iOS Tutorial - aplikacija Discount Locator

**Tim T20:**

Darijan Vertovšek

Danijel Vincijanović

Hrvoje Hrvoić

**Poveznica na GitHub repo:**

[https://github.com/dvertovsek/DiscountLocatorIOSFINAL](https://github.com/dvertovsek/DiscountLocatorIOSFINAL%20)

**Sadržaj**

[1. Arhitektura sustava i IDE 3](#_Toc441423199)

[1.1. Arhitektura iOS sustava 4](#_Toc441423200)

[1.2. Xcode 7](#_Toc441423201)

[1.3. Životni ciklus iOS aplikacije 14](#_Toc441423202)

[2. Swift i Playground 19](#_Toc441423203)

[3. ORM 36](#_Toc441423204)

[3.1. Arhitektura aplikacije 37](#_Toc441423205)

[3.2. Instalacija Cocoapods i podešavanje RealmSwifta 37](#_Toc441423206)

[4. Web servis 42](#_Toc441423207)

[4.1. Dodavanje i implementacija HTTPRequest klase (ws modul) 43](#_Toc441423208)

[4.2. Dodavanje i implementacija DataLoader i JsonAdapter klase (core modul) 46](#_Toc441423209)

[4.3. Dodavanje i implementacija WebServiceDataLoader klase 50](#_Toc441423210)

[4.4. Prikazivanje dohvaćenih podataka na korisničkom sučelju 52](#_Toc441423211)

[5. Prikaz podataka 57](#_Toc441423212)

[5.1. Prilagođavanje ćelija za prikaz (custom cells) 58](#_Toc441423213)

[6. Kreiranje izbornika 65](#_Toc441423214)

[6.1. Dodavanje SWRevealViewControllera 66](#_Toc441423215)

[7. Mape i postavke 73](#_Toc441423216)

[7.1. Mape 74](#_Toc441423217)

[7.2. Postavke 81](#_Toc441423218)

[7.3. Dodavanje modalnog kontrolera 82](#_Toc441423219)

[7.4. Dodavanje kontrolera za postavke 85](#_Toc441423220)

[7.4.1. Vizualizranje radiusa na mapi 95](#_Toc441423221)

[8. Monetizacija 99](#_Toc441423222)

[9. Push notifikacije 102](#_Toc441423223)

[9.1. Zahtjev za certifikatom 103](#_Toc441423224)

[9.2. Registriranje aplikacije 104](#_Toc441423225)

[9.3. Preuzimanje certifikata 107](#_Toc441423226)

[9.4. Podešavanje provision profila u XCodeu 110](#_Toc441423227)

[9.5. Implementacija serverske strane 112](#_Toc441423228)

[10. Modularnost 114](#_Toc441423229)

[10.1. Koncept slijepe komunikacije modula 114](#_Toc441423230)

[11. Testiranje 118](#_Toc441423231)

[11.1. Unit testovi 118](#_Toc441423232)

[11.2. UI Testovi 127](#_Toc441423233)

1. Arhitektura sustava i IDE

**SAŽETAK**

u prvome dijelu našega putovanja upoznati ćemo se sa arhitekturom iOS-a. Zatim ćemo zaviriti u integrirano razvojno okruženje (eng. IDE - Integrated Development Environment) koje će se koristiti pri realizaciji ovih vježbi, Xcode.

Osim toga saznati ćemo na koji način se pokreće aplikacija i koji događaji se tada izvršavaju tj. upoznati ćemo se sa životnim ciklusom jedne iOS aplikacije.

**KLJUČNE RIJEČI**

arhitektura, iOS, IDE, Xcode, životni ciklus iOS aplikacije

**POTREBNA PREDZNANJA**

značenje i pojam arhitekture sustava

**ISHODI UČENJA**

Nakon ove vježbe student bi trebao biti upoznat sa arhitekturom iOS-a, snaći se u njegovom razvojnom okruženju te znati osnove životnog ciklusa iOS aplikacije

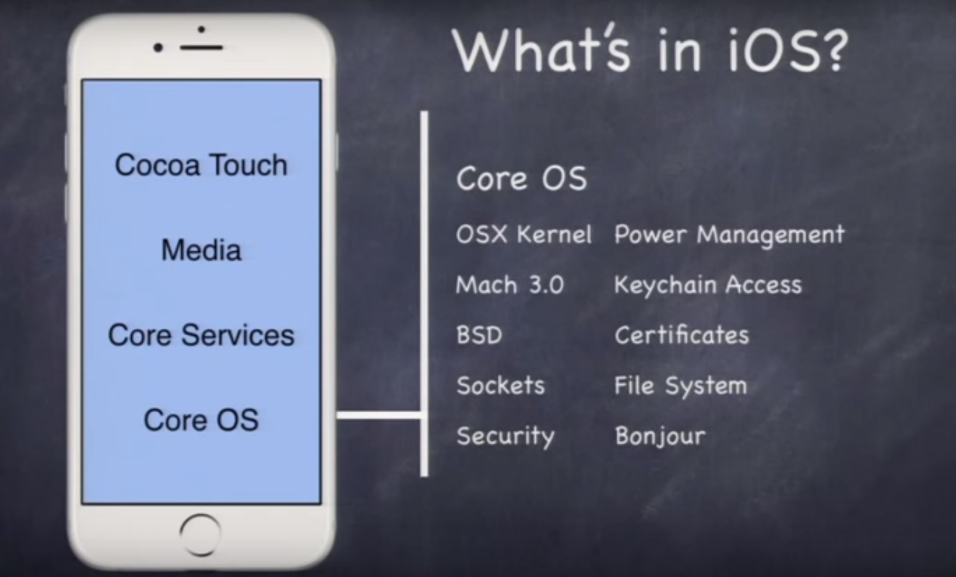


GITHUB LINK

<https://goo.gl/WCNkeX>

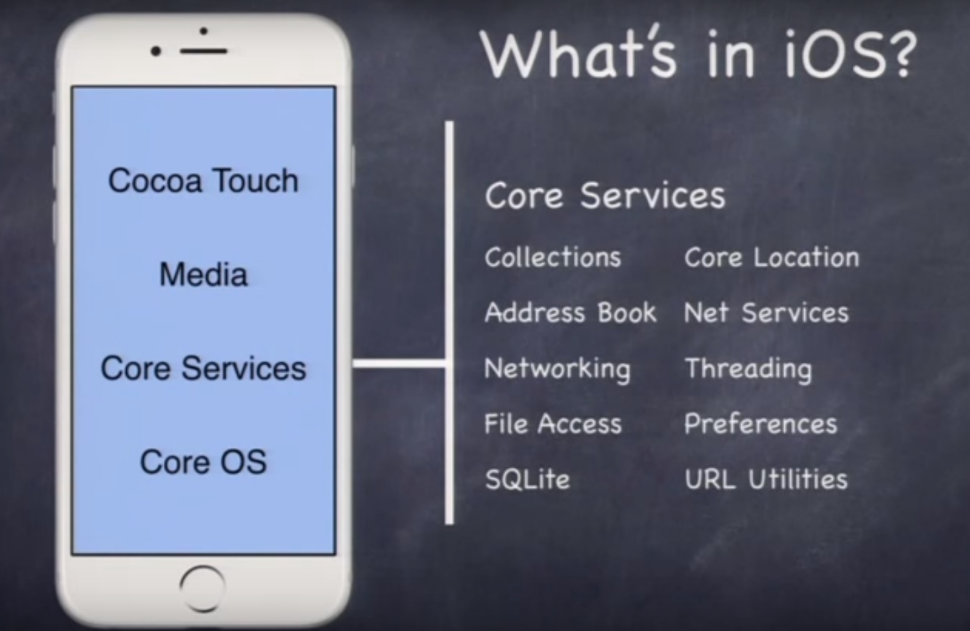
**RAZRADA**

* 1. Arhitektura iOS sustava

****

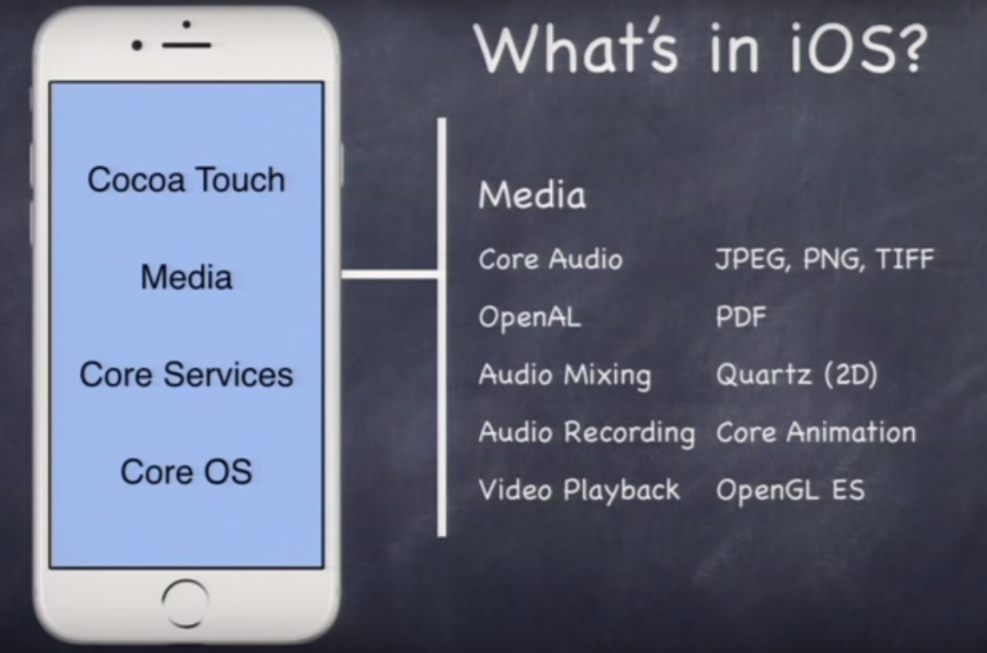
Slika 1: Arhitektura iOS-a: Core OS

Ima četiri sloja. Prvi i najniži (korisniku najudaljeniji) sloj je operacijski sustav u užem smislu (eng. Core OS). U njemu se nalazi jezgra (eng. *kernel*) koji je zapravo baziran na Unix operacijskom sustavom (BSD) pa zato možete i vidjeti sličnosti naredbi unutar ljuske između Linuxa i OS X-a, socketi, dio za upravljanje napajanjem i Bonjour - dio za upravljanje mrežnom konfiguracijom (dodjela adresa i DNS). Mach 3.0 je dio koji služi za podršku multiprocesorskom radu. Sve to i ostale komponente arhitekture koje su samoobjašnjavajuće možete vidjeti na sljedećoj slici:

****

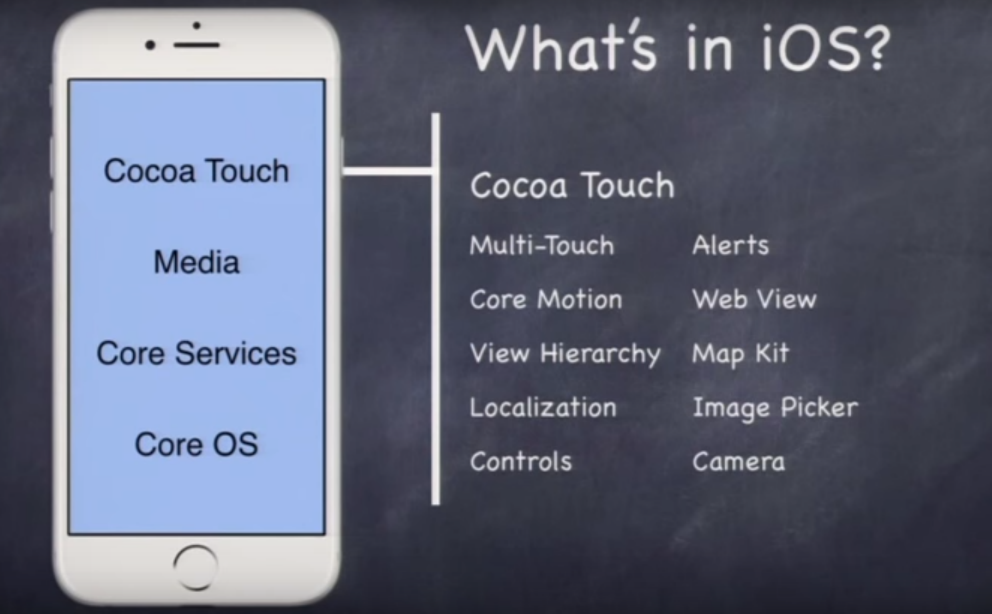
Slika 2: Sloj Core Services

Na prethodnoj slici moguće je vidjeti drugi sloj odnosno sloj glavnih servisa. To je skup osnovnih sistemskih servisa koji služe aplikacijama. Sadrži individualne tehnologije koje pružaju funkcionalnosti poput lokacije, iCloud-a (pruža vašim aplikacijama da spremaju podatke na server) i upravljanja mrežom tipa servisi za mreže ravnopravnih čvorova (eng. *P2P*). Također, kao i u Androidu pruža servis za rad s SQLite bazama podataka. Brine se za pristup sklopovlju putem objektno orijentirane paradigme preko servisa.

****

Slika 3: Multimedijski sloj

Na prethodnoj slici je multimedijski sloj koji obuhvaća razvojne okvire za 3D, audio, video formate itd. Za vas je bitno da znate da postoji taj sloj, te ako će vam taj sloj zatrebati, npr. ukoliko ćete raditi igricu ili uređivati video svakako proučite te biblioteke.

****

Slika 4. Cocoa Touch sloj

Zadnji i korisniku najbliži sloj, tj. onaj sloj s kojim ćemo se mi najviše baviti jest Cocoa Touch sloj. Služi za komunikaciju s korisnikom odnosno programer će putem ovog sloja ostvariti interakciju s krajnjim korisnikom. Sastoji se od osnovnih razvojnih okvira za izgradnju iOS aplikacija. Ovi okviri definiraju izgled vaše aplikacije. Pružaju ključne tehnologije poput multitasking-a, prepoznavanja ulaznih gesti na dodir, push notifikacija te sistemske servise visoke razine. Dakle, korisnik će prilikom dizajniranja aplikacije prvo potražiti rješenje u ovom skupu biblioteka, ukoliko njegove potrebe nisu zadovoljene strovaliti će se u niže slojeve.

* 1. Xcode

Xcode je integrirano razvojno okruženje koje sadrži niz alata za razvijanje softvera. Razvijen je od strane Applea i koristi se za razvijanje aplikacija za OSX, i iOS operacijske sustave.

Pokretanjem Xcode-a otvara se pozdravni prozor (vidi sljedeću sliku). Na lijevoj strani prozora nalaze se tri opcije.

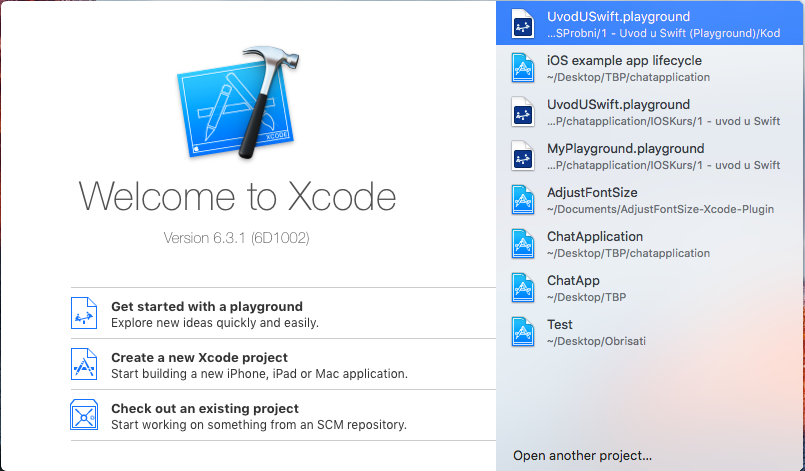
Get started with a playground - Playground jest alat u sklopu Xcode-a koji služi za real-time kompajliranje Swift programa. Svrha Playgrounda je učenje i isprobavanje programskog jezika Swift.

Create a new Xcode project - ovom opcijom pokrećemo novi projekt

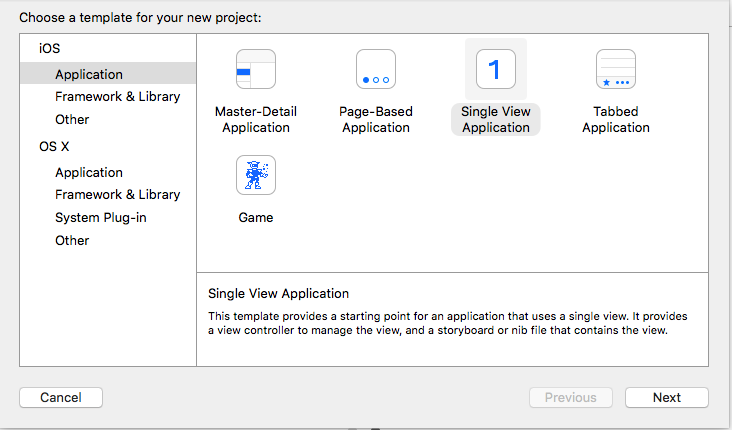
Check out an existing project - pokretanje postojećeg projekta iz repozitorija sa sustava za verzioniranje koda

S desne strane se nalaze postojeći projekti na disku računala.

