rájuk, de akár az Interface Builderben is lehetőségünk van kinézetük módosítására. Megadhatjuk a szövegek típusát, betűméretét, a TableView nézetnek megadhatjuk akár a háttér színét vagy a gombok által megjelenített ikonok képét. Az egyes képernyők közötti váltásra Segue-ekkel van lehetőség. Ezeknek az átmeneteknek id-kat adhatunk, melyekre később a forráskódban hivatkozhatunk.

Képernyő tervezéshez használatos eszköz még a beépített Auto Layout, melynek használatával Constrainteket (kényszereket) definiálhatunk egyes grafikus elemekhez. Ennek segítségével meghatározhatjuk, hogy egyes elemek, hol helyezkedjenek el a képernyőn, illetve egymáshoz viszonyított elhelyezkedésüket is.

Ezen felüli szolgáltatás még a Unit tesztelő eszköz. A unit szó, arra hivatkozik, hogy a legkisebb egység, amit tudunk vagy akarunk tesztelni. Az Xcode legenerálja a tesztosztályokat setUp és tearDown metódusokkal, melyeknek a törzsében írhatjuk meg a szükséges kódot. Majd írhatunk teszteseteket kódunk megfelelő működésének ellenőrzésére, melyeknek a test szóval kell mindig kezdődniük. Itt használhatjuk több más lehetőség mellett az XCTAssertEqual függvényt, amellyel kiértékelhetjük, hogy a teszt megfelelően futott-e le. Az első kapott paramétert összehasonlítja a másodikkal és ha megegyeznek, akkor a teszt kimenetele sikeres.

Ezekkel a funkciókkal mondhatjuk az XCode-ot kiemelkedően könnyen kezelhető és sok hasznos, fejlesztést gyorsító eszközzel ellátott fejlesztői környezetnek. Sokan emiatt is érdeklődnek az iOS fejlesztés iránt, mert a beépített eszközöknek köszönhetően kevés tudással is létre lehet hozni működő alkalmazásokat.

## Architektúrális minták

A programozásban is találkozhatunk mintákkal, melyek jelentősen megkönnyítik a fejlesztők dolgát. Ezeket három csoportba sorolhatjuk, melyek a menedzsmenti, architektúrális és tervezési minták. A menedzsmenti minták az emberek és folyamatok szintjéhez tartoznak. A tervezési minták használatával minimalizálhatjuk munkánkban a kód duplikációt és általános megoldásokat készíthetünk többször előforduló problémák megoldására az osztályok, objektumok szintjén. Az architektúrális minták viszont a rétegek és erőforrások szintjén helyezkednek el. Minden mobil alkalmazásnak megvan a saját architektúrája. Nagyon sok eltérő architektúrális mintával találkozhatunk, de a négy leggyakrabban használt: az MVC, MVP, MVVM és a VIPER. Ezek közül kettőt mutatnék be, melyeket mérlegeltem a projektem elkészítésekor.

### MVC

A Modell-Nézet-Vezérlő (Model-View-Controller) programozási mintát használjuk alapvetően iOS fejlesztésre. A programozási mintát 1970-ben mutatták be és az összes többi mintának az őse. Egyike az első objektum-orientált programozási mintának.

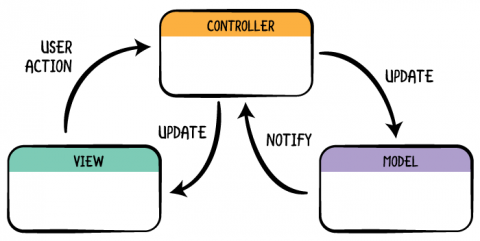
Az elnevezés az objektumok kategóriájára vonatkozik, mivel eszerint csoportosítjuk őket elsődleges szerepük alapján három kategóriába. Ezáltal a kódot szerepek alapján is szétválaszthatjuk, az átláthatóság érdekében. A három szerep részletesebb ismertetése:

A Model tárolja és kezeli az alkalmazás által használt adatokat és az azokon elvégezhető műveleteket. Nincs külön adatelérési réteg, hanem az is a Model részévé válik. Ilyen lehet például egy adatbázis egy vállalat dolgozóiról, ahol tároljuk telefonszámukat, nevüket és email címüket.

Nézetnek (View-nak) olyan objektumokat nevezünk, melyek a Model-ben tárolt adatok vizuális megjelenítéséért felelősek. Az ő felelőssége megjeleníteni az elemeket a felhasználó felületen, melyeket a rendszer használója lát. Továbbá, ide kapcsolódnak az adatokon végzett felhasználó interakciók is. Egy Model-hez több nézet is tartozhat, különböző funkciók megjelenítésére.

A Vezérlő (Controller) a View és a Model közötti összekötő. A Model-ben lévő adatokat továbbítja a View-nak megjelenítésre. Továbbá, kezeli a felhasználói eseményeket és ezekhez megfelelő működést rendel és módosíthatja a Model-ben szereplő adatokat.

Az alábbi képen látható az MVC működése:



**2.4.0.3.1. ábra MVC architektúra modellje**

Felhasználói interakció érkezik a Nézettől (View) a Vezérlőhöz (Controller), például egy gombnyomás. Majd ő átveszi ezt az eseményt a View-tól és feldolgozza. A Controller ezek alapján kapcsolatot létesít a Model-el és frissíti egyes elemeit. A View a Model alapján változtatja felhasználói felületét, a Model-ből nyert adatokkal. Viszont a Model-nek nincs tudomása a View-ról, ő csak a Controller-t érzékeli. Majd újabb eseményre várunk, mely újrakezdi a folyamatot.

Fordított esetben a Model értesíti a Controller-t az adatok változásáról és ő ez alapján frissíti az elemek megjelenítését a hozzá tartozó View-ban.

Látható, hogy a három szerepkör teljesen elkülöníthető egymástól, így az objektumok csoportokba sorolása könnyen megoldható. A View és a Model szétválasztásának előnye, hogy csökken a szerkezet bonyolultsága és ezáltal könnyebb kód újra felhasználhatóságot tesz lehetővé.

### VIPER

A VIPER (View, Interactor, Presenter, Entity és Router) egyre szélesebb körben elterjedt modernizált programozási minta. Hasonlóan az MVC programozási mintához, itt is moduláris megközelítéssel találkozhatunk. Ez bővebben azt jelenti, hogy egy funkcióhoz egy modult rendelünk hozzá. Minden egyes modulhoz öt (esetleg négy) elkülönülő osztály tartozik, ezzel kiosztva a szerepköröket.

Az osztályok részletes ismertetése:

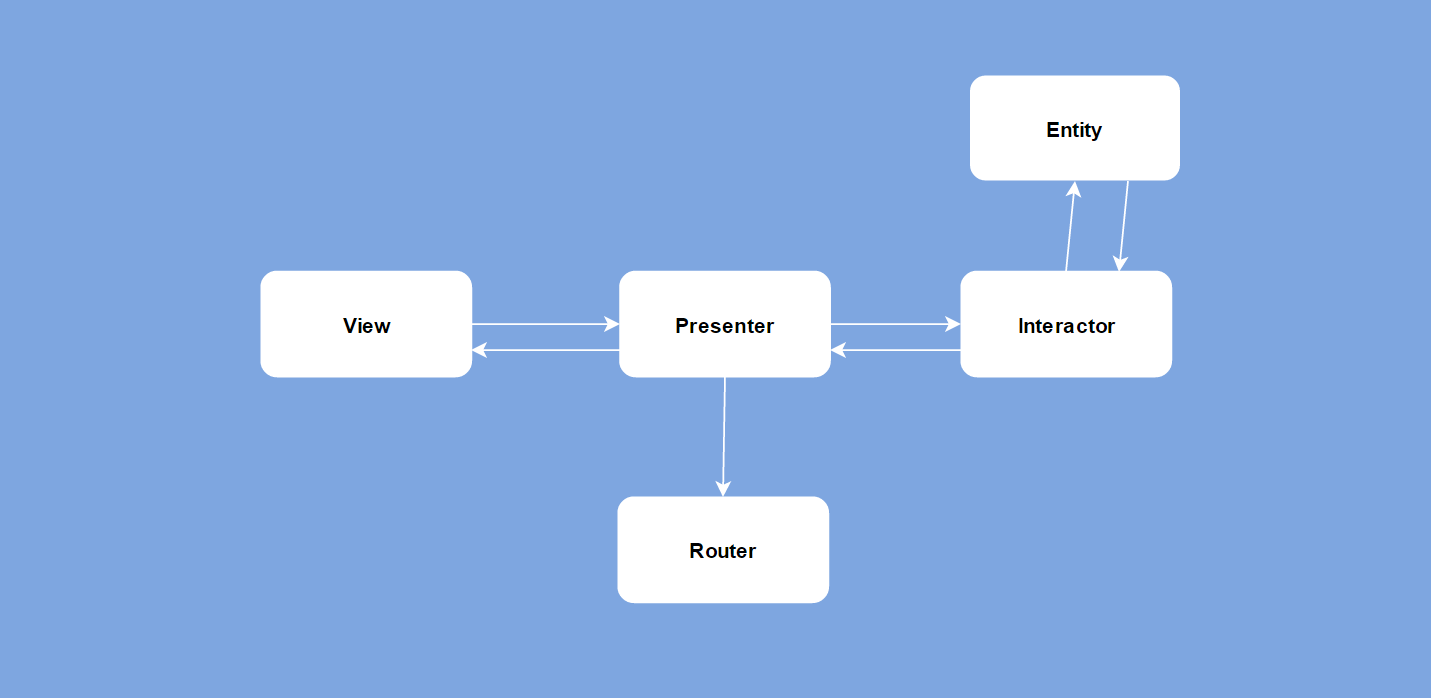
A View (Nézet) a felhasználói interfészért felelős. Az ő feladata megjeleníteni az kódot a felhasználó számára, illetve különböző interakcióit érzékelni. Majd ezekről a történésekről értesíti a Presenter-t.

A Presenter a View-tól kapott információknak megfelelően cselekszik. Egy olyan osztály, melyen keresztül kommunikálunk a többi komponenssel. Ő frissíti a View elemeinek megjelenítését, lekéri az adatokat az Interactor-tól vagy kommunikál a Router-rel (például: hálózati hívások - Interactor).

Az Interactor feladata az üzleti logika, bármi, ami számolás hozzá kapcsolódik. Ide tartozik az API hívások indítása vagy akár az adatbázis lekérése is.

A Router feladata az útvonalak elkészítése. A Presentertől kapott információk alapján jeleníti meg az egyes képernyő nézeteket. Az ő feladata a képernyők közötti navigáció.

Az Entity (entitás) egyszerű modell osztályokat tartalmaz, melyeket az Interactor használ. Itt tároljuk az alkalmazás adatait. A Presenter-ek Entity osztályokat kapnak az Interactor-tól, amelyeket átalakítanak vagy módosítás nélkül továbbítanak a View felé.



**2.4.4.1. ábra VIPER architektúra modellje**

### Viper vs MVC

A VIPER az alkalmazások logikai struktúráját több szerepkörre osztja. Ezáltal fellazítja a szoros függőségeket és megnöveli a tesztelési lehetőséget minden szinten. Ennek feltétele az is, hogy az egyes komponensek interfészek mögött helyezkedjenek el. A unit tesztek sokkal egyszerűbbé válnak, minden funkcionális követelményt tesztelni tudunk. Ezzel az előnyével emelkedik ki a másik három leggyakrabban használt programozási minta közül. Tesztelés szempontjából nézve az MVC mintában csak a Modellt tudjuk vizsgálni. A View és a Controller szoros kapcsolata miatt egyáltalán nem tesztelhetőek.

### Konklúzió

Alepvetően a VIPER is MVC alapokon nyugszik, csak a Controllert két részre osztjuk fel: Presenter és Interactor. A View ugyanaz maradt, de a Modell neve Entity-re változott. Az új rész a Router.

Tesztelés szempontjából a VIPER nagyobb előnyökkel rendelkezik, de valójában a piacon kevesen szánnak sok időt alkalmazásuk tesztelésére és akik szánnak is, azok sem biztos, hogy megfelelően csinálják. A VIPER-nél minden felelősség külön kisebb osztályokban helyezkedik el, ami néha nagy segítséget nyújt a tesztelésben, de előfordulhat az is, hogy megnehezíti. Unit tesztek írása kisebb osztályokhoz nagyon könnyű, de valójában van, hogy ezek a tesztek tulajdonképpen nem is tesztelnek semmit. Ilyen lehet a Presenter metódusainak tesztelése, ami lényegében csak egy közvetítő a View és a többi komponens között. Ennek mellékhatásaként az összes ilyen tesztet is szükséges frissíteni minden egyes változtatásnál.

Továbbá, a nagyon sok kis extra osztály és protokoll nem biztos, hogy segíti a kód átlátását. Ez azt is eredményezni, hogy akár egy gomb megváltoztatása a kódban is legalább három osztály és négy protokoll megváltoztatását jelenti. Ez nem csak fejlesztés közben lehet nagy hátrány, de teljesítményi problémákhoz is vezethet. Egy alkalmazás elindulási ideje nagy mértékben függ az osztályok számától. Egy kisebb alkalmazásnál ez nem jelent hátrányt, ahol 50-100 osztály van, de egy nagyobb, akár 10 000 darab körüli osztálynál (pl.: Facebook) már felvethet problémákat.

Összefoglalva, a VIPER programozási minta nagyon sok plusz munkát jelenthet egy alkalmazásnál. Fontos átgondolni, hogy megéri-e. Ha hosszú élettartalmat jósolunk az alkalmazásunknak, akkor érdemes mérlegelni. Továbbá, fontos szempont, hogy a specifikációnk megfelelően ki van-e dolgozva, mivel egy kisebb változtatás is nagyon sok átírást jelenthet a kódban. Illetve, a tesztelési előnyre visszatérve, pedig érdemes átgondolni, hogy valóban akarunk-e ennyi mindent tesztelni az applikációnkban, a többlet fejlesztési idő árán is.

A választásom végül az MVC mintára esetett, mert az alkalmazásom mérete nem feltétlen indokolja a VIPER használatát.

## 2.5. Verziókezelés

Fejlesztés közben nagy segítséget nyújt a verziókezelés használata. Ehhez segítségül a GitHub-ot hívtam, mely kódok tárolására alkalmas platform. A [www.github.com-on](http://www.github.com-on) bejelentkezés után lehetőségünk van feltölteni alkalmazásunk forráskódját, majd ennek segítségével verziókezelést alkalmazni rá, illetve megosztani másokkal.

Verziókezelés szempontjából van egy fő ág (master) amelyre először feltöltjük alkalmazásunkat, majd a későbbi változtatásokat külön ágakra (branch) bonthatjuk. Minden egyes külön funkcióhoz érdemes külön ágat csinálni, ezzel is könnyebben átláthatóvá téve az alkalmazást. Az ágaknál egy rész elkészülése után érdemes commit-olnunk, melyben megjegyzést írhatunk a feltöltendő kódhoz. Ezt érdemes minél lényegre törőbben megfogalmazni, hogy csapat esetén a munkatársaink számára is átlátható legyen. A commit elkészülte után a push paranccsal tölthetjük fel az águnkra a forráskódot. Majd egyes ágak elkészülése után mergel-hetjük a fő ággal őket, ez lényegében egy összefésülést jelent. Fontos, hogy csak működő (hiba nélküli) részeket töltsünk fel az águnkra és a fő ágra is.

A verziókezelés használatával bármikor visszavonhatunk hibával járó tevékenységeket. Így érdemes nagyobb változtatások előtt mindig feltölteni az águnkra az eddig működő munkánkat, hogy error esetén visszatudjunk térni a jó verzióra.

Ezen funkciói által a verziókezelést nem csak nagy csapatokban érdemes használni, hanem akár egyedüli fejlesztés esetén is. Így én is alkalmaztam az alkalmazásom írása közben, mert bármely hiba esetén vissza tudtam térni egy megfelelően működő állapotra. Ezáltal könnyebben mertem nagy változtatásokat elvégezni.

# Feladat ismertetése

## Specifikáció

TODO: alkalmazással szemben támasztott követelményeket , milyen funkciókat tud „majd” az alkalmazás, esetleg van-e hasonló app a piacon, ha van akkor azokhoz képest mivel tud többet stb.

# Technológiák ismertetése

## CocoaPods

A CocoaPods egy alkalmazás szintű függőségi menedzser, mely Swift és Objective-C nyelveken írt alkalmazásokhoz nyújt segítséget.

Lényegében külső könyvtárak beépítését teszi lehetővé alkalmazásunkba, a forrás megőrzésével. Ezáltal mások által előre elkészített könyvárakat/komponenseket integrálhatunk alkalmazásunkba. Általában olyan könyvtárak kerülnek fel ide, melyek általános megoldásokat tartalmaznak és könnyen beépíthetők bármely az előző felsorolt nyelvekben írott applikációkba.

A <https://cocoapods.org/> keresőjében rákereshetünk az egyes könyvtárakra, például naptár, üzenetek vagy egyéb beépíthető részre. Nincs szükség a könyvtárak forráskódjának letöltésére, mindössze egy Podfile-t kell létrehoznunk projektünkben, majd itt felvenni a beépíteni kívánt Pod-okat (könyvtárakat).

Podfile-t létrehozni terminál ablakból tudunk, a projektünk mappájába navigálás után a ,,pod init” parancs kiadásával. Ezután a mappánkba létrejött a Podfile, melyet egy szövegszerkesztővel tudunk módosítani. Ide a pod kulcsszó után írhatjuk egyes idézőjelek közé a kívánt könyvtárakat. Illetve, meghatározhatjuk a platform szó után a legalacsonyabb szintű iOS verziót, melyen az alkalmazásunk még megfelelően működik. Ezután a ,,pod install” parancs kiadásával telepíthetjük alkalmazásunkba a megadott külső könyvtárakat.

Az alábbi képen látható az alkalmazásom Podfile-ja. Több külső könyvtárat is felhasználtam, melyek megkönnyítették a fejlesztői munkát. Ilyen például a Programoknál található naptár rész (FSCalendar), melynél a napokat tudjuk kiválasztani. Illetve, fontosabb még az Alamofire, mely a szerverrel történő kommunikációban volt segítségemre.

*# Uncomment the next line to define a global platform for your project*

platform :ios, '9.0'

target 'EventApp' do

*# Comment the next line if you're not using Swift and don't want to use dynamic frameworks*

use\_frameworks!