Opciju koju ćemo mi odabrati je ‘Create a new Xcode project’ čime ćemo pokrenuti novi projekt.



Nakon što smo odabrali kreiranje novog projekta otvara nam se prozor u kojemu biramo predložak projekta.



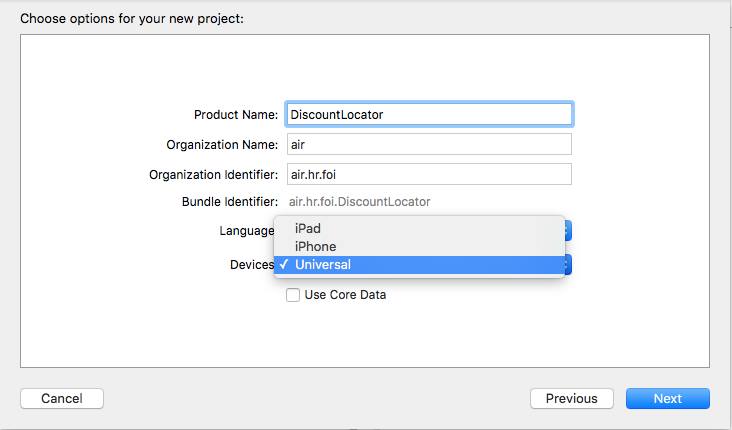
S lijeve strane se nalazi ponuđeni izbornik gdje biramo koju vrst projekta izrađujemo. Mi ćemo uvijek odabirati aplikaciju za iOS platformu. S desne strane imamo odabir predloška aplikacije. U svrhu upoznavanja sa sučeljem, za sada ćemo odabrati Single View Application. Radi se o jednostavnoj MVC aplikaciji. Odaberite Next.



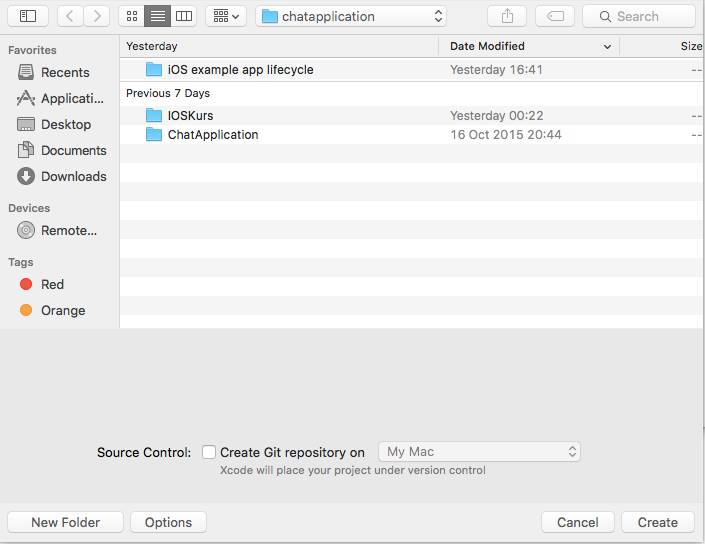
Sada odabiremo ime proizvoda, organizaciju te identifier. Koja je svrha ovoga? Na temelju organization identifiera, generira se Bundle identifier te, kada budemo objavljivali aplikaciju za Apple store, to predstavlja jedinstveni identifikator aplikacije. Može postojati više aplikacija sa istim imenom, no Bundle identifier je jedinstven za svaku aplikaciju koja se nalazi na Store-u.

Kao jezik u kojem želimo programirati ćemo odabrati Swift, no može se još odabrati i jezik Objective-C.

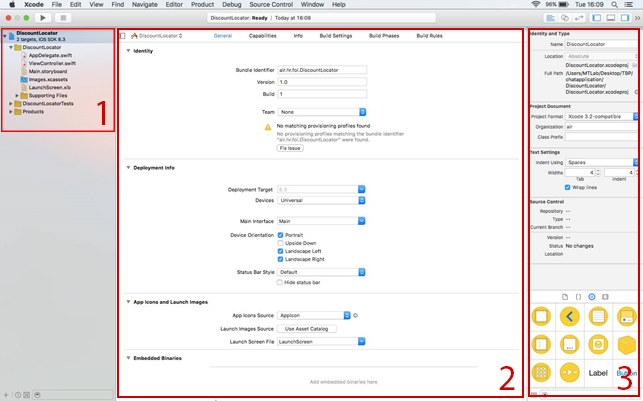
Osim toga, moguće je odabrati za koje uređaje razvijamo aplikaciju (vidi sljedeću sliku).



Odabiremo opciju Universal ukoliko želimo razvijati aplikaciju prilagođenu i za iPad i iPhone uređaje, odnosno i za mobilne telefone i za tablete.



Potrebno je odabrati gdje ćemo spremati projekt. Postoji opcija da se automatski kreira repozitorij na Git-u. Budući da već imamo kreirani repozitorij, ovu opciju ćemo ostaviti neoznačenom.

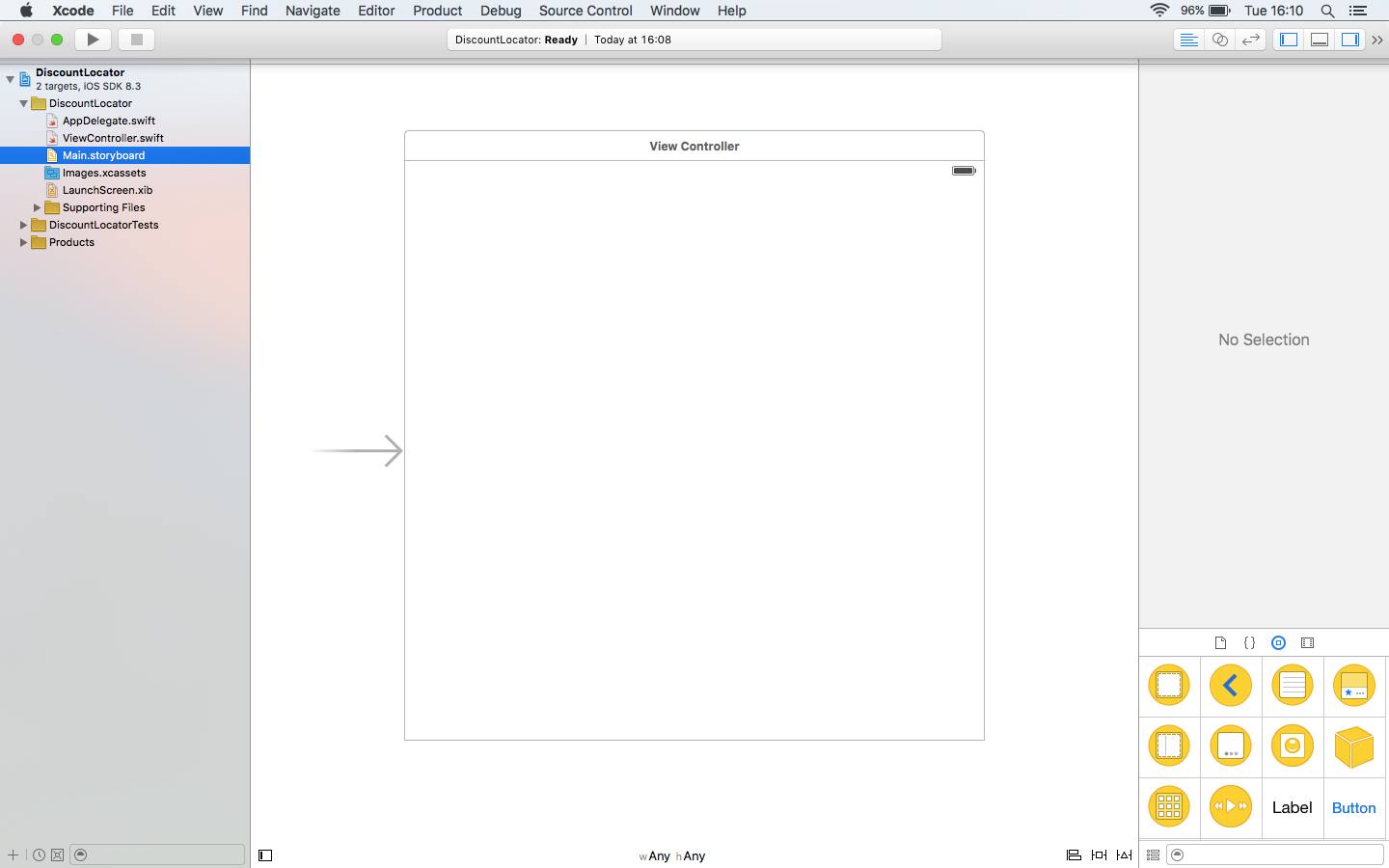


Potrebno je zamijetiti sljedeće dijelove IDE-a:

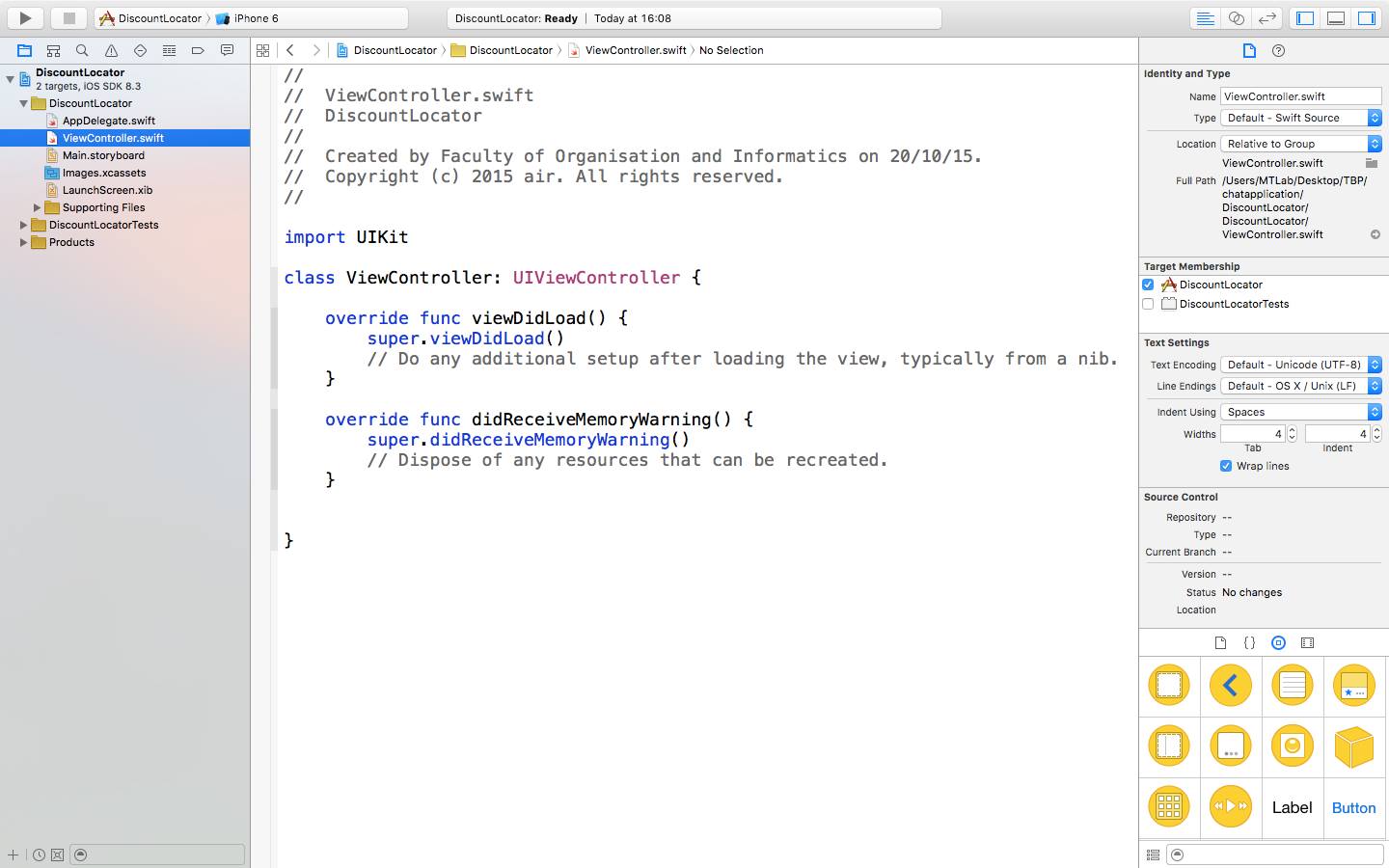
[1] – Struktura projekta koja prikazuje hijerarhiju foldera i njemu pripadnih datoteka unutar projekta. Dvije najbitnije datoteke su Main.storyboard te ViewController.swift. U MVC svijetu Main.storyboard je view, a ViewController.swift kontroler koji upravlja tim view-om (Apple forsira MVC arhitekturu).

[2] – Radna površina koja prikazuje ono što je odabrano u strukturi projekta. Kada odaberemo sam projekt s lijeve strane, u sredini se prikazuju detalji vezani za verzioniranje aplikacije, za ciljane uređaje te opće postavke poput ikone aplikacije.

[3] – U pravilu, s desne strane će se nalaziti opcije o onome što ćemo odabrati na radnoj površini (i to obično na Main.storyboardu), odnosno opcije o objektima korisničkog sučelja.



Prethodna slika prikazuje Main.storyboard koji je odabran s lijeve strane. To je zapravo, kao što smo ranije napomenuli, korisničko sučelje – View.



Sada smo odabrali ViewController.swift. Primjećujemo nekakve metode životnog ciklusa o kojima će biti riječi kasnije.



Prethodna slika prikazuje uvećanu desnu stranu IDE-a koju smo prije spomenuli. Tu se nalaze postavke o odabranom objektu korisničkog sučelja te također Object library sa elementima koje je moguće tehnikama drag-drop dodati na samo sučelje.



Ukoliko želimo pokrenuti aplikaciju, Xcode dolazi sa ugrađenim simulatorom za sve iOS uređaje (i jako je brz).



Unutar simulatora se poigrajte sa opcijama Hardware gdje možete mijenjati postavke uređaja poput orijentacije ili zaključavanja, kontrole poput home i back gumba. Ukoliko imate neki iOS uređaj pri ruci, svakako ga iskoristite i spojite ga umjesto virtualnog uređaja.

* 1. Životni ciklus iOS aplikacije

Ovu lekciju bismo zaključili sa par riječi o životnom ciklusu iOS aplikacije. U ovom dijelu ćemo biti vrlo kratki, budući da se još nismo upoznali sa Swiftom, programskim jezikom za razvoj iOS aplikacija. Što se događa kada korisnik pritisne na ikonu aplikacije? U Swiftu se poziva prvo klasa **AppDelegate**, i to iznad nje se treba nalaziti direktiva

**@UIApplicationMain**

Bez te direktive, aplikacija ne zna koju klasu treba pokrenuti i doći će do greške. Unutar te klase je već implementirano par metoda koje služe za upravljanje događajima životnog ciklusa aplikacije.

Osim metoda unutar klase AppDelegate, ranije spomenuta klasa **ViewController** također može naslijediti metode za upravljanje događajima. Unutar klase AppDelegate, u metodi application, se nalaze informacije o tome na koji je način aplikacija otvorena i s kojim parametrima. Primjerice, ukoliko je aplikacija otvorena kao rezultat nekog URL-a u pretraživaču.

Metoda **applicationWillResignActive** se poziva prije nego aplikacija pređe iz aktivnog u neaktivno stanje. Ovo se može dogoditi uslijed privremenih prekida (npr. Dolazni poziv ili korisnik pritisne tipku 'Home'). Metoda se koristi kako bi se pauzirali tekući zadaci ili zaustavilo mjerenje vremena. Igrice koriste ovu metodu za pauziranje igrice.

Metoda **applicationDidEnterBackground** se koristi kako bi se otpustili dijeljeni resursi, spremili korisnički podaci te spremilo dovoljno informacija o stanju aplikacija kako bi se kasnije iste mogle rekonstruirati. Ukoliko aplikacija može raditi u pozadini, pri izlazu iz aplikacije se poziva ova metoda umjesto **applicationWillTerminate**.

Metoda **applicationWillEnterForeground** se poziva kao dio prijelaza iz pozadinskog u neaktivno stanje. Tu se mogu vratiti promjene učinjene prilikom ulaska u pozadinsko stanje.

Metoda **applicationDidBecomeActive** ponovno pokreće sve zadatke koji su bili pauzirani dok je aplikacija bila neaktivna. Ukoliko je aplikacija bila u pozadini, tu se osvježava korisničko sučelje.

Metoda **applicationWillTerminate** se poziva prije nego što će se aplikacija terminirati. Tu je potrebno spremiti podatke, slično kao i kod metode applicationDidEnterBackground.

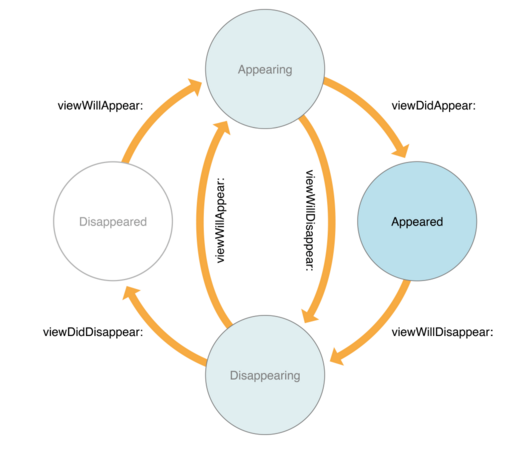
Unutar ViewController-a također postoji par metoda koje se koriste kako bi se upravljalo događajima životnog ciklusa, ali u ovom slučaju ne aplikacije, nego pogleda.

Životni ciklus pogleda se sastoji od **6 događaja**, odnosno metoda:

Metoda **viewDidLoad** se poziva nakon što se pogled učita u memoriju i tu je prikladno kreirati ili učitati sve elemente pogleda (npr. Labele)

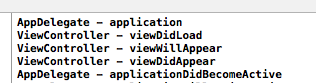
Metoda **viewWillAppear** se poziva prije nego što će se pogled pokazati i koristi se za ažuriranje podataka na formi.

Metoda **viewDidAppear** se poziva ukoliko želimo pokrenuti nove dretve koje iziskuju mnogo vremena kako bi se izvršile (npr. Poziv na web servis za dohvaćanje podataka). Budući da pogled već postoji u memoriji, moguće je prikazati poruku čekanja korisniku dok se dohvaćaju podaci.

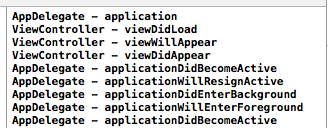


Slika 5. Životni ciklus pogleda

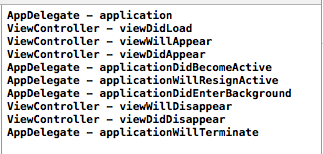
Prilikom pokretanja aplikacije, prvo se pokreće metoda **application**, zatim se pokreće view unutar ViewControllera, te zatim opet unutar klase AppDelegate metoda **appliationDidBecomeActive**.



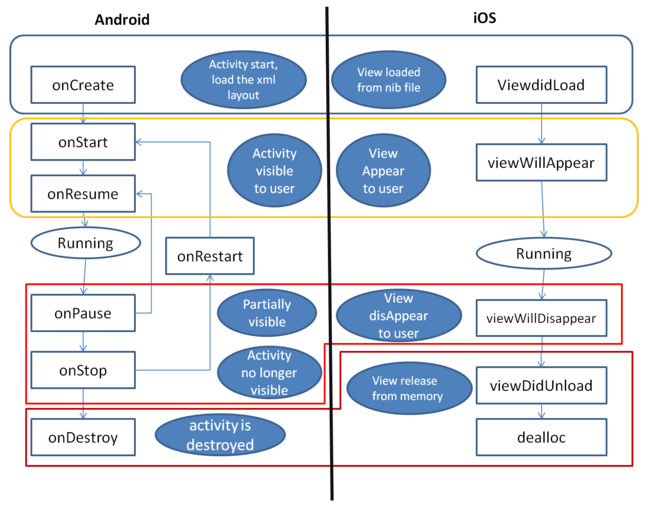
Kada pauziramo aplikacija (pritisak na tipku 'Home'), pozivaju se dvije metode (applicationWillResignActive i applicationDidEnterBackground), te prilikom ponovnog vraćanja aplikacije u fokus također dvije metode (applicationWillEnterForeground te ponovno metoda application DidBecomeActive)



Prilikom zatvaranja aplikacije poziva se metode koje prate gašenje i brisanje pogleda



Ukoliko poznajete platformu Android, evo korisne slike koja prikazuje elemente životnog ciklusa aktivnosti i uspoređuje je s elementima životnog ciklusa pogleda u iOS platformi.



Slika 6. Gruba usporedba Android activity životnog ciklusa i iOS životnog ciklusa pogleda

Korisni linkovi

<http://nshipster.com/launch-options/>

1. Swift i Playground

**SAŽETAK**

Kako bismo razvijali softver za iOS potrebno je poznavati i naučiti pripadajući programski jezik – Swift. Upoznati ćemo se sa osnovama Swifta i njegovim objektno-orijentiranim konceptima. Koristiti ćemo Appleov alat prikladan za učenje Swifta – Playground.

**KLJUČNE RIJEČI**

iOS, Swift, OOP, Apple, Playground

**POTREBNA PREDZNANJA**

osnovni programski koncepti i konstrukti te objektno-orijentirane paradigme

**ISHODI UČENJA**

Polaznik bi nakon ove vježbe trebao savladati koncepte programskog jezika Swift.



GITHUB LINK

<https://goo.gl/JE9XJS>

**RAZRADA**

Swift je novi, moćan i intuitivan programski jezik kreiran od strane Applea za razvoj iOS, OS X, watchOS i tvOS aplikacija. Dizajniran je kako bi naprednim razvojnim inženjerima dao slobodu i sposobnost koja im je potrebna kako bi razvijali aplikacije nove generacije. Također, otvara svijet novih mogućnosti za sve ostale jer je sintaksno intuitivniji za razliku od Objective-C-a koji je dosad bio jedini službeni programski jezik namijenjen razvoju aplikacija na Apple platformama.

Za izučavanje programskih koncepata u Swiftu koristiti ćemo programski alat Playground u kojemu se interaktivno mogu isprobavati djelići koda. Naime, dijelovi programskog koda se odmah izvršavaju. Playground je sastavni dio Xcode-a.

Tradicija početka učenja svakoga programskoga jezika je ispis rečenice "Hello, world!" pa ćemo na taj način započeti i učenje Swifta.

print("Hello world!")

Ukoliko ste programirali u Pythonu ovaj kod bi vam trebao izgledati poznato. Naime, Swift je programski jezik sličan (sintaksno i semantički) Javascriptu i Pythonu. Primjetimo da se na kraju naredbenog retka ne nalazi ';'.

**VARIJABLE I KONSTANTE**

U Swiftu inicijalizacija varijabli može biti izvršena na implicitni gdje na temelju dodijeljene vrijednosti kompajler zna koji tip poprima varijabla:

var str = "Ovo je playground - app za isprobavanje dijelova koda"

var mojaVarijabla = 43

ili eksplicitni način gdje se podrazumijeva da je tip varijable strogo definiran:

var str: String = "Hello world!"

Konstante se zadaju sa ključnom riječi 'let' te se jednom dodijeljena vrijednost ne može mijenjati:

let mojaKonstanta = 30

**Konkatenacija stringova**

Konkatenacija stringova se može obavljati na već poznati način, pomoću '+' znaka.

let voce = "kruska"

let jabuka = "Moja jabuka " + "nije" + voce

let broj = 2

let recenica = "Imam " + String(broj) + " jabuke"

Swift omogućuje još jedan jednostavniji način da se uključe vrijednosti u stringove gdje se vrijednosti ugrađuju u string tijekom njegove definicije. Za to koristimo znak '\' te varijablu navedemo u zagradama.

let jabuke = 666

let kruske = 333

let stringJabuke = "Imam \(jabuke) komada jabuka"

let stringVoce = "Imam \(jabuke+kruske) komada voća"

**Polja i rječnici (asocijativna polja)**

Polja i rječnici se kreiraju korištenjem uglatih zagrada '[]', a njihovim elementima se pristupa tako da se u zagrade napiše pripadni indeks ili ključ.

var polje = ["mačka", "pas","kontrabas"]

var asocPolje = ["Malcom": "U sredini", "Dewey": "Najmanji"]

asocPolje["Malcom"] //dohvaćamo vrijednost 'U sredini'

Kako bi se kreiralo prazno polje ili rječnik koristi se sintaksa inicijaliziranja.

var praznoPolje = [String]()

var prazniRijecnik = [String: Float] ()

Ako je tip polja prepoznatljiv od kompajlera tada se ono može ponovno kreirati.

polje = []

asocPolje = [:]

**Kontrolne strukture**

Selekcije: if, switch

Iteracije: for-in, for, while, do-while

**for-in iteracija**

//prvi primjer

for i in 0...4 { //... ukljucuje i 4

drugiBrojac += i

}

//drugi primjer

let individRezultati = [75, 23, 103, 84, 15, 777]

for rezultat in individRezultati {

if(rezultat > 50) {

skorTima += 5

}

else {

skorTima += 1

}

}

**switch selekcija**

Razlika između implementacije switch selekcije u ostalim jezicima i Swiftu je ta da se uspoređivati mogu svi tipovi varijable, a ne samo integeri.

Switch selekcija uvijek mora imati case koji će se izvršiti, bilo da se radi o defaultu ili nekom drugom case-u.

Unutaj case-a je moguće definirati složeni izraz kao što možemo vidjeti u primjeru.

let feferona = "ljuta feferona"

switch feferona {

case "blaga":

print("feferona je blaga")

case let x where x.hasPrefix("ljuta"): //složeni izraz u case-u

print ("Fakat je ljuta")

case "ljuta feferona":

print("feferona je ljuta")

default: //mora postojati default

print("Meni su dobre bilokakve feferone")

}

U Appleovoj službenoj dokumentaciji naići ćete na repeat-while koji se trenutno više ne koristi te on predstavlja do-while iteraciju.

Prilikom kreiranja Swifta Apple je uveo mehanizam kojim se sprječava vjerojatnost nastajanja run-time greške uslijed korištenja null vrijednosti. (u Swiftu **nil** predstavlja null)

Navedeni mehanizam je definiranje varijable kao **opcionalne varijable**. Takve varijable mogu poprimiti nil vrijednost. Naime, prilikom definiranja bilo koje varijable, ukoliko se drukčije ne navede smatra se da varijabla ne može poprimiti nil vrijednost i u tom slučaju kompajler prije pokretanja javlja grešku ukoliko prepozna da će varijabla poprimiti vrijednost nil. Za opcionalne varijable nakon tipa se dodaje '**?**'.

var opcionalniInteger: Int? = 10

opcionalniInteger = nil

var pero: String? = nil

Sljedeći kod će javiti grešku prije pokretanja:

var neOpcionalniInt = 10

neOpcionalniInt= nil //ovo baca grešku

Ponekad se if selekcija i pridruživanje vrijednosti koriste u jednom koraku za rad sa vrijednostima koje možda nisu dodijeljenje (nil).

var opcString: String? = "Jozo"

if let ime = opcString {

var pozdrav = "Hello \(ime)"

}

„Ispod haube“ će se prvo provjeriti vrijednost varijable opcString te ukoliko ona nije nil tada je pridruživanje dozvoljeno i izvršiti će se blok koda unutar if selekcije.



Probajte promijeniti vrijednost varijable *opcString* iz "Jozo" u nil. Što će se dogoditi?

**FUNKCIJE – REVOLUCIJA I**

Pomoću ključne riječi func se definiraju funkcije. Sintaksa kreiranja funkcije je sljedeća:

func naziv(imeArgumenta : tipArgumenta) -> povratniTip {

//kod

}

Funkcija može vratiti više vrijednosti u obliku objekta. Primjer jedne takve funkcije gdje funkcija vraća tri vrijednosti (min, max, sum). Tim elementima kasnije pristupamo objektnom notacijom odnosno pomoću točke.

func izracunajStat(ulPolje:[Int]) -> (min: Int, max:Int, sum: Int){

var minim = ulPolje[0]

var maxim = ulPolje[0]

var suma = 0

for broj in ulPolje {

if broj > maxim {

maxim = broj

}

else if broj < minim {

minim = broj

}

suma += broj

}

return (minim,maxim,suma)

}

let statistike = izracunajStat ([5,3,100,3,9])

statistike.0 //dohvaćamo prvi element povratnog objekta tj. statistike.min

statistike.max

statistike.sum

Funkcije također mogu primiti neodređeni broj argumenata koji će prilikom poziva funkcije spremiti u jedno polje. U sljedećem primjeru takvo polje predstavljaju 'brojevi'. Ključno je staviti '...' nakon tipa argumenta. Moguće je pozvati funkciju sa nebrojeno puno argumenata odvojenih zarezom.

func sumaBrojeva (brojevi: Int...) -> Int {

var suma = 0

for broj in brojevi{

suma+=broj

}

return suma

}

sumaBrojeva()

sumaBrojeva(20,30,40,60,708)

U Swiftu je moguće ugnježđivati funkcije. Takav princip je poznat iz programskog jezika Javascript.

func vratiDvadeset() -> Int {

var y = 10

func dodajDeset(){

y += 10

}

dodajDeset()

return y

}

vratiDvadeset()

Funkcija također može poprimati funkciju kao argument gdje se prilikom definicije funkcije u argumentima navodi potpis (eng. signature) druge funkcije.

func imaLiPogodaka(listaBrojeva: [Int] , uvjet: Int->Bool)->Bool{

for broj in listaBrojeva{

if(uvjet(broj)){

return true

}

}

return false

}

func manjiOdDeset(broj: Int) -> Bool {

return broj < 10

}

var brojcici = [20,3,5,60]

imaLiPogodaka(brojcici, manjiOdDeset)

Vidljivo je da je drugi argument funkcije 'imaLiPogodaka' zapravo potpis funkcije 'manjiOdDeset' odnosno funkcija prima integer, a bool joj je povratna vrijednost.



Za one znatiželjne slijedi pravi primjer funkcija-ceptiona. Funkcija koja vraća drugu funkciju gdje se drugoj funkciji delegira vraćanje rezultata pozivanjem iste iz prve, vanjske funkcije.

func vratiFunkciju() -> (String->Int) { //argument druge -> povratni tip druge

func dodajDeset(brojString: String)->Int{

return brojString.toInt()! + 10

}

return dodajDeset

}

**CLOSURE**

Funkcije su zapravo specijalni slučajevi closure-a odnosno blokovi koda koji se mogu pozvati kasnije. Kod unutar closure-a ima pristup varijablama i funkcijama koje su dostupne unutar njezinog dosega.

Sintaksa closure-a je sljedeća:

{ (parametri) -> povratni tip **in**

izrazi

}

Dakle, parametar closure-a se kao kod funkcije navodi u zagrade jedina razlika je da se navodi nakon otvaranja vitičastih zagrada, a ne prije. Nakon parametara se navodi povratni tip kao i kod svake funkcije, ali u ovome slučaju nakon povratnog tipa dolazi ključna riječ 'in' koja označava početak tijela funkcije. Ovakva funkcija se još naziva i anonimnom funkcijom.

var brojcici = [20,3,5,60]

let brojciciPutaDeset=brojcici.map(

{

(broj:Int) -> Int in

let rezultat = broj \* 10

return rezultat

}

)

brojciciPutaDeset // [200,30,50,600]

Dakle, u konstantu brojciciPutaDeset će biti preslikano (eng. Map) polje brojcici pomnoženih sa 10. Metoda map koja radi preslikavanje za argument prima anonimnu funkciju koja je realizirana putem closure-a. Parametar 'broj' koji je integer zapravo predstavlja element polja brojcici. Povratna vrijednost closure-a je Int odnosno preslikani broj spremljen u konstantu rezultat. Očito je da metodi map treba samostalno realizirati funkciju preslikavanja elemenata polja.



Napravite closure na isti način tako da on mapira polje brojcici gdje će neparni brojevi postati 0, a parni ostati kakvi jesu.

**Bonus bodovi:** ukoliko se zadatak riješi korištenjem ternarnog operatora ('?') i operatora modulo ('%').

**Hint**: ako zapnete ni slučajno pogledati github (link je na početku vježbe)



Ukolite želite detaljnije istražiti paradigmu closure-a na gitHubu možete proučiti ostale primjere na primjeru funkcije sort (moguće je definirati closure na četiri načina!:) )

**KLASE I STRUKTURE – REVOLUCIJA II**

Klase i strukture su tipovi podataka opće namjene. Kao što znamo unutar klase se definiraju svojstva i metode. Za razliku od ostalih programskih jezika, Swift ne traži samostalno kreiranje sučelja (eng. Interface) te njihovih implementacija unutar klasa već se za svaku klasu ili strukturu automatski kreira vanjsko sučelje te klase ili strukture. Budući da su klase i strukture u Swiftu jako slične po funkcionalnosti, Apple je uveo opći pojam instance koji se odnosi na instancu klase, ali također i na instancu strukture. Glavna razlika između klase i strukture je ta da je klasa tip koji se prenosi po referenci (eng*. reference type*), a struktura je tip koji se prenosi po vrijednosti (kopiranjem, eng. *value type*).