*# Pods for EventApp*

*#pod 'SwiftLint'*

pod 'MessageKit', :path => '../MessageKit'

pod 'MessageInputBar', :git => 'https://github.com/MessageKit/MessageInputBar.git', :branch => 'master'

pod 'FSCalendar'

pod 'Alamofire'

pod 'SwiftyJSON'

pod 'Kingfisher'

pod 'SwiftKeychainWrapper'

end

**4.1.1 FSCalendar**

Az FSCalendar egy nyilvános könyvtár, melyet a készítő a cocoapods oldalán szabad felhasználásra bocsátott. Az oldalon megtalálhatjuk a hozzá tartozó github repository-t is, melyen elérhető a teljes forráskódja. Továbbá, találhatunk még itt példa projekteket is a megfelelő használathoz. A naptár Objective-C-vel és Swift-tel egyaránt kompatibilis. Mindössze annyit kell tennünk a projektünkben, hogy felveszünk egy View-t (üres nézetet) a Stroyboard-on és a hozzá tartozó View Controller forráskódjába felvesszük ezt a View-t és a típusának beállítjuk az FSCalendar-t. Ezzel a Storyboard-ban nem válik láthatóvá, mert ő nem tudja automatikusan megjeleníteni a külső könyvtárakat, de szimuláláskor már láthatjuk a naptárat és lehetőségünk van a napok között ugrálni.

**4.1.2 Alamofire**

Az Alamofire egy HTTP hálózati könyvtár Swift nyelven. Az Alamofire segítségével számos hálózati kérést egyszerűbben végezhetünk el. Főbb funkciói közé tartoznak a láncolható kérés/válasz metódusok, a JSON paraméter és válasz szerializáció és az authentikáció. Később a szerverrel történő kommunikáció fejezetében részletezem használatát.

**4.1.3 Kingfisher**

A Kingfisher hívtam segítségül a szervertől kapott képekhez tartozó URL-ek feloldására. Itt a tárolom a szervertől visszakapott String értéket, majd átalakítom URL típusúvá. Ezután a változtatni kívánt image után írok egy .kf-et, ami a Kingfisher-t jelöli és ezután hívom meg a setImage(with: url) metódust. Ennek használatával könnyen alkalmazhatunk URL-eket kép megadásához alkalmazásunkban.

**guard** **let** iconString = event[0].iconURL **else** { **return** }

**guard** **let** url = URL(string: iconString) **else** { **return** }

iconImage.kf.setImage(with: url)

## JHipster

Az alkalmazásom a kliens funkciót tölti be, mivel a szerverrel kommunikál és a tőle kapott információkat jeleníti meg grafikus felületén. A JHipster segítségemre volt az alkalmazásomhoz tartozó szerver elkészítésében. A szerverrel történő kapcsolat felépítéséről a későbbi részekben lesz szó, ebben a fejezetben a szerver elkészítésének folyamatát mutatom be.

A JHipster egy fejlesztői platform, mely segítséget nyújt Angular/Ract webes alkalmazások generálásához és fejlesztéséhez.

A használatával lehetőségünk van testreszabott backend generálására applikációnkhoz. Szerver oldalon Java kóddal találkozhatunk, a frontend-en pedig választhatunk Angular vagy React webes alkalmazás platformból, melyek Bootstrap-et használnak, ezzel reszponzívvá téve az alkalmazást bármely eszközre.

A backend generálásához létre kell hoznunk egy mappát, ahova szeretnénk letölteni a szükséges fájlokat. Ezután egy terminál ablakban be kell lépnünk ebbe a mappába, majd kiadni az ,, npm install -g generator-jhipster” parancsot, mely telepíti a fájlokat. A parancs kiadása után a ,,jhipster” parancs kiadásával a terminál ablakban a JHipster feltesz kérdéseket azzal kapcsolatban, hogy milyen backend-re van szükségünk. Itt választhatjuk ki, hogy milyen típusú authentikációt szeretnénk használni. A lehetőségek között szerepel a JWT, ami egy JSON Web token alapú authentikáció. Illetve, az OAuth 2.0 / OIDC authentikáció, mely az alkalmazáson kívül oldja meg az authentikációt. Ez biztonságosabb mint a JWT token használata, de OpenID Connect szerverre van szüksége, így bonyolultabb. Továbbá, használhatunk még HTTP Session authentikációt, mely egy klasszikus session-alapú authentikáció. Illetve, használhatjuk még a JHipster UAA szerver authentikációját, de ezt külön le kell generálnunk a projektünkhöz. A lehetőségek közül a default opciót választottam, ami a JWT token, a könnyebb fejlesztés miatt, illetve a legtöbb felhasználó ezt használja.

Főbb kérdésként szerepel még, hogy milyen adatbázist szeretnénk használni. Típusok közül az SQL, MongoDB, Cassandra, Couchbase és adatbázis nélküli lehetőség szerepel. Itt az SQL típust választottam, mivel az volt számomra ismert egyetemi tanulmányaimnak köszönhetően.

Illetve, a fejlesztői adatbázis kiválasztásánál a H2 adatbázist választottam, melyből szintén két lehetőség közül lehet választani:

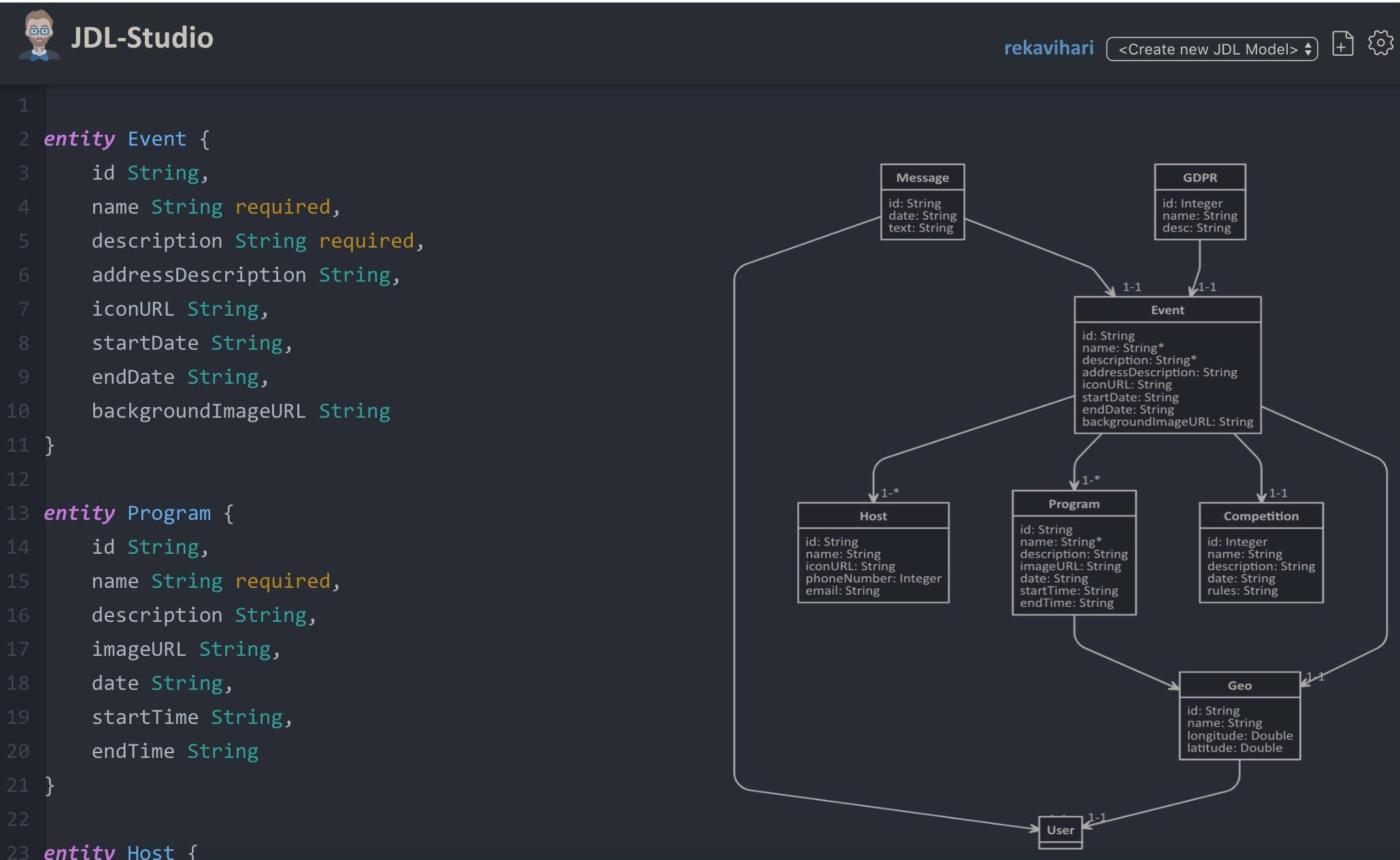
* memory-running: mely a szerver újraindításánál törli az adatokat,
* a disk based, ami tárolja a disk-en az adatok, így újraindítás után sem vesznek el.

Itt az utóbbit választottam.

Keretrendszernek pedig Angular-t választottam, mert architektúrája az MVC minta szerint készült, melyet az alkalmazásom készítésénél választottam.