Klase i strukture u Swiftu imaju mnogo dodirnih točaka, oboje mogu:

- definiranje **svojstava** za spremanje vrijednosti

- definiranje **metoda** za pružanje funkcionalnosti

- definiranje **konstruktora** (Appleov vokabular - initializers)

- slijede **protokole** kako bi pružili standardnu funkcionalnost

Ono što imaju klase, a ne strukture je sljedeće:

**- nasljeđivanje**

- typecasting (dinamički polimorfizam – prevazilaženje metode u nasljeđenoj klasi)

- definiranje destruktora (Appleov vokabular - deinitializers)

- pridruživanje po referenci

Nadalje, nakon malo teorije slijedi primjer deklaracije klase te instanciranja objekta.



|  |
| --- |
| class Lik { |
| var brojStranica = 0 |
| var name: String |
| init (name: String){ //konstruktor |
| self.name = name //self = this |
| } |
| func opis() -> String { |
| return "Lik s \(brojStranica) stranica." |
| } |
| } |
| var kvadrat = Lik(name: "kvadrat") // nema new |
| kvadrat.brojStranica=4 |
| var opis = kvadrat.opis() |

}

**Kod: Klasa (predložak za kod unutar vježbi)**

Ono što jest novo u odnosu na druge jezike je definicija konstruktora ključnom riječi *init* te umjesto this korištenje operatora self (referenca na objekt ove klase). Neki autori, pak preporučaju izbjegavanje korištenja operatora self pošto Swift ne treba self kako bi pristupio objektovim svojstvima/metodama već samo korištenje istog kod konstruktora i kod referenciranja svojstava unutar closure-a. Kod konstruktora – za diferenciranje između argumenata konstruktora i svojstava klase kao što je to u prethodnom primjeru napravljeno. Kod closure-a unutar klase, također, radi čitljivosti prilikom referenciranja svojstava klase. Inače, pisanje tipova (klasa, struktura, enumeracija) se preporuča UpperCamelCase-om dok se za pisanje metoda/funkcija te svosjstava/varijabli/konstanti preporuča lowerCamelCase. Za principe stila pisanja u Swiftu, što raditi, a što ne (nepisana pravila – konvencije, kao što npr. Java/android imaju svoje konvencije) preporučamo da prolistate po ovom kratkom, ali [zanimljivom članku](https://github.com/raywenderlich/swift-style-guide#use-of-self).

Nadalje, što se tiče operatora pristupa oni su također na neki način revolucionarni u odnosu na druge programske jezike. U Swiftu su isti bazirani na konceptu **modula i izvornih datoteka.** Naime, za module u Swiftu za sada ćemo reći da su jedinice distribucije, odnosno okvir/dio aplikacije koji se bilda i distribuira kao samostalna jedinka i može se uključiti riječju import. Modul se sastoji od izvornih datoteka pa su tako operatori pristupa klasama/enumeracijama/strukturama i ostalim tipovima definirani na sljedeći način:

**Private** – entiteti su dostupni/vidljivi unutar iste izvorne datoteke

**Internal** – entiteti su dostupni/vidljivi bilo kojoj izvornoj datoteci koja je *unutar* istog modula, ovo je ujedno i inicijalni (eng. *default*) slučaj

**Public** – dostupni iz svih modula (pa tako i izvornih datoteka) aplikacije

*Getter* i *setter* funkcije za neko svojstvo su iste kao u drugim jezicima, no tu imamo novine, odnosno funkcije slušače (eng. *observers*) za postavljanje vrijednosti atributa, koji su po sintaksi slični setter-u ali se pokreću prije (*willSet*) ili poslije (*didSet*) postavljanja (set-a) varijable. Unutar tih slušača moguće je koristiti *newValue* i *oldValue* ugrađene varijable koje služe za dohvaćanje nove/stare vrijednosti ovisno gdje nam treba (prije/poslije postavljanja). Koncepti OOP paradigme su sačuvani u Swiftu te je moguće vidjeti par njih, kao i prethodno spomenute slušače na sljedećem primjeru:



|  |
| --- |
|  |
| class Kvadrat: Lik { |
| var duljinaStranice: Double { |
| willSet { |
| println("Stara vrijednost je \(duljinaStranice), nova vrijednost je \(newValue)"); |
| } |
| //vrijednost je settana |
| didSet { |
| println("Stara vrijednost je \(oldValue), nova vrijednost je \(duljinaStranice)"); |
| } |
|  |
| } |
| init(duljinaStranice: Double, ime: String){ |
| self.duljinaStranice=duljinaStranice |
| super.init(name: ime) |
| brojStranica = 4 |
| } |
| var opseg: Double { |
| get { //primjer gettera |
| return duljinaStranice \* 4 |
| } |
| } |
| func povrsina()-> Double { |
| return duljinaStranice \* duljinaStranice |
| } |
| override func opis() -> String { |
| return "Kvadrat sa stranicama duljine \(duljinaStranice)" |
| } |
|  |
| let test = Kvadrat (duljinaStranice: 5.9, ime: "Kvadratko") |
| test.povrsina() |
| test.opis() |
| test.opseg |
| test.duljinaStranice = 8 |

**Kod: Klasa (nasljeđivanje)**

Dakle, u prošlom primjeru je prikazano nasljeđivanje klasa čija je sintaksa ista kao i u jezicima baziranim na c jeziku (klasa – dvotočka - klasa). Također prikazan je polimorfizam (dinamički/runtime) koji je vidljiv u nadjačavanju funkcije opis koja je prvotno definirana u baznoj klasi. Također, primjetiti kod kreacije objekta, ukoliko imate konstruktor, morate eksplicitno navesti imena argumenata. Nadalje, na svojstvu duljinaStranice je prikazan primjer prethodno spomenutih slušača willset/didSet gdje se koriste ugrađene varijable *newValue* i *oldValue*.

U Swiftu, osim strukttura kao vrijednosnih tipova postoje enumeracije koje su drugačije nego kod c-nalik jezika gdje je vrijednost enumeracije bila izražena slovčano a „sirova vrijednost“, odnosno ona prava vrijednost je bio neki broj (enumeracije = *po****broj****enja*). Ta sirova vrijednost u Swiftu može biti bilo što, npr. string. Slijedi primjer enumeracije:



|  |
| --- |
|  |

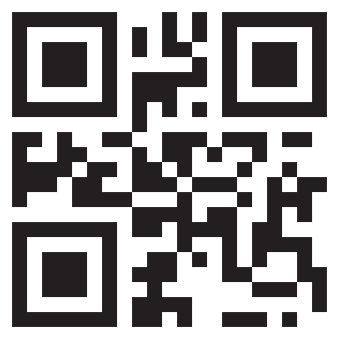
|  |
| --- |
| enum Planet { |
| case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune |
| } |
|  |
| var planet = Planet.Earth //var planet je tipa Planet |
|  |
| switch planet { |
| case .Earth: print ("Pozdrav Zemljanine") |
| default: print ("Pozdrav Izvanzemaljče o.O") |
| } |

**Kod: Enumeracija**

Najjednostavniji način za određivanje vrijednosti instance neke enumeracije (budući da podvarijanti pobrojenja u većini slučajeva ima poveći broj) jest selekcija *switch*. Primijetite da se unutar case-a switcha koristi objektna notacija kako bi se doznala specifična vrijednost. Pogledajmo primjer barkoda za proizvode. Najčešće korišteni tipovi barkoda su: UPC – A format koji koristi 4 broja, gdje je prvi broj sistemski, drugi je proizvođačev, treći je proizvodov kod te četvrti broj je broj provjere kod skeniranja te 2D QR kodovi koji enkodirani daju string maksimalne duljine 2953 znakova (po ISO standardu).



Slika 7: UPC-A kod



Slika 8: QR Code



|  |
| --- |
|  |

|  |  |
| --- | --- |
| enum Barcode { | |
| case UPCA(Int, Int, Int, Int) //signature (definicija) za UPCA | |
| case QRCode(String) //signature za QRCode | |
| } | |
|  | |
| var productBarcode = Barcode.UPCA(8, 85909, 51226, 3) | |
| productBarcode = .QRCode("ABCDEFGHIJKLMNOP") /\*postaje novi barkod tipa QRCode (implicitno se zna da je iz enum Barcode - zbog prijašnje dodjele vrijednosti) \*/ | |
|  | |
|  | |
| switch productBarcode { | |
| case .UPCA(let numberSystem, let manufacturer, let product, let check): | |
| print("UPC-A: \(numberSystem), \(manufacturer), \(product), \(check).") | |
| case .QRCode(let productCode): | |
| print("QR code: \(productCode).")  } | |
| //c-like enumeracija |
|  |
| enum PlanetRaw: Int { |
| case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune |
| } |
|  |
| var rawPlanet = PlanetRaw.Mars.rawValue |
| switch rawPlanet{ |
| case 4 : print("Pao si s Marsa") |
| default: print("Pozdrav ne-Maršanine") |
| } |

**Kod: Enumeracija – Barkod**

Pažljivi čitatelj će primijetiti da ovdje selekcija switch ne sadrži inicijalni (*default*) slučaj pošto su pokrivene sve moguće opcije (Swift na temelju tipa enumeracije zaključuje da su pokriveni sve varijante iste, eng*. exhaustive switch*). Ovdje se unutar slučajeva također koristi razlaganje instance enumeracije objektnom notacijom ali uz ispis njihovih vrijednosti koje obilježuju određenu varijantu enumeracije, npr. za QRCode se ispisuje string. Također, ispod toga je priložen kod za tipičnu *c-like* numeraciju.

Još jedan tip u Swiftu koji će se obraditi su protokoli koji su zapravo ekvivalentni sučeljima u Javi. U protokolu se definiraju sva svojstva i metode koje će klasa/struktura implementirati. Opaska: kada se radi sučelje za strukturu tada se ispred definicije metoda mora staviti ključna riječ *mutating* (modificiranje varijabli strukture iz metode - nakon povratka iz metode varijabla strukture je izmjenjena, podsjetnik - struktura je vrijednosni tip), ukoliko se to ne uradi Swift će vratiti grešku. Kod klasa to nema veze pošto se kod njih svojstva prenose po referenci.



|  |
| --- |
|  |

|  |
| --- |
| protocol Schteker { |
| var opis: String {get} |
| mutating func podesi() // mutating ispred func - za modificiranje svojstva |
| } |
| class Klasa: Schteker { |
| var opis: String = "Jednostavna klasa koja" |
| func podesi(){ |
| opis += " slijedi protokol (conforms to /adopts protocol)" |
| } |
| } |
|  |
| struct Struktura: Schteker { |
| var opis: String = "Jednostavna struktura koja" |
| mutating func podesi(){ //glavna razlika u odnosu na klasu |
| opis += " slijedi protokol (conforms to /adopts protocol)" |
| } |
| } |

**Kod: Primjer protokola**

**Korisni linkovi**

Ukoliko smatrate da niste putem ove vježbe usvojili sve točke ishoda učenja, slijede korisni linkovi:

* <https://developer.apple.com/library/prerelease/mac/documentation/Swift/Conceptual/Swift_Programming_Language/>
* <https://github.com/raywenderlich/swift-style-guide>
* <https://www.airpair.com/swift/complete-guide-to-swift>

1. ORM

**SAŽETAK**

Nakon što smo se upoznali sa razvojnim okruženjem i programskim jezikom Swift nastaviti ćemo sa potrebnim nam Dependency Managerom - Cocoapods i ORM sustavom za IOS/Swift - RealmSwift.

**KLJUČNE RIJEČI**

ORM, RealmSwift, Cocoapods, Target

**POTREBNA PREDZNANJA**

Poznavanje razvojnog okruženja Xcode i osnove objektno orijentiranog programiranja u Swiftu. Osim toga poželjno je poznavanje osnova relacijskih baza podataka te principa rada ORM sustava.

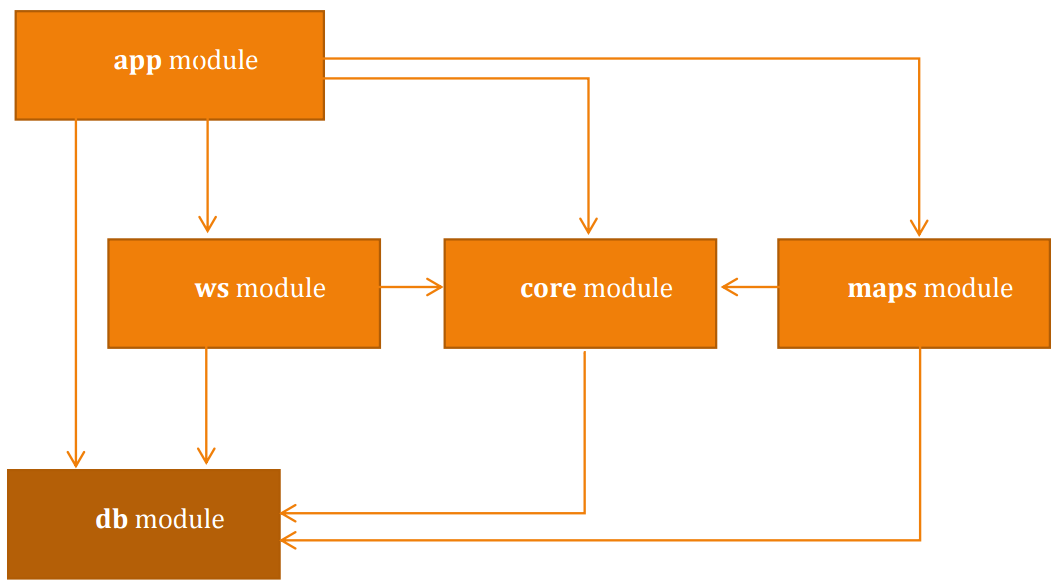
**ISHODI UČENJA**

Nakon ove vježbe moći ćete primjeniti ORM sustav RealmSwift u svojim projektima te će vam to znatno olakšati posao oko baze. Osim toga, upoznati ćete se sa Dependency Managerom *Cocoapods* putem kojeg možete dodavati third-party biblioteke. Nakon toga ćemo imati temelje za implementaciju komunikacije Web Servisa s bazom i sinkronizaciju lokalne i vanjske (eng. *remote*) baze podataka.

**RAZRADA**

* 1. Arhitektura aplikacije

Arhitektura naše aplikacije uključuje nekoliko međusobno povezanih modula koji su prikazani na sljedećoj slici.

****

**App modul** – predstavlja glavni modul koji će integrirati sve ostale module u cjelinu

**Ws modul** – web servis koji ispunjava infrastrukturalne zahtjeve, povezan je sa Core modulom i DB modulom

**Map modul** – ispunjava funkcionalne zahtjeve odnosno realizaciju mapa.

**Core modul** – služi kao posrednik u komunikaciji između modula

**DB modul** – dohvaća i pohranjuje podatke iz/u bazu

Moduli se u Xcode-u nazivaju 'Target', ali mi ćemo koristiti naziv modul.

* 1. Instalacija Cocoapods i podešavanje RealmSwifta

Kako bi započeli rad sa Swift ORM-om (eng. Object-relational mapping) potrebno je naučiti što je Cocoapods.

Cocoapods je dependency manager za Swift i Objective-C projekte. Cocoapods ima preko deset tisuća biblioteka koje olakšavaju razvoj aplikacije. Cocoapods je izgrađena na Ruby-ju pa zato i koristimo Ruby naredbu prilikom same instalacije.

Otvorimo konzolu i napišimo sljedeću naredbu kojom ćemo instalirati Cocoapods:

$ sudo gem install cocoapods

Sada kada smo instalirali Cocoapods možemo krenuti na instalaciju i konfiguriranje ORM-a – RealmSwift.

Budući da objektno-orijentirano programski jezici poput Swifta prikazuju entitete na prirodan način kao objekte sa svojim svojstvima i ponašanjima potrebno je rješenje koje će podatke iz relacijskih baza podataka pretvoriti u objekte. Upravo ta različitost uzrokuje problem pretvorbe podataka iz jednog pristupa u drugi te se kao rješenje nameće ORM.

Kako bismo podesili zavisnosti RealmSwifta u naš projekt potrebno je u konzoli otvoriti folder našega projekta:

$ cd „Documents/Discount Locator“

Nakon toga ćemo unutar trenutnoga foldera kreirati tekstualnu datoteku koja će se zvati *'podfile'* te ju otvoriti sa text editorom (npr. nano) te u nju spremiti sljedeće:

use\_frameworks!

pod 'RealmSwift'

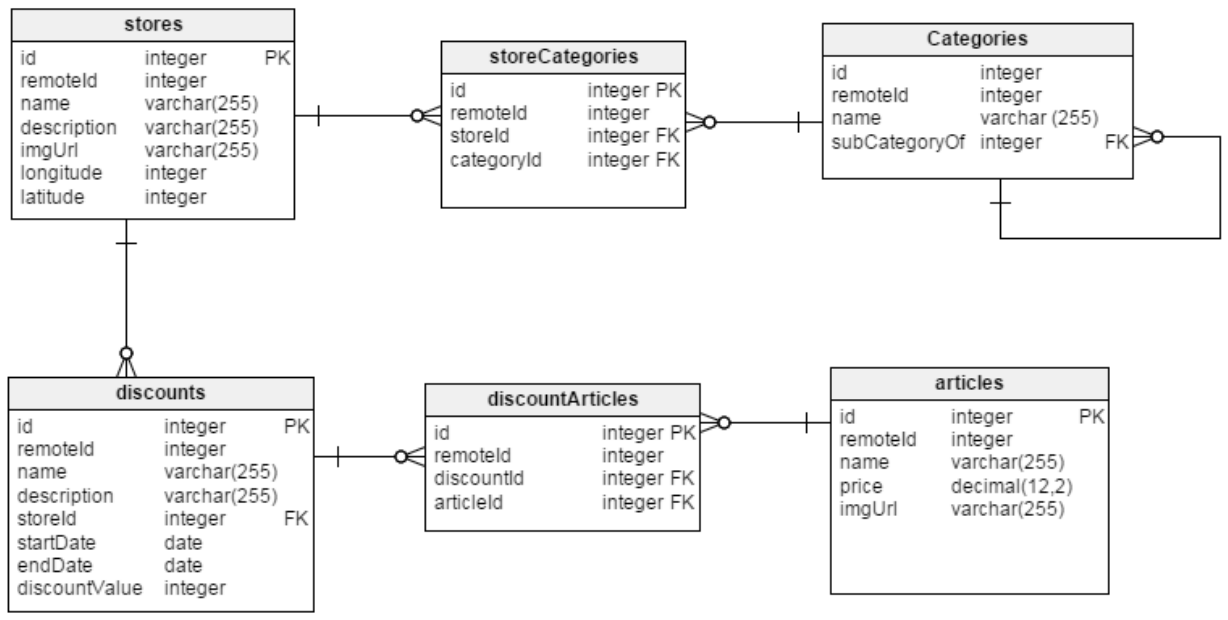
**Podfile** je tekstualna datoteka u kojoj 'opisujemo' koji kod odnosno ovisnosti želimo povući u projekt. Ovisnosti su sada spremne za instalaciju pa u konzoli pokrenemo sljedeću naredbu

$ pod install

Uspješno smo instalirali RealmSwift sada je samo potrebno pokrenuti .xcworkspace datoteku koju je Cocoapods generirala u naš folder te možemo nastaviti sa projektom.