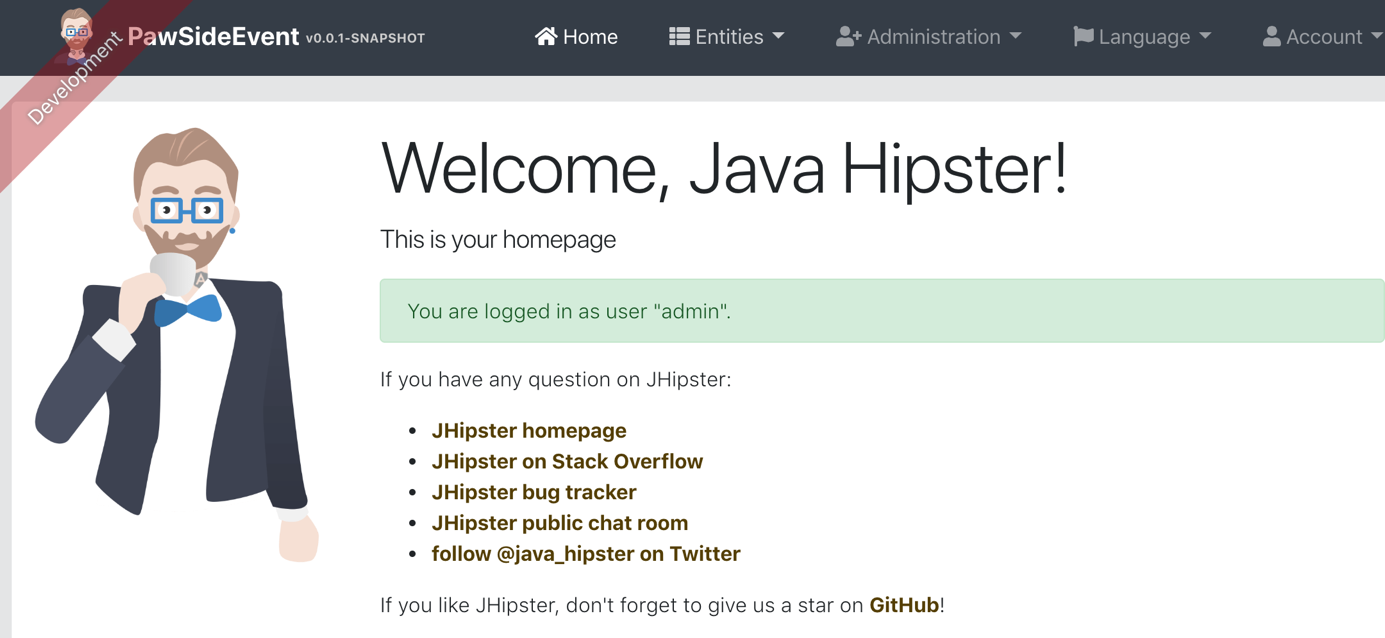
A generálás után szükséges a JHipster oldalán elkészíteni az alkalmazásunkhoz tartozó osztálydiagramot, melynek importálásával a projektbe a JHipster le tudja generálni az egyes osztályokat és a hozzájuk tartozó szükséges fájlokat. Továbbá, az adatbázis táblákat (a sémát) is legenerálja.

Az osztálydiagramot nem nekünk kell megrajzolnunk, csak fel kell vennünk az entitásokat és a köztük lévő kapcsolatokat, melyből a modell generátor (https://start.jhipster.tech/jdl-studio/) elkészíti nekünk.



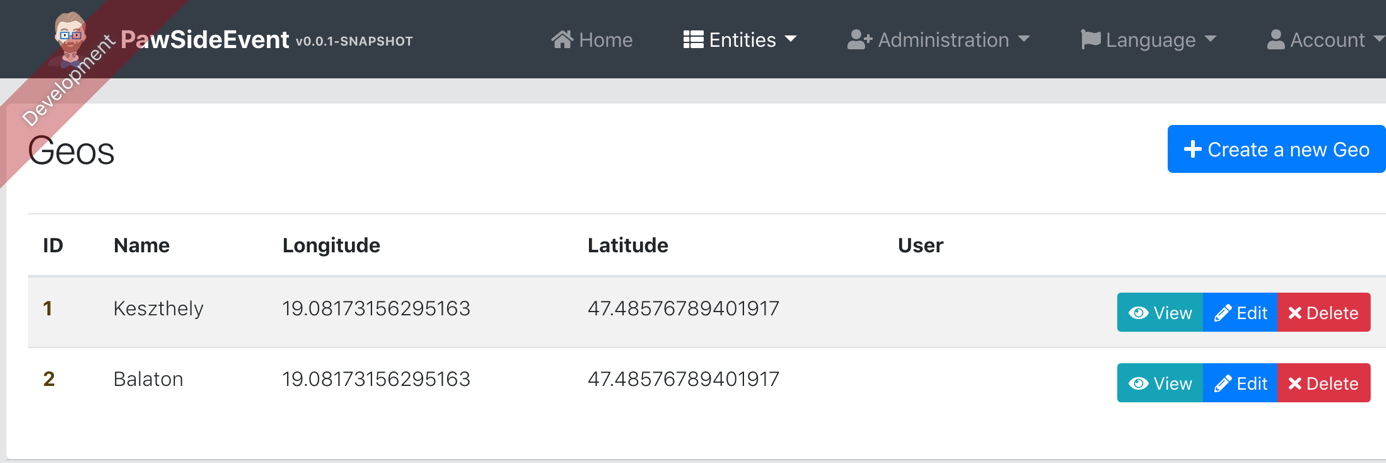
Eztuán egy jdl fájlban letölthetjük elkészített entitásainkat és kapcsolatainkat, majd a projektünk mappájában kiadott ,,jhipster import-jdl ~/Documents/jhipster-jdl.jh” utasítással legenerálja a fájlokat.

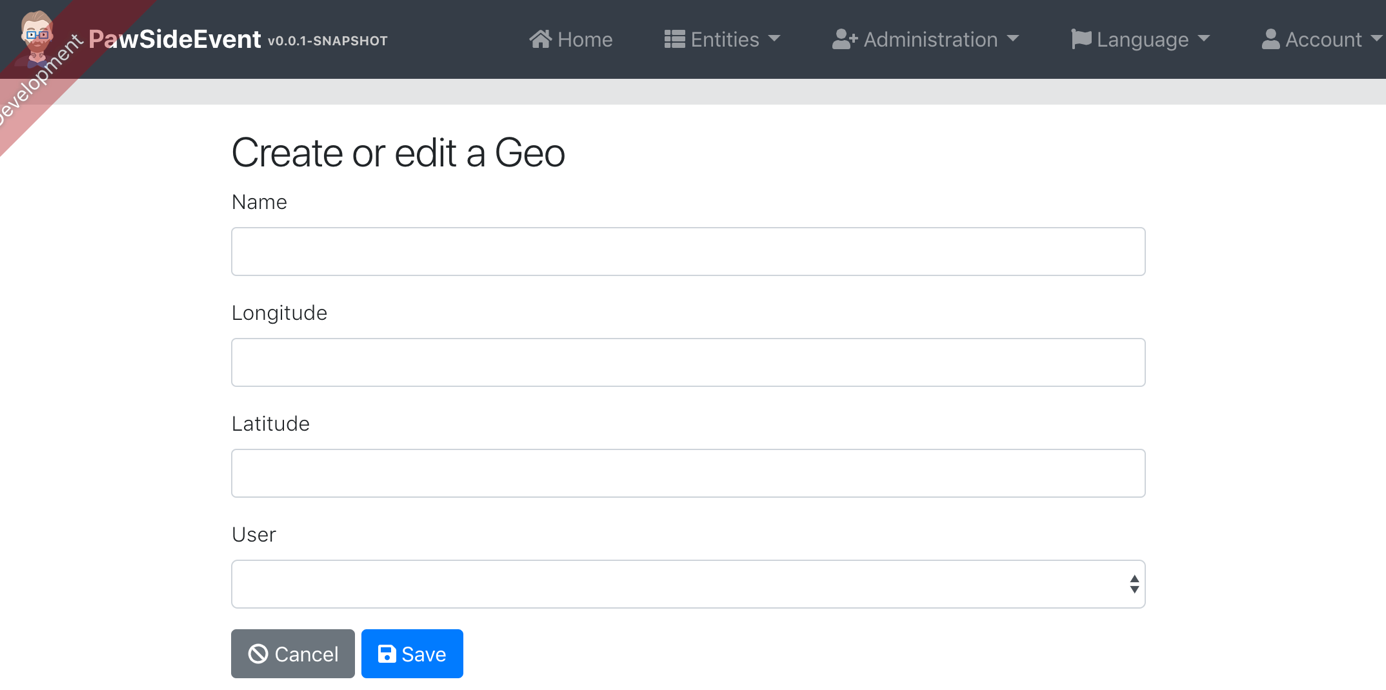
A következő lépés, hogy egy fejlesztői környezetben, melyből én az IntelliJ IDEA-t használtam, elindíthatjuk a backend-et, melyet alapvetően a <http://localhost:8080-on> érhetünk el, de ha ezt a lokális szervert már másra használjuk, akkor a projekt yo-rc.json fájljában a serverPort-ot átírva máshova is futtathatjuk alkalmazásunkat.



Bejelentkezni alapvetően az admin-admin vagy user-user párossal lehet, ezt publikus szerver esetén érdemes megváltoztatni. Admin jogosultsággal kezelhetjük a felhasználókat is, User-ként viszont csak az entitásokat változtathatjuk.

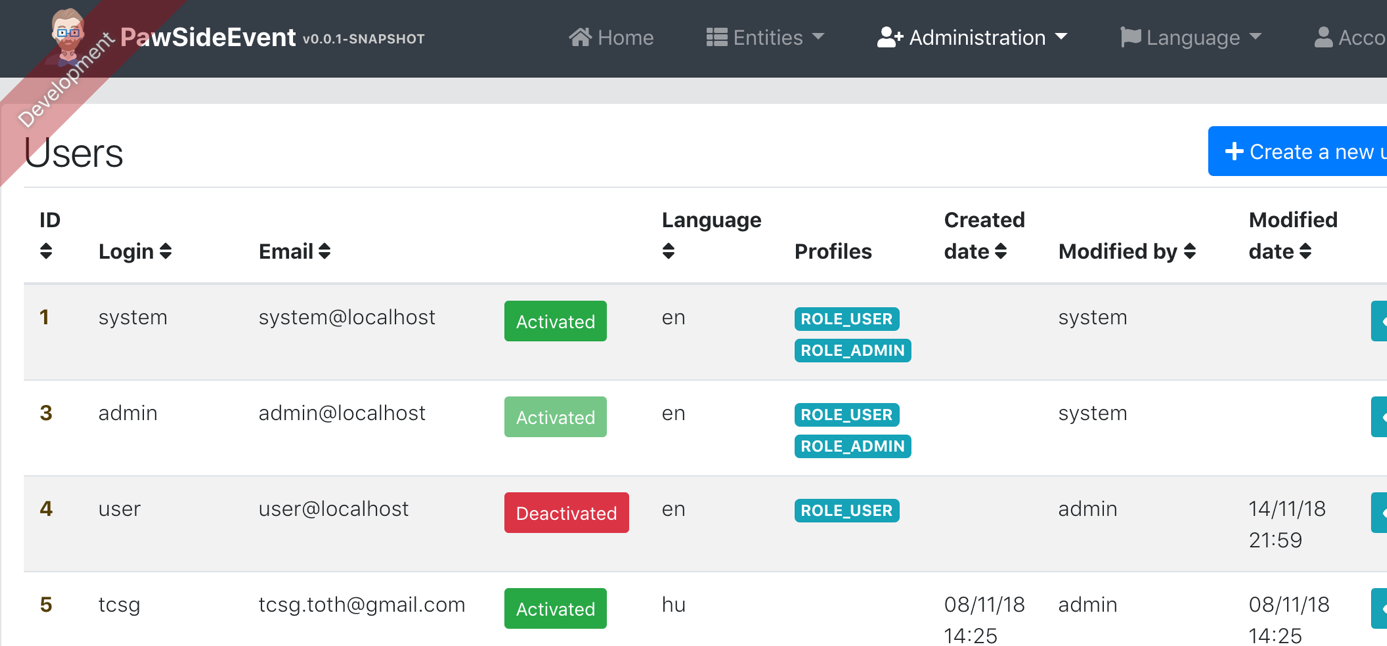
Az entitások felületén programozói tudás nélkül is könnyen szerkeszthetjük a már felvett entitásokat, de újakat is adhatunk hozzájuk. A képen a Geos entitáshoz tartozó adatokat láthatjuk, ezek az ID, Name (név), Longitude (hosszúság), Latitude (szélesség) és az esetlegesen hozzá tartozó User (felhasználó).



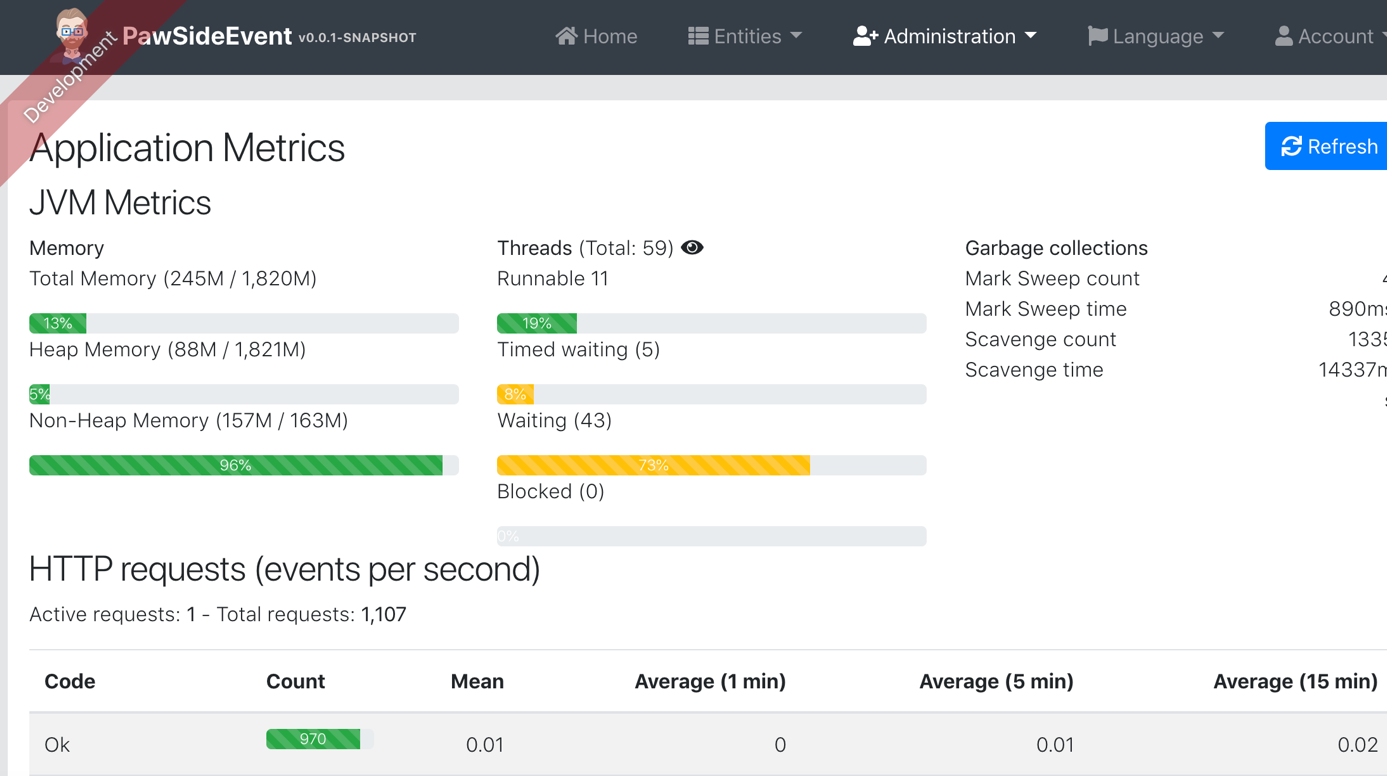


Az Administration menü lenyitásával több opció közül választhatunk.

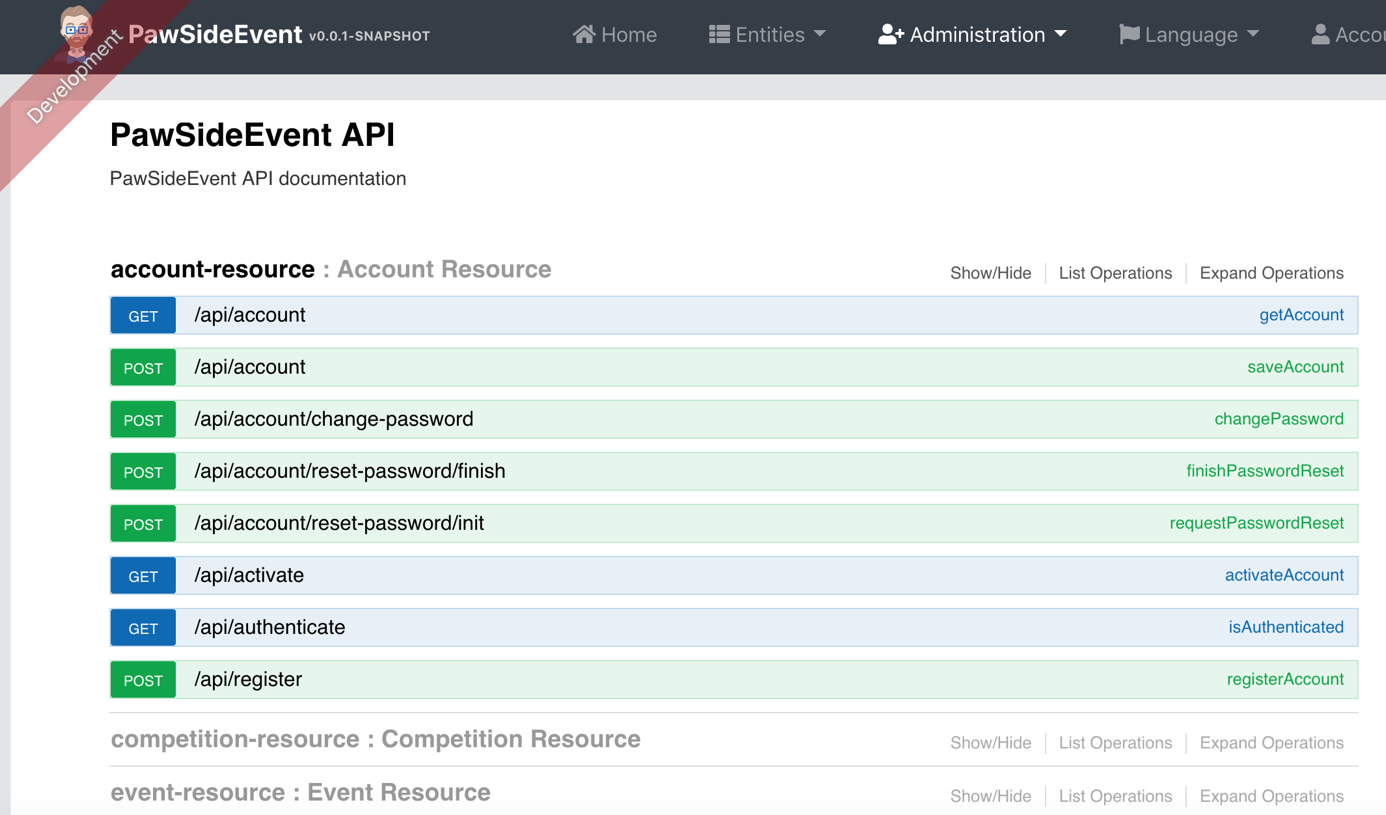
Lehetőségünk van a User-ek kezelésére, melynél beállíthatjuk adataikat vagy akár jogosultságaikat. Itt tudjuk aktiválni az applikáció által beregisztrált felhasználókat. Aki nincs aktiválva, az nem tud bejelentkezni. Akár itt is tudunk új felhasználókat beregisztrálni a ,,Create a new user” gomb megnyomásával egy hasonló felületre kerülünk, mint az entitások hozzáadásánál.