1. **Definiranje entiteta i kreiranje klasa**

Sljedeća slika prikazuje ERA model naše aplikacije:



U nastavku ćemo kreirati klase koje predstavljati entitetne klase.

1. Prije kreiranja klasa kreirati ćemo novi modul i nazvati ćemo ga '*db*' modul. Nove module dodajemo tako da odemo na File 🡪 New 🡪 Target
2. Dodati sljedeće klase u db module: *Store, Discount, Article, DiscountArticle, Category, StoreCategory, DbController*. Klasu dodajemo desnim klikom na modul, odaberemo New 🡪 Swift File. Sve klase extendaju *Object* klasu koja je implementirana u RealmSwiftu. Paralelno tome u ActiveAndroidu se extenda klasa *Model* prilikom definiranja objekata tablice. Unutar file-a od navedenih klasa potrebno je dodati import RealmSwift
3. Dodati atribute klase

Trenutno nećemo implementirati metode kojima ćemo unositi podatke u lokalnu bazu već ćemo to ostaviti za sljedeću vježbu kada budemo izrađivali komunikaciju i primanje podataka sa web servisa.





|  |
| --- |
|  |
| import Foundation |
|  |
| import RealmSwift |
|  |
| class Store: Object |
| { |
| dynamic var remoteId: String = "" |
|  |
| dynamic var name: String = "" |
| dynamic var desc: String = "" |
| dynamic var imgUrl: String = "" |
|  |
| dynamic var longitude: Int = 0 |
| dynamic var latitude: Int = 0 |
|  |
| var discounts: [Discount] |
| { |
| return linkingObjects(Discount.self, forProperty: "store") |
| } |
| } |

**Kod: Primjer entity klase - Store**

Dakle , iz prethodnog koda moguće je zaključiti kako se importa neki vanjski modul, u ovom slučaju RealmSwift – pomoću ključne riječi import.

Klasa Store, kao što je već rečeno, mora nasljeđivati klasu Object te je potrebno koristiti ključnu riječ dynamic prilikom definiranja atributa. Primjetite kako se unutar klase Store drži lista popusta koji su zapravo svojstvo koje se dinamički vraća prilikom dohvaćanja atributa. Prilikom definiranja RealmSwifta ne može se koristiti ključna riječ get (primjetite kako je ovo zapravo getter).



|  |
| --- |
|  |
| import Foundation |
| import RealmSwift |
| class Discount: Object |
| { |
| dynamic var remoteId: String = "" |
| dynamic var name: String = "" |
| dynamic var store: Store? |
|  |
| func storeSet(s: Store) |
| { |
| self.store = s |
| } |
| } |

**Kod: Primjer entity klase - Discount**

Na isti način je kreirana i klasa Discount koja u sebi osim njenih svojstvenih atributa sadrži i atribut Store koji predstavlja jednu trgovinu u kojoj se taj popust nalazi (veza 1-više).

1. Web servis

**SAŽETAK**

U ovoj vježbi će biti opisan rad sa web servisima putem frameworka – Alamofire. Osim toga, parse-ati ćemo dohvaćene podatke sa web servisa pomoću SwiftyJSON frameworka.

**KLJUČNE RIJEČI**

Alamofire, JSON, web servis, SwiftyJSON

**POTREBNA PREDZNANJA**

Način dodavanja frameworka pomoću Cocoapods dependency managera. Osnovno razumijevanje POST/GET zahtjeva i odgovora.

**ISHODI UČENJA**

Nakon ove vježbe naučiti ćete koristiti Alamofire framework kako bi dohvaćali podatke sa određenog web servisa te parse-anje podataka pomoću SwiftyJSON-a. Upoznati ćemo se sa protokolima u Swiftu te vidjeti način njihove implementacije.

**RAZRADA**

Ovu vježbu ćemo podijeliti u četiri koraka:

1. Dodavanje i implementacija HTTPRequest klase (ws modul)
2. Dodavanje i implementacija DataLoader i JsonAdapter klase (core modul)
3. Dodavanje i implementacija WebServiceDataLoader klase (Discount Locator modul)
4. Prikazivanje dohvaćenih podataka na korisničkom sučelju
   1. Dodavanje i implementacija HTTPRequest klase (ws modul)

Kako bi integrirali Alamofire (napomena: Alamofire nije podržan u iOS 7) u naš Xcode projekt koristeći Cocoapods dodati ćemo sljedeće u podfile tekstualnu datoteku.

source 'https://github.com/CocoaPods/Specs.git'

platform :ios, '8.0'

use\_frameworks!

pod 'RealmSwift'

pod 'Alamofire', '~> 3.0'

pod 'Alamofire', '~> 3.0' – ova linija označava da je 3.0 minimalna verzija Alamofite frameworka koja će se importati u naš projekt.

Nakon toga ćemo pokrenuti naredbu

$ pod install

Sljedeći korak je kreiranje novog modula kojeg ćemo nazvati '*ws*'. (File 🡪 New 🡪 Target) Zatim ćemo kreirati novu klasu HTTPRequest koja će imati metodu za poziv prema servisu. Prije implementacije klase HTTPRequest ćemo konfigurirati određene postavke.

Alamofire zahtjev se izvršava asinkrono. To bi značilo da se specifira callback funkcija koja upravlja zahtjevom nakon što je on primljen te je time izbjegnuto blokiranje daljnjeg izvršavanja programa. Dakle, u pozadini Alamofire kreira novu dretvu koja čeka na dolazak podataka te onda okida callback funkciju. Možete primjetiti sličnost sa npr. jQuery Ajax zahtjevom prema web servisu.

Budući da šaljemo zahtjev prema nesigurnoj adresi (http) moramo promijeniti Info.plist unutar Discount Locator modula. Info.plist (eng. Property list file) je xml datoteka u kojoj su definirane postavke koje vrijede za određeni modul. Korijenski element xml datoteke je dict. Sustav koristi ključ-vrijednost kako bi definirao informacije o postavkama modula.

Desni klik na Info.plist 🡪 Open as 🡪 Source Code daje prikaz xml datoteke koju ćemo urediti.

Prije zatvaranja korijenskog xml elementa *dict* ćemo dodati sljedeći komad teksta:

<key>NSAppTransportSecurity</key>

<dict>

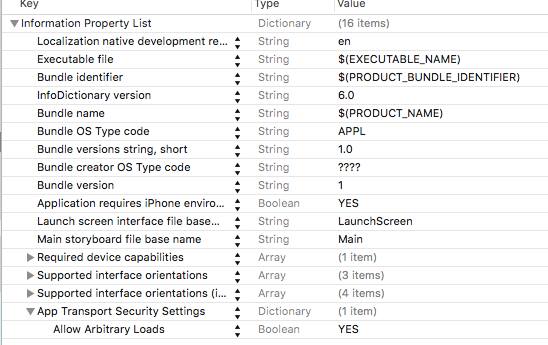
<key>NSAllowsArbitraryLoads</key>

<true/>

</dict>

Napomena: dodavanje prethodne postavke u **Info.plist** se inače ne preporuča jer time omogućujemo zahtjeve prema nesigurnim adresama što u većini slučajeva ne želimo.

Tako dodana nova postavka se može pregledati kao *source code* što smo prethodno i radili, ali također ju možemo pregledati i izmijeniti pomoću *property liste* (desni klik na isti info.plist 🡪 Open as 🡪 Property list)



Sada kada smo konfigurirali postavke možemo krenuti na implementiranje klase HTTPRequest koja će izgledati ovako:



|  |
| --- |
|  |
| import Alamofire |
| import SwiftyJSON |
|  |
| public class HTTPRequest |
| { |
| public static var sharedWSInstance = HTTPRequest() |
|  |
| public func httprequest(url: String, params: [String:String], completion: (result: AnyObject) -> Void) |
| { |
| Alamofire.request(.POST, url) |
| .responseJSON { response in |
| if let json = response.result.value{ |
|  |
| completion(result: json) |
|  |
| } |
| } |
| } |
| } |

**Kod: Klasa HTTPRequest**

Klasa HTTPRequest će nam biti singleton objekt prema design patternu. HTTPRequest klasa sadrži metodu httprequest koja kao argumente prima url prema kojemu šaljemo zahtjev te closure funkciju completion. Iz koda možemo primjetiti da je jednostavno pozivati naš servis pomoću Alamofire-a - funkciji Alamofire.request ćemo proslijediti tip zahtjeva kao prvi argument i url kao drugi te je moguće dodati parametre zahtjevu kao treći argument.

Alamofire ima built-in metode odgovora:

* response()
* responseData()
* responseString(encoding: NSStringEncoding)
* responseJSON(options: NSJSONReadingOptions)
* responsePropertyList(options: NSPropertyListReadOptions)

Mi ćemo koristiti responseJSON metodu jer dobivamo json kao odgovor na naš zahtjev.

* 1. Dodavanje i implementacija DataLoader i JsonAdapter klase (core modul)

Prije nego krenemo sa realizacijom klase koja će pozivati metodu za dohvaćanje podataka potrebno je implementirati klase za parse-anje json-a te kreirati protokol (interface u drugim oop programskim jezicima).



|  |
| --- |
|  |
| import Foundation |
| import db |
| public protocol DataLoader |
| { |
| var stores:[Store]{get set} |
| var discounts:[Discount]{get set} |
| func LoadData() |
| } |

**Kod: protokol DataLoader**

Protokol se definira sa ključnom riječi '*protocol*'. Mi smo ga definirali kao javni jer ga implementiramo u drugom modulu. Prije nego krenemo sa implementacijom potrebno je dodati *db* modul iz razloga što ćemo koristiti tipove podataka iz navedenoga modula (import db). Klasa koja će implementirati ovaj protokol morati će implementirati dvije varijable i jednu funkciju odnosno polje *Stores* i polje *Discounts* te funkciju *LoadData*.



Proučite načine rada **protokola** na sljedećoj poveznici:

<https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html>

Sljedeća klasa je JsonAdapter koja će nam služiti za parse-anje dohvaćenih podataka. Prije nego krenemo sa njezinom implementacijom potrebno je podesiti framework koji ćemo koristiti za parse-anje json-a – SwiftyJSON. Otvorimo podfile te na kraj dodamo sljedeće:

pod 'SwiftyJSON', :git => 'https://github.com/SwiftyJSON/SwiftyJSON.git'

Nakon toga u terminalu u mapi u kojoj se nalazi podfile izvršimo naredbu:

$ pod install

Instalacija SwiftyJSON-a je identična instalaciji frameworka Alamofire koju smo opisali u prvom koraku ove vježbe.

Nakon instalacije SwiftyJSON-a možemo implementirati našu JsonAdapter klasu.



|  |
| --- |
|  |

|  |
| --- |
| import Foundation |
| import SwiftyJSON |
| import db |
| public class JsonAdapter |
| {  //… metoda getStores i getDiscounts |
| } |

**Kod: Klasa JsonAdapter**



|  |
| --- |
|  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | public static func getStores(json: AnyObject) -> Array<Store> | | { | | var stores = [Store]() | | let jsonStores = JSON(json) | |  | | var items = jsonStores["items"] | | var itemsString = String(items) | |  | | itemsString = itemsString.stringByReplacingOccurrencesOfString(String(Character(UnicodeScalar(92))), withString: "", options: NSStringCompareOptions.LiteralSearch, range: nil) | |  | | if let dataFromString = itemsString.dataUsingEncoding(NSUTF8StringEncoding, allowLossyConversion: false) { | | let json2 = JSON(data: dataFromString) | | for (key, subJson) in json2 { | | var s:Store = Store() | | s.remoteId = Int(String(subJson["id"]))! | | s.name = subJson["name"].string! | | s.desc = subJson["description"].string! | | s.imgUrl = subJson["imgUrl"].string! | | s.latitude = Float(String(subJson["latitude"]))! / 1000000 | | s.longitude = Float(String(subJson["longitude"]))! / 1000000 | | print("lat: ", s.latitude) | | print("lng: ", s.longitude) | | stores.append(s) | | } | | } | | return stores | | } | |
|  |
|  |
|  |
|  |
|  |
|  |

**Kod: Metoda getStores klase JsonAdapter**



|  |
| --- |
|  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | public static func getDiscounts(json: AnyObject) -> Array<Discount> | | { | | var discounts = [Discount]() | | let jsonStores = JSON(json) | |  | | let items = jsonStores["items"] | | var itemsString = String(items) | |  | | itemsString = itemsString.stringByReplacingOccurrencesOfString(String(Character(UnicodeScalar(92))), withString: "", options: NSStringCompareOptions.LiteralSearch, range: nil) | |  | | if let dataFromString = itemsString.dataUsingEncoding(NSUTF8StringEncoding, allowLossyConversion: false) { | | let json2 = JSON(data: dataFromString) | | for (key, subJson) in json2 { | |  | | var d:Discount = Discount() | | d.remoteId = Int(String(subJson["id"]))! | | d.name = subJson["name"].string! | | d.desc = subJson["description"].string! | | d.discount = Int(String(subJson["discount"]))! | | d.endDate = subJson["endDate"].string! | | d.startDate = subJson["startDate"].string! | | d.storeId = Int(String(subJson["storeId"]))! | | discounts.append(d) | | } | | } | | return discounts | | } | |
|  |
|  |
|  |
|  |
|  |
|  |

**Kod: Metoda getStores klase JsonAdapter**

Klasa JsonAdapter sadrži dvije statičke metode – *getStores* i *getDiscounts.* Gore je navedena implementacija *getStores* metode koja će vraćati polje trgovina i implementacija *getDiscounts* metode koja će vraćati polje. Prvo ćemo iskoristiti metodu iz frameworka SwiftyJSON – *JSON* koja *AnyObject* pretvara u JSON objekt. Nakon toga ćemo iterirati kroz polje *items* te ujedno kreirati objekt *Store* te ga puniti podacima iz JSON-a (*items*) te dodavati na kraj polja trgovina (*append*).

* 1. Dodavanje i implementacija WebServiceDataLoader klase

Spremni smo za kreiranje i implementaciju klase WebServiceDataLoader modula DiscountLocator.



|  |
| --- |
|  |
|  |
| import Foundation |
| import db |
| import core |
| import ws |
|  |
| public class WebServiceDataLoader:DataLoader |
| { |
| public var stores: [Store] = [] |
| public var discounts: [Discount] = [] |
| public var storesTableView: UITableView? |
| private var discountsLoaded: Bool = false |
| private var storesLoaded: Bool = false |
| private var prefs = NSUserDefaults() |
| public func LoadData() { |
| var params:[String:String] = ["method": "getAll"] |
| if(NetConnection.Connection.isConnectedToNetwork() && prefs.boolForKey("EnableWebService")){ |
| print("loadam s web servisa") |
| HTTPRequest.sharedWSInstance.httprequest("http://cortex.foi.hr/mtl/courses/air/stores.php", params: params) |
| { |
| (result: AnyObject) in |
| self.stores = JsonAdapter.getStores(result) |
| self.storesLoaded = true |
| self.showLoadedData() |
| } |
| HTTPRequest.sharedWSInstance.httprequest("http://cortex.foi.hr/mtl/courses/air/discounts.php", params: params) |
| { |
| (result: AnyObject) in |
| self.discounts = JsonAdapter.getDiscounts(result) |
| self.discountsLoaded = true |
| self.showLoadedData() |
| } |
| } |
| else { |
| print("loadam lokalno") |
| self.showDataFromLocalDB() |

}

}

**Kod: Klasa WebServiceDataLoader I. dio**

…nastavak implementacije



|  |
| --- |
|  |
|  |
| private func showLoadedData() |
| { |
| if(self.storesLoaded && self.discountsLoaded){ |
| self.bindData() |
| storesTableView?.reloadData() |
| } |
| } |
| private func showDataFromLocalDB() |
| { |
| let stores = DbController.sharedDBInstance.realmFetch(Store) |
| let discounts = DbController.sharedDBInstance.realmFetch(Discount) |
|  |
| for store in stores{ |
| self.stores.append(store as! Store) |
| } |
| for discount in discounts{ |
| self.discounts.append(discount as! Discount) |
| } |
| storesTableView?.reloadData() |
| } |
| private func bindData() |
| { |
| DbController.sharedDBInstance.realm.beginWrite() |
| DbController.sharedDBInstance.realm.deleteAll() |
| try! DbController.sharedDBInstance.realm.commitWrite() |
| for store in stores |
| { |
| print(DbController.sharedDBInstance) |
| DbController.sharedDBInstance.realmAdd(store) |
| for discount in discounts |
| { |
| if(discount.storeId == store.remoteId) |
| { |
| discount.storeSet(store) |
| DbController.sharedDBInstance.realmAdd(discount) |
| } |
| } |
| } |
| } |
| } |

**Kod: Klasa WebServiceDataLoader II. dio**

Klasa *WebServiceDataLoader* nasljeđuje (implementira ga) protokol *DataLoader*.