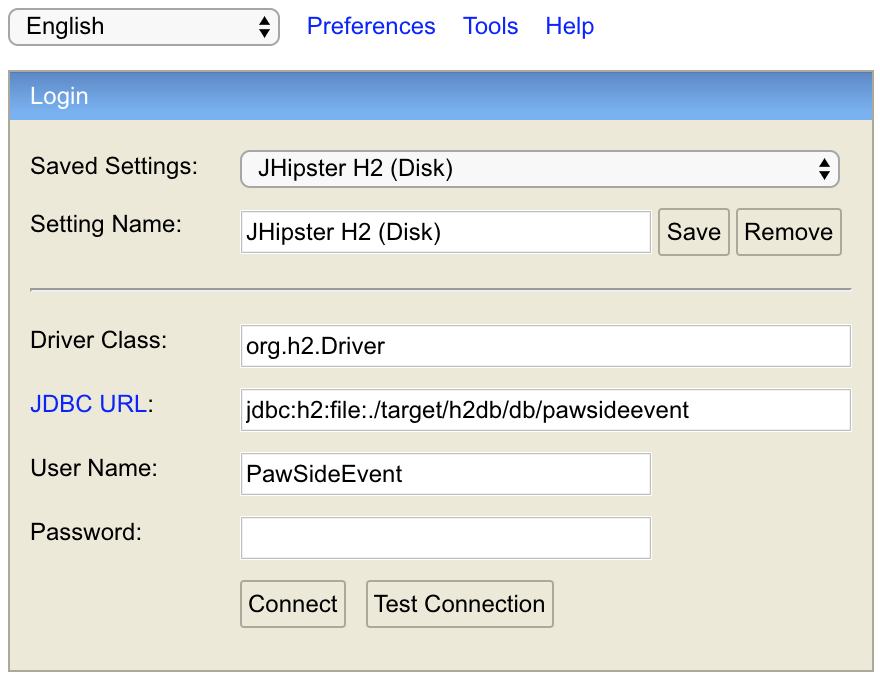
A Metrics fülön láthatjuk alkalmazásunk memória használatát, a HTTP kéréseket és egyéb statisztikákat.



Fontosabb kiválasztható fül még az API, ahol a szerverrel történő kommunikációhoz szükséges végpontokat találhatjuk. Illetve, lehetőségünk van egyes bemenetekre tesztelni is a szerver válaszát. Láthatjuk, hogy az egyes kéréseknél milyen paramétereket kell felküldenünk a szervernek, hogy a várt kimenetet kapjuk. Továbbá, az egyes adatlekérésekhez szükséges információkat is itt találjuk meg.



Innen érjük el az adatbázist is, melynél, ha rákattintunk a Database fülre, akkor átkerülünk a H2 adatbázisunk bejelentkező felületére. A bejelentkezésnél csak a felhasználónév megadása szükséges, melyet automatikusan kitölt az alkalmazás. Jelszót a belépés után van lehetőségünk beállítani, ha szeretnénk.



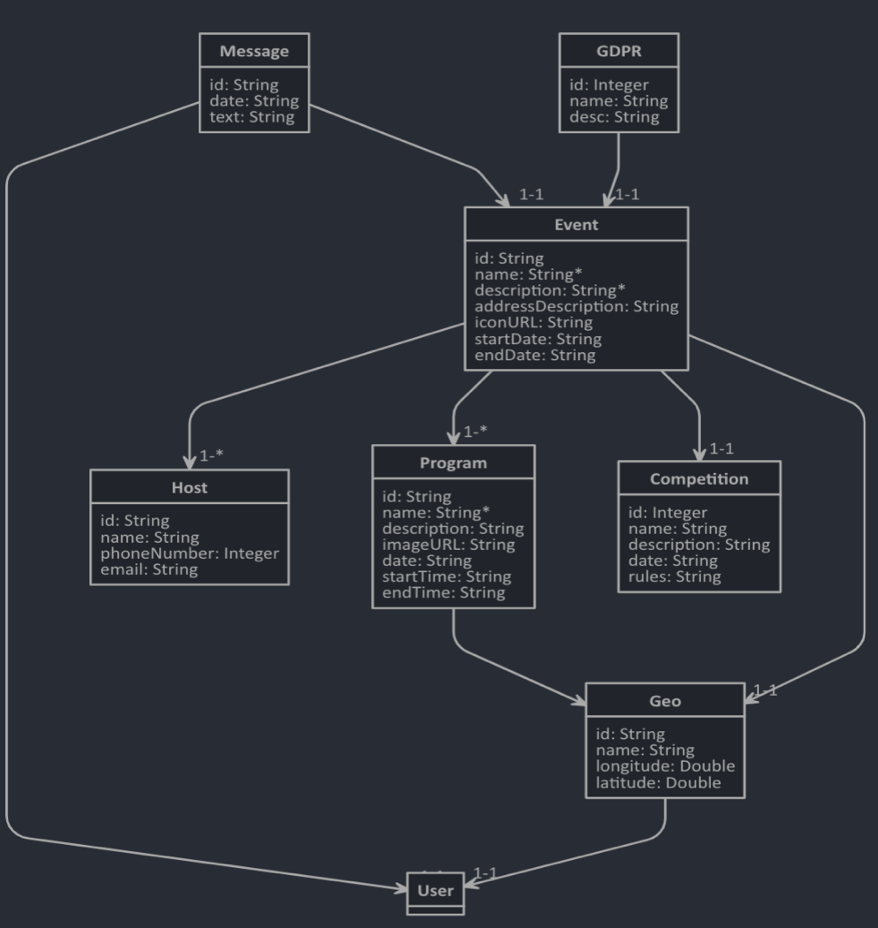
, mint az előbbiekben említettem,

# Tervezés

## Adatbázis

Az adatbázisom tervezésénél az elsődleges feladat az entitások megtervezése volt. A JHipsterrel nehézségekbe is ütköztem, mert a User entitás alapértelmezetten szerepel az adatbázisban, így nem adható hozzá. Ez azt is jelentette, hogy nem vehetőek fel hozzá egyéni attribútumok, hanem a megadottakkal lehet csak dolgozni. Más entitás felvétele a felhasználók számára azért nem tűnt jó megoldásnak, mert authentikáció szempontjából csak az alapértelmezett User entitással lehet dolgozni. Így azt a megoldást találtam kézenfoghatónak és egyszerűnek, hogy a szükséges további attribútumokat új entitásokként kapcsolatokkal adom hozzá a meglévő entitáshoz.

Így alakult ki az alábbi osztálydiagram:



Szerver oldalról bármely entitást szerkeszthetünk, illetve hozzá adhatunk újat. Kliens oldalról viszont csak az alkalmazásban meghatározott entitásokat szerkeszthetjük, mely lehetőségét az entitásoknál említem.

Event entitás: itt tárolom az eseményeket. A hozzá tartozó attribútumok az azonosító, név, leírás, cím, ikon, kezdeti- és befejező dátum. Ezek közül az azonosító és a név kötelezően megadandó, amikor a szerveren szeretnénk felvinni új eseményt.

Host entitás: a szervezők tárolására szolgál, melyekből egy eseményhez több is tartozhat. Nekik is van azonosítójuk, nevük és ezen felül tárolunk hozzájuk telefonszámot és email címet. Ezen attribútumok megadása nem kötelező, így előfordulhat az is, hogy az eseményhez nem tartozik szervező.

Program entitás: az eseményekhez tartozó programok szerepelnek itt. Az entitásnak van azonosítója, neve, leírása, képe, dátuma, illetve kezdeti- és befejező időpontja. A név attribútum megadása kötelező. A programok több-egy kapcsolatban állnak az esemény entitással.

Competition entitás: a neve alapján az eseményekhez tartozó versenyek tárolására szolgál, de ennek hiányában, akár egy részletesebb leírást is készíthetünk vele az eseményről. Van azonosítója, neve, leírása, dátuma és szabályai. De a szabályok helyére akár az eseményhez kapcsolódó fontos tudnivalókat is beírhatjuk.

GDPR entitás: az eseményhez tartozó adatvédelmi szabályzat entitása. A szerveren keresztül feltölthetjük mindig eseményünkhöz az aktuális szabályokat, ezzel is eleget téve a nemrég életbe lépő adattárolás jogi szabályainak. Itt információt adhatunk a felhasználónak, hogyan kérheti adatai törlését, vagy esetlegesen, hogyan kérdezheti le a róla tárolt adatokat. Egy eseményhez egy adatvédelmi szabályzat tartozhat. Attribútumoknak megadhatunk azonosítót, nevet és leírást.

Geo entitás: a felhasználók, az események és a programok lokációjának tárolására szolgáló entitás. Az entitásokkal egy-egy kapcsolatban áll. Az eseményekhez és a programokhoz tartozó helyek szerver oldalon adhatóak meg. A felhasználói helyzetek pedig az alkalmazáson keresztül jutnak fel a szerverre. Tárolok hozzá egy azonosítót, nevet és a hosszúsági- és szélességi koordinátákat

Message entitás: az üzenetek kezelésére szolgál. Egy üzenet egy felhasználóval áll egy-egy kapcsolatban, illetve az eseménnyel több-egy kapcsolatban. Tartozik hozzá egy azonosító, dátum és szöveg. Üzenetek létrehozására az alkalmazáson keresztül is van lehetőség. Itt az Üzenetek menü alján található szövegdobozba írással, majd a Küldés gombra kattintással felküldhetjük üzenetünket a szerverre. De szerver oldalról is hozzá lehet adni új üzeneteket.

## 

## 

## Kommunikáció a szerverrel

Az alkalmazásom használata közben a legtöbb felhasználói interakció szerverhívást kezdeményez, ehhez szükséges volt felépíteni a kapcsolatot a szerverrel.

A szerverrel az alkalmazás JSON segítségével kommunikál. A hívásban küldött paraméterek és a szerver által küldött válaszok is JSON formában jelennek meg.

A JSON objektumok név:érték párokból állnak, ebből a név általában egy karakterlánc (String) érték. A hozzá tartozó érték lehet szöveg, szám, logikai igaz/hamis, objektum, tömb, de akár null érték is. A JSON tömbök értéke szintén lehet nulla vagy egyszerre több objektumot tartalmazhat.

Az alábbi diagramon látható a felépítése:



5.1. ábra JSON objektum felépítése

A szerverrel történő kommunikáció megvalósításához az Alomofire-t használtam, mely technológiát már az előbbi fejezetben bemutattam.

A megvalósításhoz létrehoztam egy NetworkManager osztály, amely felépíti a kapcsolatot a szerverrel. A kérésekben, a megcímzett URL-ek (végpontok) vége változik, így ehhez létrehoztam egy felsorolást Endpoints néven, amely definiálja az egyes entitásokhoz tartozó végpontokat.

**enum** Endpoints: String {

**case** events = "events"

**case** program = "programs"

**case** location = "geos"

**case** message = "messages"

}

A NetworkService osztályban definiáltam az egyes kérésekhez szükséges paramétereket és fejléceket. Az adatok lekérésehez tartozó általános get metódust is itt hoztam létre. Az adatbázis eléréséhez paraméterként a felhasználónevet kell megadni, ezek parameters néven szerepelnek. A lokáció lekéréséhez tartozó paraméterek a felhasználónév, hosszúsági- és szélességi helyzet, illetve a lokációhoz tartozó név, mely felhasználók helyzete esetén a felhasználó neve. A header mindkért esetben a tokent tartalmazza.

**typealias** ResponseType = ((Data?, Error?) -> Void)?

**class** NetworkService {

**static** **let** shared = NetworkService()

**private** **let** baseUrl = URL(string:"http://localhost:8080/api/")

**private** **let** parameters: Parameters = [

"username": UserDefaults.standard.getUsername()

]

**private** **let** parametersLoc: Parameters = [

"username": UserDefaults.standard.getUsername(),

"longitude": UserDefaults.standard.getLongitude(),

"latitude": UserDefaults.standard.getLatitude(),

"name": UserDefaults.standard.getUsername(),

]

**private** **let** headers: HTTPHeaders = ["Authorization":UserDefaults.standard.getToken()]

*// Singleton, tehat nem lehet belole tobbet letrehozni, ezert privat az init*

**private** **init** () {}

A get és post metódusok bemenetének szükséges megadni a végpont URL címét, illetve, hogy milyen típusú választ várunk. A get kérések esetén vizsgáljuk, hogy a felhasználó bevan-e jelentkezve és ha nem akkor érték nélkül térünk vissza. Ha minden feltétel megfelel, akkor az adatokat adja vissza a get kérés. Post esetén a szerver adatbázisába küldünk fel adatot és visszakapjuk válaszként hogy sikeres volt-e a felküldés. A metódusok belsejében látható, hogy az Alamofire-t hívtam segítségül a kommunikációhoz.

**func** get(endpoint: Endpoints, completion: ResponseType) {

**guard** **let** baseUrl = baseUrl, **let** url = URL(string: "\(baseUrl)\(endpoint.rawValue)") **else** { **return** }

Alamofire.request(url, method: .get, parameters: parameters, encoding: JSONEncoding.default, headers: headers).responseJSON { (response) **in**

*// Ha hiba van a lekerdezesben*

**if** response.response?.statusCode ?? 300 > 299 {

completion?(**nil**, response.error)

} **else** **if** **let** data = response.data {

completion?(data, **nil**)

}

}

}

**func** post(endpoint: Endpoints, completion: ResponseType) {

**guard** **let** baseUrl = baseUrl, **let** url = URL(string: "\(baseUrl)\(endpoint.rawValue)") **else** { **return** }

Alamofire.request(url, method: .post, parameters: parametersLoc, encoding: JSONEncoding.default, headers: headers).responseJSON { (response) **in**

**if** **let** data = response.data {

completion?(data, **nil**)

}

}

}

}

A NetworkService-n felül szükség volt még egy DownloaderService osztályra is, ahol a különböző szerveren lévő entitások kezeléséhez írom meg a metódusokat.

Itt is két fajta kérés létezik. A getter metódusok között szerepel a getEvents és a getPrograms. Az alábbi kódban látható, hogy a NetworkService-ből létrehoztam egy példányt a metódusban a kommunikáció felépítéséhez. A getEvents metódus a visszatérésnél válaszban egy Event tömbbel tér vissza, melyben szerepel az összes szerveren lévő esemény. A completion résznél pedig meghívásnál megadhatunk egy esemény tömböt, ahova a lekérdezett adatokat mentjük.

**func** getEvents(completion: (([Event]) -> Void)?) {

**let** networkService = NetworkService.shared

networkService.get(endpoint: .events, completion: { response, error **in**

**var** events: [Event]

**if** **let** response = response {

**do** {

events = **try** JSONDecoder().decode([Event].**self**, from: response)

completion?(events)

} **catch** {

print("A dekodolas sikertelen volt.")

}

}

})

}

Meghívása az alábbiak szerint történik, ahol látható, hogy az events visszatérésben fognak szerepelni a lekérdezett események, melyet egy előre létrehozott event tömbhöz adok hozzá.

**var** event = [Event]()

**let** downloaderService = DownloaderService.shared

downloaderService.getEvents(completion: { events **in**

**self**.event.append(contentsOf: events)

print(events)

**self**.setUpView()

})

Hasonlóan történik a DonwloaderService osztályban a szerverre történő adatok felvitele is. Az addLocation mindössze annyiban tér el a getEvents metódustól, hogy a válaszában egy Geo típussal tér vissza, illetve a NetworkService meghívása post kéréssel történik.

**func** addLocation(completion: (([Geo]) -> Void)?) {

**let** networkService = NetworkService.shared

networkService.post(endpoint: .location, completion: { response, error **in**

**var** locations: [Geo]

**if** **let** response = response {

**do** {

locations = **try** JSONDecoder().decode([Geo].**self**, from: response)

completion?(locations)

} **catch** {

print("A dekodolas sikertelen volt.")

}

}

})

}

A felhasználók helyzetének felküldése a szerverre az alábbi módon történik. Az alkalmazás előzetesen engedélyt kér a felhasználóktól helyzetük megosztására, majd ezután az alkalmazásba bejelentkezásük után automatikus felküldi a szerverre helyzetüket. Illetve, engedélyezés hiányában nem küld fel adatokat. A lokációt is a tokenhez hasonlóan a UserDefaults-ban tároltam, melyről a következő fejezetben írok részletesebben.

**guard** **let** sourceCord = locationManager.location?.coordinate, CLLocationCoordinate2DIsValid(sourceCord) **else**{

print("Forrás koordináta üres!")

**return**

}

**let** latiSource = sourceCord.latitude

**let** longiSource = sourceCord.longitude

UserDefaults.standard.setLatitude(value: latiSource)

UserDefaults.standard.setLongitude(value: longiSource)

**let** locdownloaderService = DownloaderService.shared

locdownloaderService.addLocation(completion: { geos **in**

print(geos)

})

## Authentikáció

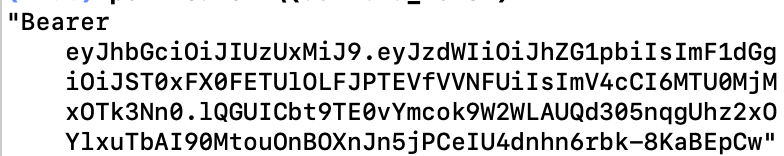
Az authentikáció a JSON Web Token (JWT) segítségével történik. A JWT egy nyitott szabvány, ami definiál egy utat, melyen biztonságosan továbbíthatunk információkat JSON formában az egyes felek között. Az információ digitális aláírással rendelkezik, így verifikálható és megbízható. Az aláírást titkos (HMAC algoritmus) vagy publikus kulccsal (például: RSA) végezhetjük.

Két féle lehetőségünk van a tokenek használatára. Egyrészt használhatjuk authentikációhoz, ez a leggyakoribb használata a JWT tokeneknek. A User bejelentkezésnél megkapja a szervertől a JWT tokent, melyet ezután az összes szerver-kérésnek tartalmaznia kell, hogy az authentikálás létre jöjjön.

Másrészt információ továbbításhoz is használhatjuk, mellyel biztonságosan küldhetünk adatokat a felek között. A tokenek aláírással rendelkezhetnek, mely alapján biztosak lehetünk, hogy a küldő valóban az, akinek mondja magát. Továbbá, az aláírás segítségével azt is vizsgálhatjuk, hogy az információt manipulálták-e.