Klasa sadrži četiri funkcije – *LoadData* (funkcija implementirana iz protokola koja vrši zahtjev prema servisu, ukoliko je Internet nedostupan podaci se dohvaćaju iz lokalne baze), *showLoadedData* (nakon što je izvršen zahtjev ova funkcija će ispisati podatke na korisničko sučelje), *bindData* (funkcija koja povezuje objekte tipa *Discount* s objektima tipa *Store* i ujedno zapisuje podatke u lokalnu bazu) te *showDataFromLocalDB* (funkcija koja će dohvatiti podatke iz lokalne baze)

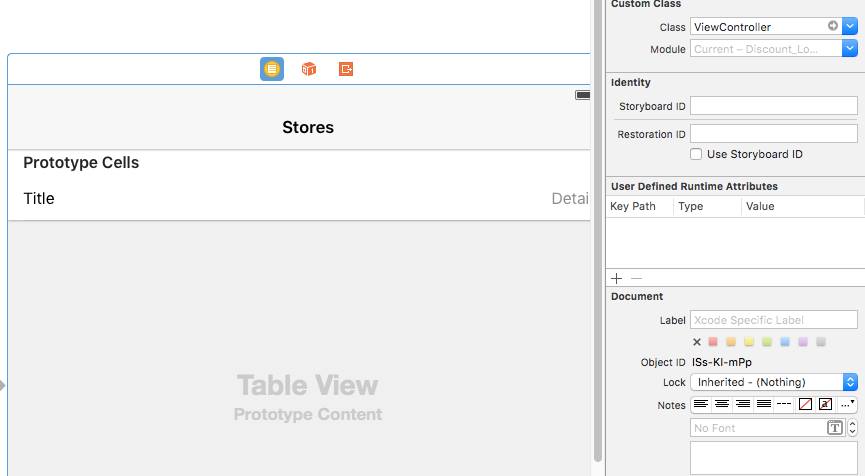


Budući da metoda httprequest iz klase HTTPRequest prima closure kao argument proučite isti na sljedećoj poveznici:

<https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Closures.html>

* 1. Prikazivanje dohvaćenih podataka na korisničkom sučelju

Dohvaćene podatke prikazati ćemo pomoću TableView-a. Kliknemo na Main.storyboard te izbrišemo trenutni viewController. Nakon toga ćemo otići u object library i povući navigation controller unutar radne površine. Kad smo to napravili pokazati će nam se Navigation controller i na njega povezan root view controller. Odaberemo root view controller te promijenimo ime klase u ViewController odnosno definiramo klasu koja će kontrolirati taj root view.



Sada odaberemo *Prototype Cells* odnosno prikaz kako će naše ćelije izgledati i pod opcijama s desne strane odaberemo *Attributes* *Inspector* i za *Style* odaberemo *Right* *Detail.* Trenutno ćemo koristiti ovaj stil ćelije gdje će se sa lijeve strane nalaziti ime trgovine, a sa desne opis. Kako budemo napredovali sa projektom tako ćemo i promijeniti stil u *Custom.*



Ukoliko pod *Prototype Cells* odabere *Title* ili *Detail* s desne strane možete uređivati stil i način prikazivanja labele. Poigrajte se sa navedenim te korigirajte opis trgovine tako da bude prikazan u više redova.

Nakon toga napraviti ćemo sljedeće promjene u klasi ViewController:



|  |
| --- |
| import UIKit |
| import db |
| class ViewController: UITableViewController { |
|  |
| @IBOutlet weak var storesTableView: UITableView! |
|  |
| var webServiceDataLoader = WebServiceDataLoader() |
|  |
| override func viewDidLoad() { |
| |  | | --- | | super.viewDidLoad() | | Realm.Configuration.defaultConfiguration = Realm.Configuration( | | schemaVersion: 3, | | migrationBlock: { migration, oldSchemaVersion in | | }) | | webServiceDataLoader.storesTableView = self.storesTableView | | webServiceDataLoader.LoadData() | | self.animationCheck() //gasi animacije pošto je initial view controller |   } |
|  |
| override func didReceiveMemoryWarning() { |
| super.didReceiveMemoryWarning() |
| // Dispose of any resources that can be recreated. |
| } |
|  |
| override func tableView(tableView: UITableView,  numberOfRowsInSection section: Int) -> Int { |
| return webServiceDataLoader.stores.count |
| } |
|  |
| override func tableView(taleView: UITableView,  cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell { |
| |  | | --- | | let cell = self.tableView.dequeueReusableCellWithIdentifier("Cell", forIndexPath: indexPath) as! StoreTableViewCell | | var store: Store | | store = webServiceDataLoader.stores[indexPath.row] | | cell.storeTitle.text = store.name | | cell.storeDescription.text = store.desc | | cell.storeImageView.kf\_setImageWithURL(NSURL(string:store.imgUrl)!) | | return cell |   } |
|  |

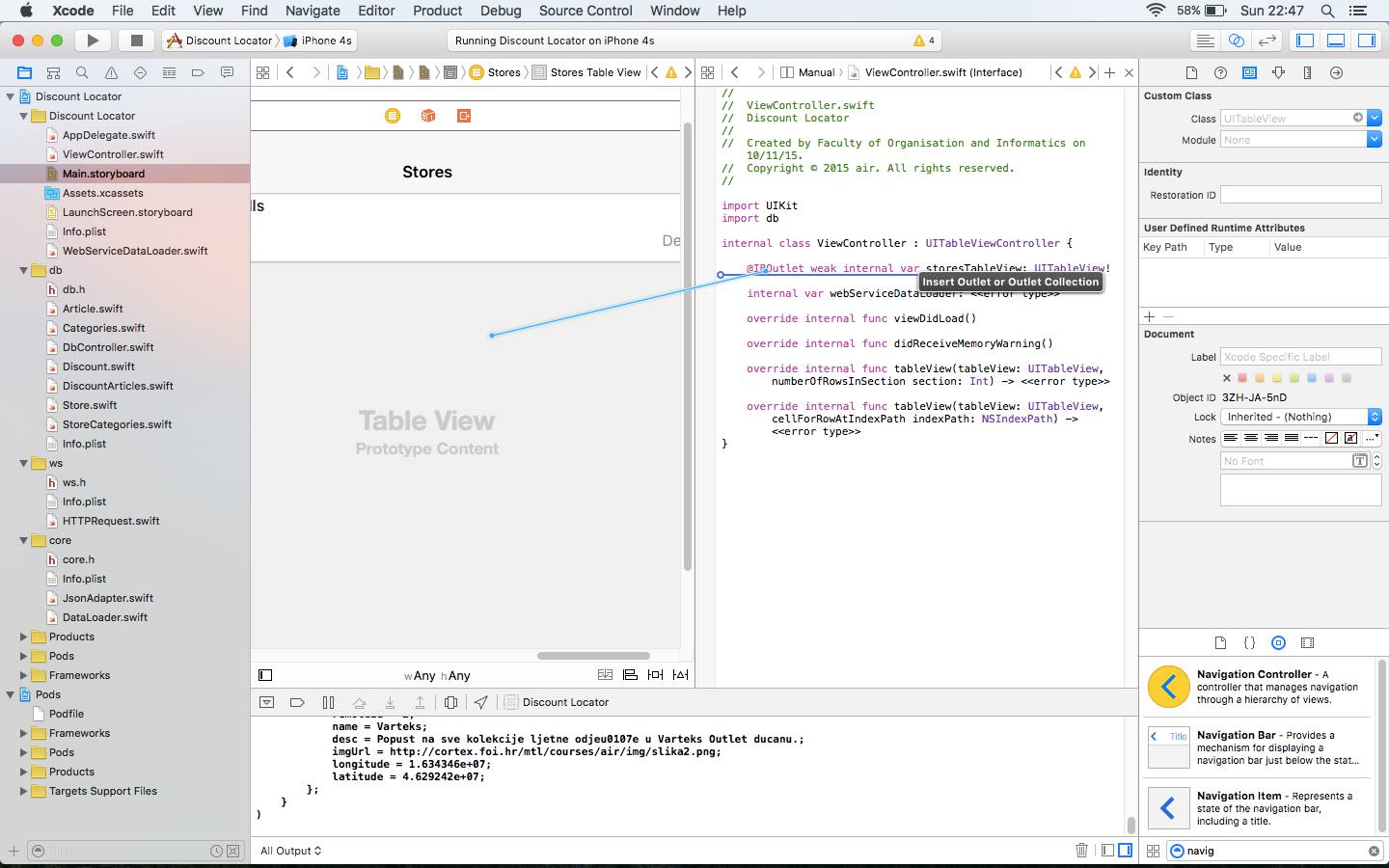
**Kod: Izmjena klase ViewController**

U prethodnom kodu smo promijenili klasu koju nasljeđuje ViewController, pa je to sada *UITableViewController*.

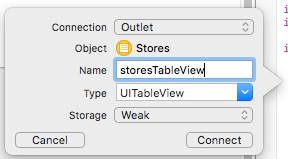
Zatim ćemo otvoriti *main.storyboarda* kako bi povukli table view unutar našeg kontrolera te ih povezali. Kliknimo na *assistant editor* odnosno na srednju ikonu sa slike:

https://scontent-vie1-1.xx.fbcdn.net/hphotos-xpa1/v/t34.0-12/12243811_10206956657908714_720195333_n.jpg?oh=e5be7a860f72ee1aa2a958a8021105f0&oe=564B54C4

Nakon toga će se sa jedne strane ekrana nalaziti ViewController, a s druge main.storyboard. Sada jednostavno uhvatimo mišem tableView i odvučemo ga u ViewController:



Nakon toga će nam se pojaviti sljedeći prozor u kojemu unesemo ime TableView-a te kao Storage odabiremo *Weak* i kliknemo na Connect. Na ovaj način smo povezali TableView sa ViewControllerom i moći ćemo mu pristupati iz koda.



Navedeni TableView ćemo iz ViewControllera ćemo dodijeliti kao varijablu klase WebServiceDataLoader koja će napuniti TableView nakon što dohvati podatke sa servisa i poveže ih.

Kako bismo popunili table view potrebno je implementirati dvije metode. U prvoj metodi definiramo broj redaka koji će se pojaviti u TableViewu, a u drugoj povezujemo podatke na ćeliju tablice.

Sljedeći korak su male promjene u *WebServiceDataLoader* klasi. U klasu ćemo dodati sljedeću liniju koda ispod ostalih varijabli:

public var storesTableView: UITableView?

te ćemo preurediti *showLoadedData* metodu da izgleda ovako:

|  |
| --- |
| private func showLoadedData() |
| { |
| if(storesLoaded && discountsLoaded) |
| { |
| self.bindData() |
| storesTableView?.reloadData() |
| } |
| } |

*storesTableView?.reloadData()* metoda osvježava podatke na TableView-u. Isprva će TableView biti prazan jer su polja *stores* i *discounts* prazna, ali nakon poziva web servisu u njih učitavamo podatke pa je potrebno osvježiti TableView.

Sada možemo build-ati projekt i naša lista sadrži podatke.

1. Prikaz podataka

**SAŽETAK**

U ovoj vježbi ćemo pokazati na koji način prilagoditi ćelije tableview-a u kojima smo prikazali podatke za dućane te ćemo implementirati tab bar za navigaciju. Osim toga dodati ćemo novi tableview controller za prikaz popusta za pojedini dućan. Pokazati ćemo kako izgraditi jednostavni UI i kako napraviti prijelaz i prijenos podataka iz jednog view-a u drugi.

**KLJUČNE RIJEČI**

TableView, Tab Bar, segue, navigacija

**POTREBNA PREDZNANJA**

Potrebno je poznavati na koji način popuniti tableview podacima, ažuriranje podfilea i instaliranje frameworka.

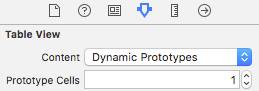
**ISHODI UČENJA**

Nakon prolaska ove vježbe student bi trebao znati spajati View Controllere te prenositi podatke između istih, implementirati Tab Bar za navigaciju i kreirati custom ćelije.

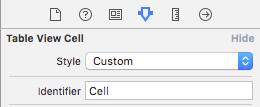
**RAZRADA**

* 1. Prilagođavanje ćelija za prikaz (custom cells)

Prvi korak ove vježbe je prilagođavanje ćelija za prikaz popisa dućana. Izmijeniti ćemo ćelije u custom ćelije tako da sadrže sliku, naziv te opis dućana. Otvorimo Main.Storyboard te u Document outline-u gdje se nalaze scene odabiremo Stores Table View te s desne strane odabiremo Attributes Inspector kako bi podesili svojstva Table View-a. Uvjerimo se da je Content postavljen na Dynamic Prototypes te da je broj Prototype ćelija postavljen na 1.



Odaberimo Cell koji se nalazi u Stores Table View-u. Nakon toga na isti način odabiremo Attributes Inspector ikonu i pod Style odabiremo Custom.



Sada možemo posložiti ćeliju kako nama odgovara pa ćemo tako doraditi ćeliju da izgleda slično kao na sljedećoj slici. S lijeve strane se nalazi Image View objekt kojeg smo dovukli iz biblioteke objekata, a s desne strane su dvije labele od kojih će nam jedna služiti za naziv dućana, a druga za opis dućana.

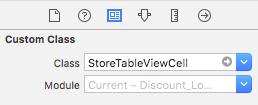


Prije nego nastavimo sa kreiranjem klase za custom ćelije moramo u podfile dodati framework KingFisher koji u Image View postavlja sliku iz navedene url putanje. Uvedimo promjene u podfile i ažurirajmo frameworke.

$ pod 'KingFisher','~> 1.8'

Potrebno je definirati klasu koja će biti zadužena za upravljanje custom ćelijom. U modulu DiscountLocator ćemo dodati novu klasu i nazvati ćemo ju StoreTableViewCell. Prilikom kreiranja klase potrebno je postaviti opciju **Subclass** of na UITableViewCell.

Nadalje, pod Utilities – pod Identity inspektor je pod klasu potrebno odabrati StoreTableViewCell klasu koju smo upravo kreirali kako bismo povezali objekt na viewu (MainStoryboard-u) s klasom koja ga kontrolira.



Kreirajmo outlete za svaki objekt koji se nalazi u našoj ćeliji. To ćemo napraviti na način da ćemo s jedne strane imati Main.Storyboard, a s druge klasu StoreTableViewCell te ćemo navedene objekte odvući u klasu. Nakon toga uvesti ćemo manje promjene u metodi tableview unutar klase ViewController koja kontrolira tablicu dućana.

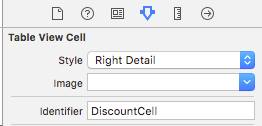


|  |
| --- |
|  |
|  |
| override func tableView(tableView: UITableView,  cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell { |
|  |
| let cell = self.tableView.dequeueReusableCellWithIdentifier("Cell",  forIndexPath: indexPath) as! StoreTableViewCell |
|  |
| var store: Store |
| store = webServiceDataLoader.stores[indexPath.row] |
|  |
| cell.storeTitle.text = store.name |
| cell.storeDescription.text = store.desc |
| cell.storeImageView.kf\_setImageWithURL(NSURL(string:store.imgUrl)!) |
|  |
| return cell |
| } |

**Kod: Izmjena metode tableview u klasi ViewController**

Primjetite da smo izmijenili castanje iz TableViewCell u StoreTableViewCell, a zatim smo dohvatili outlete unutar ćelije i dodijelili im pripadne vrijednosti. Za povezivanje slike i url putanje na Image View korištena je metoda kf\_setImageWithURL iz Frameworka KingFisher.

Sada ćemo implementirati da se korisniku pritiskom na određenu trgovinu prikažu pripadajući popusti. U Android verziji je to učinjeno pomoću expandableListView, no dotična paradigma nije nativna na IOS platformi te ju zapravo rijetko tko koristi. Ovdje se to implementira na sljedeći način. Prvo je potrebno na scenu odvući novi tableViewController iz knjižnice objekata. Kako bi izbjegli posao sa custom ćelijama u ovom slučaju ćemo primjera radi u prethodno spomenutom Attributes inspectoru odabrati Style->Right Detail te joj dodijelite identifikator.



Dodajte novu klasu koja nasljeđuje UITableViewController te je nazovite DiscountsViewController. Kod klase se nalazi u nastavku.



|  |
| --- |
| class DiscountsViewController: UITableViewController { |
|  |
| var discounts: [Discount]? |
|  |
| //... |
| override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int { |
|  |
| return discounts!.count |
|  |
| } |
|  |
| override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell { |
|  |
| let cell = self.tableView.dequeueReusableCellWithIdentifier("DiscountCell", forIndexPath: indexPath) as UITableViewCell |
|  |
| cell.textLabel!.text = self.discounts![indexPath.row].name |
| cell.detailTextLabel!.text = self.discounts![indexPath.row].desc |
|  |
| return cell |
| } |

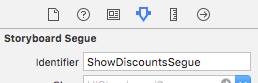
**Kod: klasa DiscountViewController**

Sve je identično kao i kod tableViewControllera kojeg smo objasnili kod klase Store. Radi se o metodama koje dinamički generiraju ćelije. Potrebno je samo primjetiti polje discounts u koje ćemo iz glavnog viewControllera proslijediti popis popusta za određenu trgovinu.