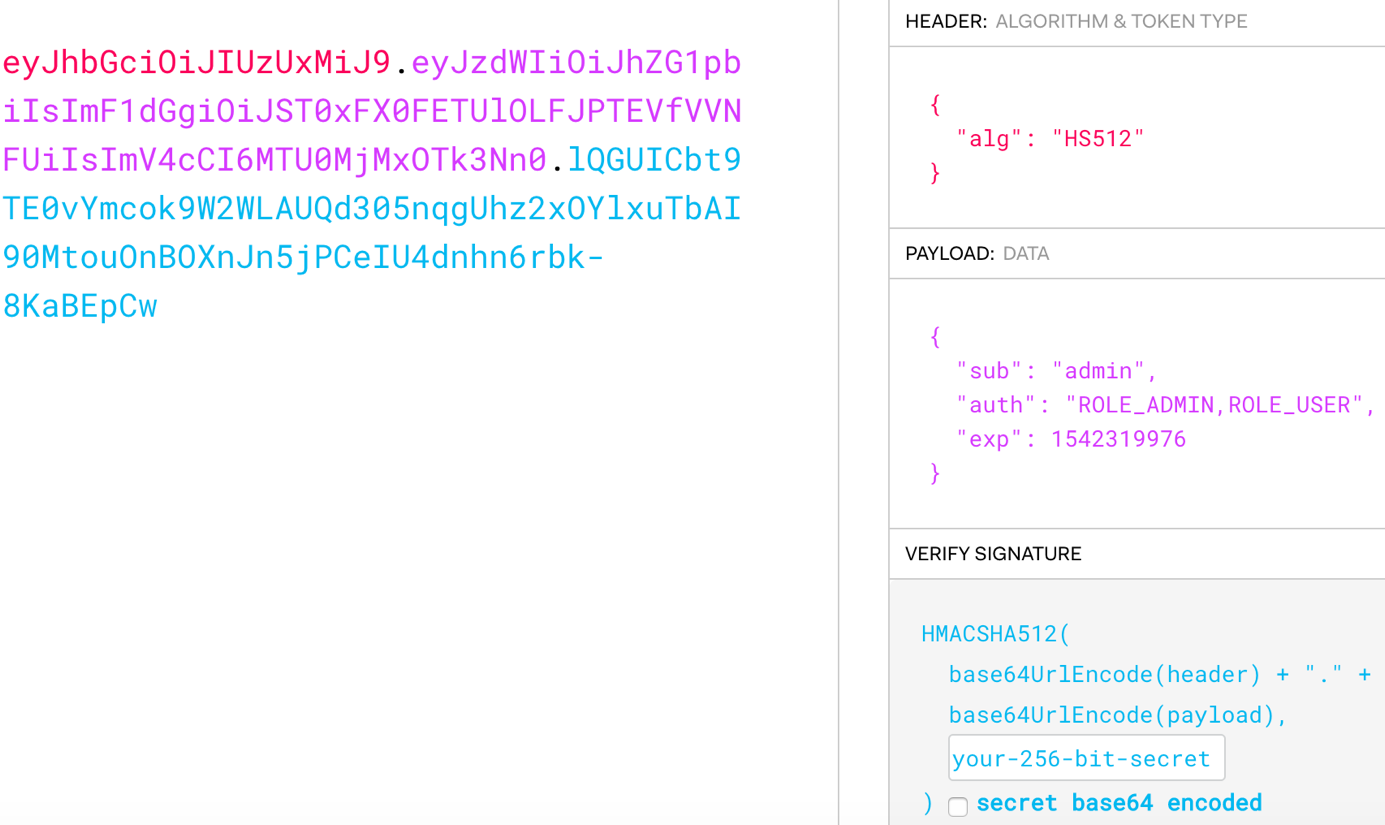
Egy JWT három részből áll: ezek a Header, Payload és Signature (aláírás). A Header tipikusan két részt tartalmaz: a token típusát és a használt hash algoritmusból áll. A Payload tartalmazza a claims-eket, ilyenek az egyes entitásokhoz tartozó információk (például: felhasználóknál név, email, stb.) és egyéb további adatok. A Signature pedig a küldő digitális aláírása, amivel verifikálhatjuk az üzenetet.

Az alkalmazásomban szereplő aktuális tokennek a szerkezete szerepel az alábbi képen. A tokenben szereplő két pont választja el a JWT token három részét egymástól.



A JWT hivatalos oldalán lehetőségünk van tokenünk dekódolására, mely az alábbi képen látható. A Header részből kinyerhető információ alapján a tokenünk a HS512 algoritmust használja. A Payload-ból láthatjuk, hogy a bejelentkezett felhasználó neve admin, aki admin és felhasználói jogokkal egyaránt rendelkezik. A Signature-ből pedig láthatjuk, hogy HMAC SHA512 algoritmust használ.

Az authentikációhoz a már említett Alomofire-t hívtam segítségül.



A tokent a szervertől kapjuk meg, miután elküldtünk neki egy beregisztrált felhasználónév és jelszó párost az alkalmazás bejelentkező felületén. A megkapott tokent a szerver hálózati kéréseknél a HTTP Header-ben kell feltüntetni, ezzel jogosultságot kapva a kérésben szereplő adatok lekéréséhez. A Header kulcsának neve az ,,Authorization” és az értéke a token az alábbi formában:

**var** headers: HTTPHeaders = ["Authorization":"Bearer token"]

Itt a kérésben, a szervertől kapott válasz alapján helyettesítjük a tokent a kapott értékkel az authentikációhoz. A header értékét a kapott tokennel az alábbi updateValue() metódussal változtathatjuk:

headers.updateValue("Bearer \(auth.id\_token)", forKey: "Authorization")

Itt az auth.id\_token a szervertől kapott token értéke.   
Mivel ennek a tokennek minden hálózati kérésben szerepelnie kell, ezért szükséges tárolnunk. A token tárolására a UserDefaults tárolást használtam.

A UserDefaults egy interfész, amely a felhasználó default adatbázisa. Itt kulcs-érték párokat tudunk perzisztensen tárolni. Az itt megadott paraméterek alapértelmezett értékekre vonatkoznak, melyek általában a rendszer alapértelmezett állapotát határozzák meg induláskor vagy az egyes alapértelmezett viselkedéseit.

A UserDefaults gyorsítótárazza a kapott információkat, ezáltal nem kell mindig megnyitnunk az alapértelmezett adatbázist, hogy ezeket az értékeket megkapjuk.

A UserDefaults osztályhoz érdemes írni egy wrapper osztályt, mellyel lekérhetjük és megadhatjuk az itt tárolt értékeket. Az alkalmazásomban a felhasználónevet és a tokent tároltam itt az alábbi módon:

A UserDefaults kulcsai közé felvettem az értékeket:

**enum** UserDefaultsKey: String {

**case** username

**case** accestoken

}

Majd írtam hozzájuk getter és setter metódusokat az értékek megadásához és lekéréséhez. Például az alábbiakban látható a token lekérése. Ennek fontos volt a tárolása, mert minden egyes hálózati kérésben szükséges elküldeni. Látható, hogy a token egy String érték, melyet a UserDefaultsKey használatával a kulcs nevének megadásával tudunk beállítani. Lekérésnél pedig ugyanerre a megadott kulcs névre hivatkozunk. Ha nem sikerült a token tárolása, vagy még nem kaptunk, akkor a getter metódus ,,nil” értékkel tér vissza.

**func** setToken(value: String) {

set(value, forKey: UserDefaultsKey.accestoken.rawValue)

synchronize()

}

**func** getToken() -> String? {

**guard** **let** token = UserDefaults.standard.string(forKey: "accestoken")

**else** {

**return** nil

}

**return** token

}

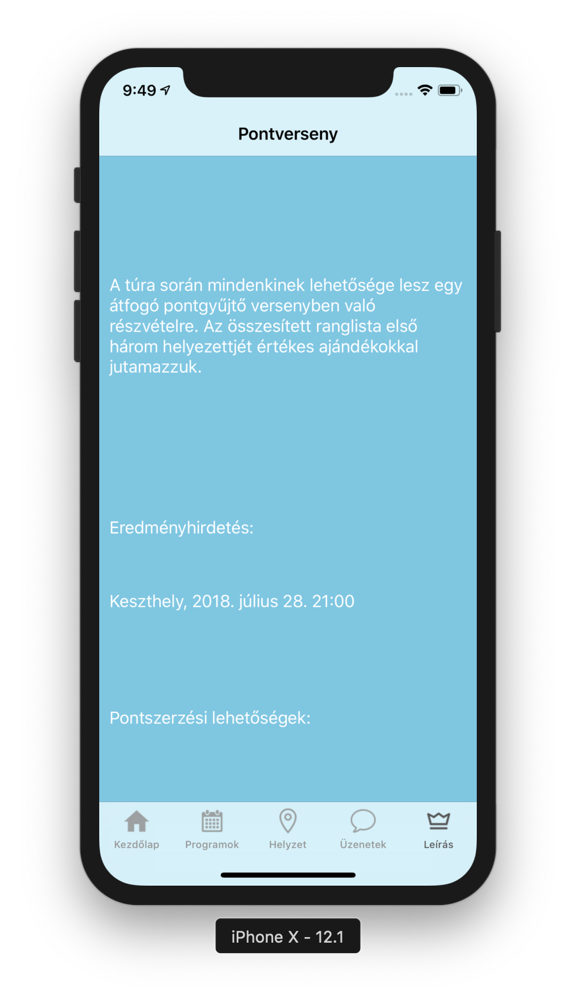
# Megvalósítás

## Képernyők

## Funkciók

* + 1. **Főoldal – Menü**

I



# Tesztelés

# Továbbfejlesztési lehetőségek

# Összefoglaló

# Irodalomjegyzék