Potrebno je odabrati ćeliju trgovine te pazite da je lijevo u izborniku DocumentOutline označen ContentView te ćelije (kao na slijedećoj slici).



Zatim je potrebno držati tipku *Cmd* te odvući liniju na TableViewController koji smo prethodno kreirali. Nadalje, primjetimo plavu liniju koja se kreirala. Ta linija se naziva *Segue* a označuje tranziciju između dva view-a (u Androidu je to intent objekt s kojim definiramo prijelaz na drugu aktivnost). Potrebno je odabrati tu liniju te joj u Attributes Inspectoru dodijeliti identifikator budući da ćemo tu tranziciju referencirati u kodu.



Sada ćemo u ViewController dodati jednu malu metodu koja je nadjačana te se poziva prije nego što se dogodi Segue odnosno prijelaz. Ta metoda će slati podatke o popustima u DiscountsViewController.



|  |
| --- |
|  |
|  |
| override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) { |
| if segue.identifier == "ShowDiscountsSegue" { |
| if let destination = segue.destinationViewController as? DiscountsViewController { |
| if let storeIndex = tableView.indexPathForSelectedRow { |
|  |
| destination.discounts = webServiceDataLoader.stores[storeIndex.row].discounts |
|  |
| } |
|  |
| } |
| } |
| } |

**Kod: Dodavanje metode PrepareForSegue**

Prvo se provjerava radi li se o dotičnom segue-u kojemu smo dodijelili identifikator ShowDiscountsSegue. Ukoliko je to istina provjeravamo radi li se o DiscountsViewControlleru zatim u storeIndex spremamo indeks odabrane ćelije i nakon toga u liniji

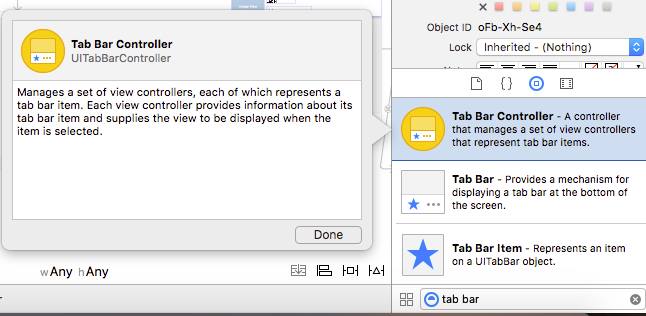
destination.discounts = webServiceDataLoader.stores[storeIndex.row].discounts

šaljemo destinaciji (DiscountsViewControlleru) predviđene popuste.

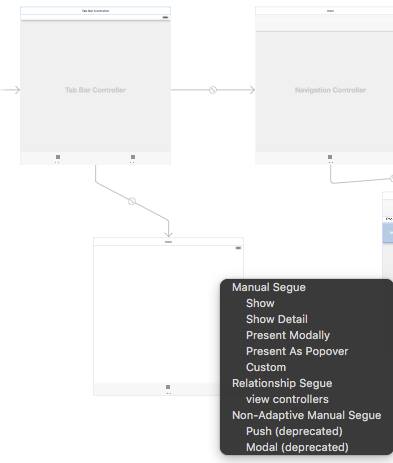
Navigaciju ćemo implementirati pomoću Tab Bar controllera. Budući da navigation drawer nije nativna opcija za navigaciju u iOS-u koristiti ćemo Tab Bar u ove svrhe.

Tab Bar controller služi za organizaciju aplikacije u više odvojenih operacija. Tab Bar se sastoji od view-a kojima direktno upravlja tab bar controller te view-a koji su upravljani sadržajem view controllera kojega mu pridodamo. U nastavku ćemo implementirati tab bar te pokazati na koji način se on koristi.

Otvorimo Main.storyboard te povucimo Tab Bar Controller iz biblioteke objekata na radnu površinu Main.storyboarda. Nakon toga će nam se prikazati Tab Bar Controller i automatski će dodati svoja dva ViewControllera. Na donjoj slici možemo vidjeti kako izgleda Tab Bar Controller u biblioteci objekata.



Obrisati ćemo oba ViewControllera tako da nam ostane samo Tab Bar Controller. Nakon što smo obrisali oba ViewControllera potrebno je prvo povezati Tab Bar Controller sa već postojećim Navigation Controller. To ćemo uraditi na način ta držimo Ctrl tipku, kliknemo na Tab Bar Controller te povučemo mišem do Navigation Controllera nakon čega će nam se pojaviti dialog box kao na sljedećoj slici.



U dialog box-u odabiremo način na koji će se prikazati view za odabrani tab. Ovdje ćemo odabrati opciju view controllers pod Relationship Segue. Nakon toga ćemo dodati novi View Controller iz biblioteke objekata te ga na isti način povezati kao u prethodnom slučaju.

Detaljnije o ostalim opcijama prijelaza može se pronaći na sljedećoj poveznici:

<https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/UsingSegues.html>

Sada možemo buildati i pokrenuti projekt te smo dobili dva taba od kojih jedan pokazuje na Table View sa popisom dućana dok je drugi prazan i njegova implementacije slijedi u sljedećoj vježbi.

1. Kreiranje izbornika

**SAŽETAK**

U ovoj vježbi kreirati ćemo izbornik odnosno navigation drawer.

**KLJUČNE RIJEČI**

Navigation drawer, Objective-C, SWRevealViewController,

**POTREBNA PREDZNANJA**

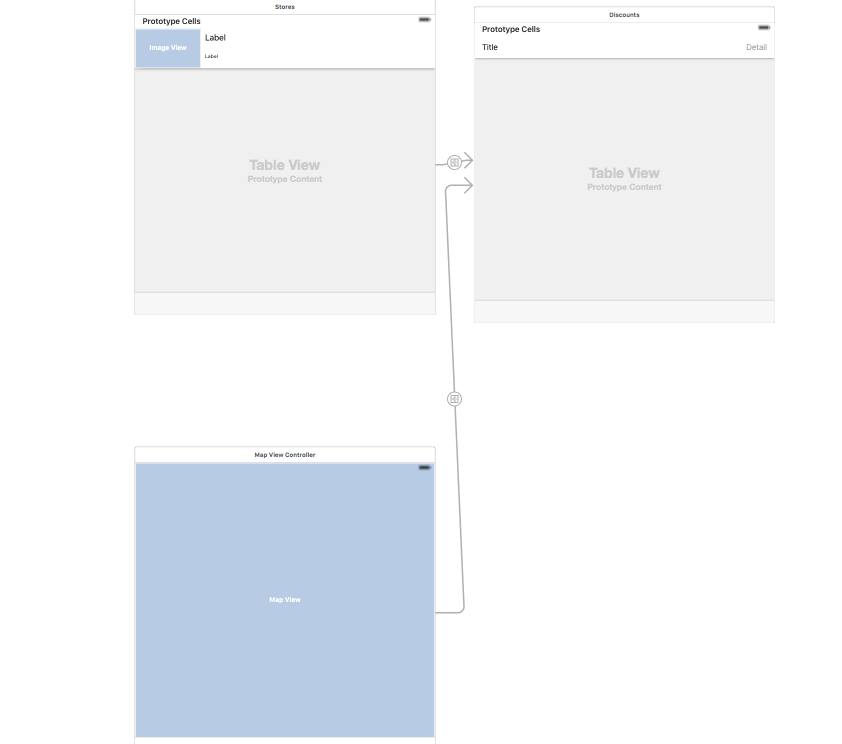
Dodavanje objekta u main.Storyboard i upravljanje segueima.

**ISHODI UČENJA**

Naučiti ćemo koristiti objective-C biblioteku i upotrijebiti ju u Swiftu.

**RAZRADA**

* 1. Dodavanje SWRevealViewControllera

Obrisati ćemo TabViewController iz prethodne vježbe i prijeći na navigation controller. Dio MainStoryboard nam sada treba izgledati ovako budući da više nećemo koristiti TabViewController: iOS nativno ne podržava ovakav oblik izbornika stoga je potrebno dodati vanjsku biblioteku. Za kreiranje navigation drawera potrebno je koristiti biblioteku - SWRevealController. SWRevealController je UIViewController podklasa za prikaz navigation drawera.

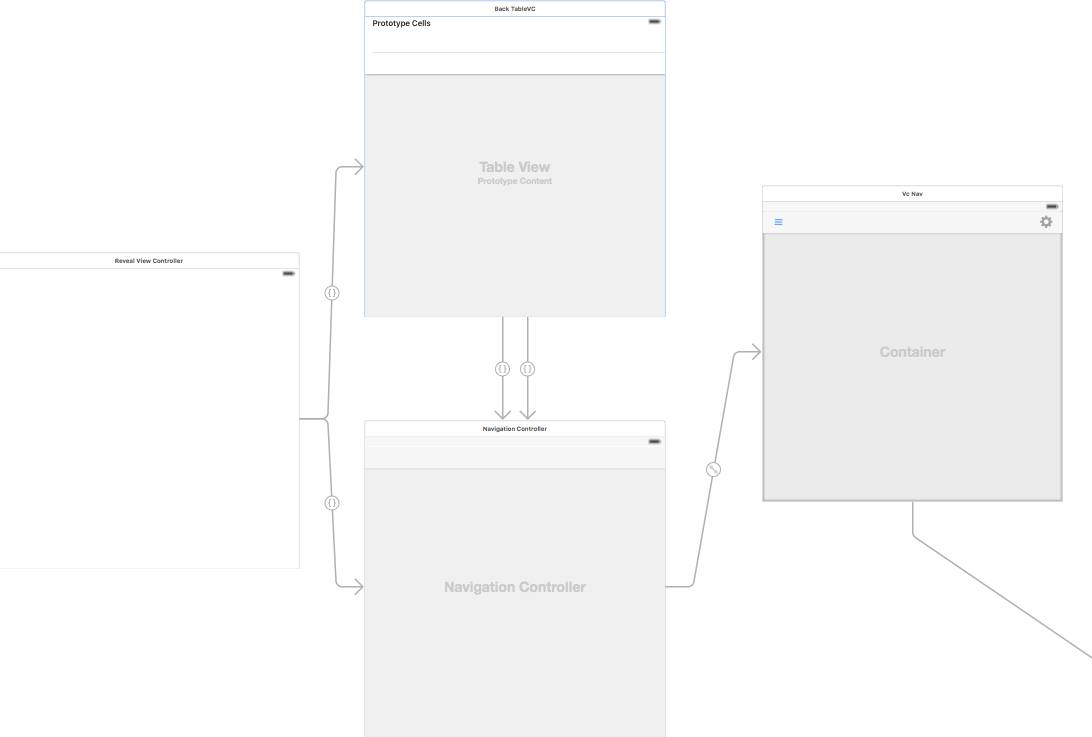
Poveznica na SWRevealController –

<https://github.com/John-Lluch/SWRevealViewController>

Prvo treba ručno kopirati dvije datoteke iz navedenoga repozitorija u projekt. To su sljedeće dvije datoteke:

* SWRevealViewController.m – objective-C datoteka koja sadrži metode i atribute za implementaciju navigation drawera
* SWRevealViewController.h – bridging datoteka koja predstavlja premosnicu između objective-C jezika i Swifta.

U main storyboardu potrebno je dodati UIViewController i postaviti prilagođenu klasu. Također, postavite UIViewController kao inicijalni kontroler koji će se prvi pokrenuti prilikom pokretanja aplikacije.



U osnovi, SWRevealViewController je korijenski kontroler koji kontrolira dva kontrolera, a to su: navigation drawer controller (na slici je to table view kontroler s imenom Back TableVC) i UIViewController koji sadrži ContainerView u koji će se učitavati pogled. UIViewController sa containerom ispred sebe mora imati Navigation Controller kako bi se na njemu prikazivao navigation bar. Navigation bar sadrži hamburger ikonu za izbornik te ikonu za postavke. Dodajte NavigationBar iz palete objekata iznad ContainerViewa te mu dodajte Bar Button Itema, jedan s lijeve i drugi s desne strane te ćemo im pridružiti odgovarajuće slike koje smo dodali u Assets.xcassets dokument (vidljiv u project structure izborniku).

https://scontent-vie1-1.xx.fbcdn.net/hphotos-xfa1/v/t34.0-12/12596415_10207346222207578_262149977_n.jpg?oh=3a29b5c9fdea59c4bddad81863d9c31e&oe=56A2FD83

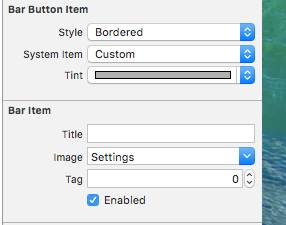
Za dodavanje novog asseta (slike) treba unutar pogleda istoimenog resursa uraditi sljedeće: desni klik-> new Image set. Prikazat će se pogled u kojem spremamo različite veličine tih slika kako bi ikona lijepo izgledala na svim uređajima. Preporuke za veličine ikona jesu:

* 1x = 24x24px
* 2x = 48x48px
* 3x = 75x75px



Ikone možete naći na stranici iconfinder.com ili vectoricons.com.

Nadalje, prethodno dodanim bar buttonima je potrebno pridružiti dodane slike:



Na ovako postavljen main storyboard potrebno je dodati dvije segue odnosno prijelaz između kontrolera. SWRevealController zahtjeva da segue koji se veže na navigation drawer controller bude tipa '*reveal view controller set controller*'. Nakon toga dodajemo još jedan segue na isti način prema Navigation controlleru. Važno je napomenuti kako se na prvi segue treba postaviti attribute identifier na sw\_rear. Isto tako, potrebno je postaviti drugi segue pod attribute identifier na sw\_front. Time dajemo do znanja SWRevealControlleru koji je ViewController prednja strana, a koja zadnja. Primjetimo kako Table View Controllera sadrži dvije stavke koje će nas voditi na dućane ili kartu zbog toga je potrebno iz svake ćelije dodati segue na Navigation Controller View Controllera koji ima Container te njihovu klasu postaviti na SWRevealControllerSeguePushController. Container se dodaje iz Object palete. Implementirati ćemo BackTableVC klasu. Navedena klasa prestavlja kod istoimenog ViewController koji je zapravo tablica/stavke izbornika. Kao i kod svakoga tableViewControllera potrebno je definirati tri glavne funkcije – tableView (preopterećenje prevaziđenih funkcija). Prva metoda tableView vraća ukupni broj ćelija odnosno ukupni broj stavki izbornika. Druga vraća samu ćeliju, dok treća upravlja događajem odabira ćelije te se u njoj mijenja statičko svojstvo klase VcNav (vidi kod ispod). Slijedi kod klase BackTableVC: VcNav kontroler unutar sebe ima Container View u koji ćemo dinamički učitavati pogled. U nastavku slijedi kod VcNav kontrolera:



|  |
| --- |
| import Foundation |
| class BackTableVC: UITableViewController{ |
| var tableArray = [String]() |
| override func viewDidLoad() { |
| super.viewDidLoad() |
| tableArray = ["Stores list", "Map"] |
| } |
| override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int { |
| return tableArray.count |
| } |
| override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell { |
| let cell = tableView.dequeueReusableCellWithIdentifier(tableArray[indexPath.row], forIndexPath: indexPath) as UITableViewCell |
| cell.textLabel?.text = tableArray [indexPath.row] |
| return cell |
| } |
| override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) { |
| VcNav.viewControllerNumber = indexPath.row |
| } |
| } |

**Kod: Kod BackTableVC kontrolera**



|  |
| --- |
|  |
|  |
| import Foundation |
| class VcNav:UIViewController{ |
| @IBOutlet var containerView: UIView! |
| @IBOutlet var Open: UIBarButtonItem! |
| static var viewControllerNumber:Int = 0 |
|  |
| var firstViewController: UIViewController? |
| var secondViewController: UIViewController? |
| var backTableViewController: BackTableVC? |
| private var activeViewController: UIViewController? { |
| didSet { |
| removeInactiveViewController(oldValue) |
| updateActiveViewController() |
| } |
| } |
| private func removeInactiveViewController(inactiveViewController: UIViewController?) { |
| if let inActiveVC = inactiveViewController { |
| // call before removing child view controller's view from hierarchy |
| inActiveVC.willMoveToParentViewController(nil) |
|  |
| inActiveVC.view.removeFromSuperview() |
|  |
| // call after removing child view controller's view from hierarchy |
| inActiveVC.removeFromParentViewController() |
| } |
| } |
|  |
|  |

**Kod: Kod VcNav kontrolera**

Nastavak implementacije:



|  |
| --- |
|  |
|  |
| private func updateActiveViewController() { |
| if let activeVC = activeViewController { |
| // call before adding child view controller's view as subview |
| addChildViewController(activeVC) |
| activeVC.view.frame = containerView.bounds |
| containerView.addSubview(activeVC.view) |
| // call before adding child view controller's view as subview |
| activeVC.didMoveToParentViewController(self) |
| } |
| } |
| override func viewDidLoad() { |
| super.viewDidLoad() |
| firstViewController = UIStoryboard.firstViewController() |
| secondViewController = UIStoryboard.secondViewController() |
| if(VcNav.viewControllerNumber==0) { |
| activeViewController = firstViewController |
| } |
| else { |
| activeViewController = secondViewController |
| } |
| Open.target=self.revealViewController() |
| Open.action = Selector("revealToggle:") |
| self.view.addGestureRecognizer(self.revealViewController().panGestureRecognizer()) |
| } |
| override func didReceiveMemoryWarning() { |
| super.didReceiveMemoryWarning() |
| // Dispose of any resources that can be recreated. |
| } |
| } |
| |  | | --- | | private extension UIStoryboard { | | class func mainStoryboard() -> UIStoryboard { return UIStoryboard(name: "Main", bundle: NSBundle.mainBundle()) } | | class func secondViewController() -> MapViewController? { | | return mainStoryboard().instantiateViewControllerWithIdentifier("MapViewController") as? MapViewController | | } | | class func firstViewController() -> ViewController? { | | return mainStoryboard().instantiateViewControllerWithIdentifier("StoresViewController") as? ViewController | | } | | } | |

**Kod: Kod VcNav kontrolera**

VcNav kontroler sadrži dva atributa odnosno dva ViewControllera kojima ćemo puniti Container View. *firstViewController* nam je referenca na kontroler za dućane dok je *secondViewController* referenca na kontroler za mape. View Controllere programski dohvaćamo iz main storyboarda pomoću ekstenzije. U ekstenzijama nije moguće imati nove atribute stoga je potrebno napraviti type metodu koja će vratiti objekt main storyboard. Objektu u ekstenziji moramo na ovaj način pristupiti. Atribut *activeViewController* će biti View Controller koji se trenutno nalazi unutar Container Viewa. Primjetimo da kod atributa *activeViewController* postoji programski uzorak *property observer*. Nakon postavljanja vrijednosti varijable *activeViewController* izvršiti će se *didSet* dio koda. Osim *didSet* postoji i *willSet .* Također, u tome dijelu koda se briše se stari ViewController koji je bio unutar Containera dok se postavlja novi metodom *updateActiveViewController*. Argument *oldValue* funkcije *removeInactiveViewController* je spremljena stara vrijednost svojstva unutar *didSet* bloka koda.

Metoda *removeInactiveViewController* će maknuti stari ViewController iz Containera dok će metoda *updateActiveViewController* će dodati željeni ViewController ovisno o korisnikovom odabiru unutar navigation drawera odnosno izbornika.

Dakle, svaki puta kada odaberemo željenu ćeliju unutar tableViewControllera backTableVC, odnosno odaberemo jednu stavku navigation drawera tada se okine jedan od dva definirana seguea. Zatim se unutar klase backTableVC okine tableView metoda koja postavi statički atribut VcNav klase *viewControllerNumber* na broj pozicije odabrane ćelije, VcNav-u dajemo do znanja koji viewController treba učitati u Container View. Tada se pokrene seguea i pokrene metoda viewDidLoad unutar klase VcNav koji učita odabrani viewController.

Zadnje tri linije koda metode *viewDidLoad* omogućavaju funkcioniranje navigation drawera. Klasa *SWRevealController* pruža *revealToggle* metodu za upravljanje otvaranjem i zatvaranjem navigation drawera. Cocoa touch koristi target-action mehanizam za komuniciranje između kontrole i nekog drugog objekta. Postavljamo ikoni izbornika svojstvo 'open'. Zatim, target te ikone je *revealViewController* i pokreće se akcija *revealToggle*. Time će se pritiskom na ikonu za otvaranje navigation drawera pokazati isti. Također, dodajemo i prepoznavač gesti (*eng. Gesture recognizer)* za mogućnost otvaranja i zatvaranja navigation drawera klizanjem po ekranu (eng. Swipe).

1. Mape i postavke

**SAŽETAK**

U ovome dijelu ćemo implementirati Apple mape i definirati korisničke postavke.

**KLJUČNE RIJEČI**

Apple mape, korisničke postavke (NSUserDefaults), anotacije na mapi, pin, constraints

**POTREBNA PREDZNANJA**

Kreiranje klase, postavljanje outleta, poznavanje objekt biblioteke

**ISHODI UČENJA**

Nakon ovoga poglavlja znati ćemo implementirati mape i postaviti korisničke postavke. Također, naučiti ćemo postaviti ograničenja u main storyboardu.

**RAZRADA**

* 1. Mape

U ovoj vježbi ćemo implementirati mape i na njima prikazati dućane. Odabirom dućana otvarati će nam se prozor sa dodatnim informacijama o popustima. Za implementaciju mapa koristimo Apple mape - <https://mapsconnect.apple.com/>

Za prikaz mape napraviti ćemo dvije nove klase – *Artwork* i *MapViewController*.

Klasa *Artwork*  će nam služiti za prikaz dućana na mapi pomoću pinova. Zapravo, klasa će predstavljati jedan pin na mapi. Potrebno je definirati sljedeće atribute klase koje želimo spremiti i prikazati na određenome pinu. U konstruktoru klase potrebno je postaviti te atribute i naposljetku pozvati konstruktor nadređene klase, u ovome slučaju nadređena klasa je *NSObject*. Također, klasa implementira protokol *MKAnnotation.* Kako bi ispoštovali protokol potrebno je dodati svojstvo *subtitle* koje će vraćati *locationName*. Napomenimo kako klasa treba importati *MapKit.*

Slijedi kod *Artwork* klase:



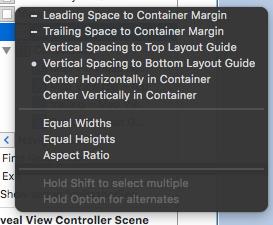
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import MapKit | | class Artwork: NSObject, MKAnnotation { | | let title: String? | | let locationName: String | | let discipline: String | | let storeId: String | | let coordinate: CLLocationCoordinate2D | | init(title: String, locationName: String, storeId:String, discipline: String, coordinate: CLLocationCoordinate2D) { | | self.title = title | | self.locationName = locationName | | self.discipline = discipline | | self.coordinate = coordinate | | self.storeId = storeId | | super.init() | | } | | var subtitle: String? { | | return locationName | | } | | } | |
|  |

**Kod: Artwork klasa**

Potrebno je dodati novi ViewController u storyboard i povezati ga sa TableViewom Discounts Scene. Segue ćemo nazvati *onShowDiscountSegue* i definirati ćemo ga kao segue tipa show.

Zatim ćemo u objekt biblioteci pronaći Map Kit View te ga povući na ViewController u main storyboardu. U hijerarhiji locirajte dodani Map Kit View te kliknite na njega držeći tipku Control odvucite na nadređeni element odnosno View.

Prvo odaberemo *Leading space to container margin,* ponovimo postupak pa odaberemo *Trailing space to container margin. Postupak ćemo ponoviti još dva puta te ćemo odabrati* Vertical Spacing to top layout guide, odnosno vertical spacing to bottom layout guide.



Sljedeći korak je implementacija klase *MapViewController*. Klasa nasljeđuje *UIViewController* i implementira protokole *MKMapViewDelegate* i *CLLocationManagerDelegate*. Također, potrebno je importati MapKit, CoreLocation, UIKit i db modul.

Sada je potrebno u novododani viewController na sceni desno u identity inspector dodati klasu MapViewController.

Kada smo to napravili, na ranije opisani način ćemo odvući map kit view unutar klase te kreirati outlet i nazvati ga *mapView*.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import UIKit | | import MapKit | | import db | | import CoreLocation | | class MapViewController: UIViewController, MKMapViewDelegate { | |  | | @IBOutlet weak var mapView: MKMapView! | |  | | var stores: [Store]? | |  | | var annotations: [MKAnnotation]? | |  | | var senderView: MKAnnotationView? | |  | | let locationManager = CLLocationManager() | | var currentLocation = CLLocation() | |  | |
| } |

**Kod: Svojstva klase MapViewController**



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | override func viewDidLoad() { | | super.viewDidLoad() | | mapView.delegate = self | | mapView.mapType = MKMapType.Hybrid | | let stores = DbController.sharedDBInstance.realm.objects(Store) | | self.stores = stores.reverse() | | //get users location | | // Ask for Authorisation from the User. | | self.locationManager.requestAlwaysAuthorization() | | // For use in foreground | | self.locationManager.requestWhenInUseAuthorization() | | if CLLocationManager.locationServicesEnabled() { | | locationManager.delegate = self | | locationManager.desiredAccuracy = kCLLocationAccuracyNearestTenMeters | | locationManager.startUpdatingLocation() | | } | | mapView.showsUserLocation = true | | currentLocation = CLLocation(latitude: 46.310409, longitude: 16.343013) //ZA PROBU | | centerMapOnLocation(currentLocation) | | generateAnnotations() | | } | |
|  |

**Kod: viewDidLoad metoda MapViewControllera**

Ovdje ćemo se koristiti delegatima, ali detaljniji opis istoga možemo pronaći u sljedećim poglavljima.

U metodi viewDidLoad postavljamo tip mape, učitavamo dućane iz lokalne baze te prikazujemo mapu i centriramo prikaz. Centiranje prikaza izvršiti ćemo metodom *centerMapOnLocation* koja kao parametar prima lokaciju tipa *CLLocation.*



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | func centerMapOnLocation(location: CLLocation) { | | let regionRadius: CLLocationDistance = 20000 | | let coordinateRegion = MKCoordinateRegionMakeWithDistance(location.coordinate, | | regionRadius \* 2.0, regionRadius \* 2.0) | | mapView.setRegion(coordinateRegion, animated: true) | | } | |
|  |

**Kod: centerMapOnLocation metoda**

U navedenoj metodi definiramo radijus koji će biti prikazan te definiramo koordinate mape u ovisnosti o navedenom radijusu. Za detaljnije opcije manipularanja mapama posjetiti poveznicu: <http://www.raywenderlich.com/90971/introduction-mapkit-swift-tutorial>

Mapa je prikazana i centrirana pa je jedino preostalo prikazati markere dućana. Postavljanje markera dućana izvršava se metodom *generateAnnotations.*

Na početku metode definiramo prazno polje annArray koje ćemo puniti sa markerima dućana koji se nalaze unutar radijusa. Iterirati ćemo kroz dućane koje smo učitali iz lokalne baze te ćemo za svaki od njih izračunati da li se njihova lokacija nalazi unutar radijusa postavljenog u postavkama. Nakon toga definiramo artwork instancu i napunimo ga potrebnim informacija te ga dodamo u polje. Naposljetku polje dodamo na mapu i time su markeri prikazani.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | func generateAnnotations(){ | | var annArray: [MKAnnotation] = [] | | print("current radius is:", storeRadiusInMeters) | | for store in stores! { | | let location = CLLocation(latitude: Double(store.latitude), longitude: Double(store.longitude)) | | print("current distance for store is:",location.distanceFromLocation(currentLocation)) | | if(location.distanceFromLocation(currentLocation)<storeRadiusInMeters){ | | let artwork = Artwork(title: store.name, | | locationName: store.desc, | | storeId: String(store.remoteId), | | discipline: "NaN", | | coordinate: CLLocationCoordinate2D(latitude: Double(store.latitude),longitude: Double(store.longitude))) | | annArray.append(artwork) | | } | | } | | mapView.addAnnotations(annArray) | | } | |
|  |

**Kod: generateAnnotations metoda**

Prilikom pritiska na određeni marker otvara se prozor u koji ćemo postaviti gumb za pristup detaljnijim informacijama o popustu. Za to će nam služiti sljedeća metoda. Metoda mapView će na svaki marker dodati gumb sa poveznicom na detalje. Prvo se provjerava je li anotacija koja se prikazuje na mapi za korisničku lokaciju te ako je vraća se nil kako bi se prikazala predodređena anotacija za lokaciju korisnika (plavi kružić) a ne anotacija jednaka anotaciji za trgovine.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | func mapView(mapView: MKMapView, viewForAnnotation annotation: MKAnnotation)  -> MKAnnotationView? {     |  | | --- | | if annotation.isKindOfClass(mapView.userLocation.classForCoder) | | { | | return nil | | } | | | let reuseId = "pin" | | var pinView = mapView.dequeueReusableAnnotationViewWithIdentifier(reuseId) | | if(pinView == nil) | | { | | pinView = MKPinAnnotationView(annotation: annotation, reuseIdentifier: reuseId) | | pinView?.canShowCallout = true | | } | | let button = UIButton(type: UIButtonType.DetailDisclosure) | | pinView?.rightCalloutAccessoryView = button | | return pinView | | } | |
|  |

**Kod: mapView metoda**

Druga mapView metoda će biti pozvana kada korisnik pritisne dodani gumb. Ukoliko je pritisnuta desna kontrola, spremiti ćemo u atribut klase o kojem se markeru radi te ćemo pozvati segue sa identifikatorom koji smo ranije definirali – *onShowDiscountSegue*



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | func mapView(mapView: MKMapView, annotationView view: MKAnnotationView,  calloutAccessoryControlTapped control: UIControl) { | | if control == view.rightCalloutAccessoryView{ | | print(view.annotation!.title) // annotation's title | | print(view.annotation!.subtitle) // annotation's subttitle | | self.senderView = view | | performSegueWithIdentifier("onShowDiscountSegue", sender: nil) | | } | | } | |
|  |

**Kod: mapView metoda**

Posljednje je potrebno implementirati metodu *prepareForSegue* koja se poziva svaki puta prije nego će se dogoditi segue iz viewControllera. Ovo nam je potrebno iz razloga što trebamo destinationViewControlleru proslijediti dućan koji je korisnik odabrao kako bi mogli prikazati detalje.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) | | { | | if segue.identifier == "onShowDiscountSegue" | | { | | if let destination = segue.destinationViewController as? DiscountsViewController | | { | | var clickedStore: Store? | | for store in stores! | | { | | if(store.name == (senderView?.annotation!.title)!) | | { | | clickedStore = store | | } | | } | | destination.discounts = clickedStore!.discounts | | } | | } | | } | |
|  |

**Kod: prepareForSegue metoda**

Metoda LocationManager se poziva stalno čime se obnavljaju informacije o lokaciji uređaja. Metoda dobavlja latitude i longitude lokacije i određuje regiju na temelju dobivenih koordinata.



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | func locationManager(manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) { | | let location = locations.last! as CLLocation | | let center = CLLocationCoordinate2D(latitude: location.coordinate.latitude,  longitude: location.coordinate.longitude) | | let region = MKCoordinateRegion(center: center, span:  MKCoordinateSpan(latitudeDelta: 0.01, longitudeDelta: 0.01)) | | self.mapView.setRegion(region, animated: true) | | } | |
|  |

**Kod: LocationManager metoda**

* 1. Postavke

Na main storyboard ćemo dodati sljedeće objekte iz objekt biblioteke:

- Navigation Controller

- ViewController (sadrži Modal Radius Picker)

- TableViewController (naziv - Settings )

- TableViewController (naziv – Application Settings)

- ViewController (naziv – Modal)

Sljedeće je potrebno povezati novododane objekte. Iz ikone settings koja se nalazi u VcNav Containeru ćemo povući segue na ViewController modal i odabrati present modally kao tip seguea. Iz settings gumba ćemo povući segue tipa push prema Navigation Controlleru. Iz navigationControllera ćemo povući segue prema TableView Controlleru (settings).

Dodajmo novu klasu Settings kojom ćemo raditi konkretne izmjene na globalnoj razini



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import Foundation | | import UIKit | | class Settings { //klasa koja sluzi za konkretne izmjene na globalnoj razini | | static func changeAnimationSetting(newState:Bool){ | | UIView.setAnimationsEnabled(newState) | | } | | } | | extension UIViewController { //koristenjem extenzija se prosiruju funkcionalnosti postojecih klasa | | func backSwipeCheck(){ //svaki UIViewcontroller ce sada imati funkcionalnost za provjeru jel se smije backswipeati il ne | | if(NSUserDefaults.standardUserDefaults().boolForKey("UseBackSwipe")){ | | self.navigationController!.interactivePopGestureRecognizer!.enabled = true | | } | | else { | | self.navigationController!.interactivePopGestureRecognizer!.enabled = false | | } | | } | | func animationCheck(){ | | if(NSUserDefaults.standardUserDefaults().boolForKey("UseAnimations")){ | | Settings.changeAnimationSetting(true) | | } | | else { | | Settings.changeAnimationSetting(false) | | } | | } | | } | |
|  |

**Kod: Settings klasa**

Klasa settings sadrži statičku metodu changeAnimationSetting koja kao parametar prima bool koji će biti true ukoliko želimo uključiti animacije. Kako bi definirali da li želimo animacije potrebno je pozvati UIView statičku metodu setAnimationsEnabled i proslijediti joj kao argument bool vrijednost. Unutar datoteke setting.swift osim definirane klase Setting definirano je i proširenje UIViewController klase. Klasu UIViewController smo proširili sa dvije nove funkcionalnosti. Proširenjem svaki UIViewController dobiva funkcionalnost za provjeru smije li back-swipeati ili ne te provjeriti je li smije koristiti animacije.

* 1. Dodavanje modalnog kontrolera

Zatim dodajemo klase koje ćemo pridodati određenim objektima u main storyboardu. Prva na redu je klasa ModalViewController pa napravimo novu klasu i dodajmo ju na ViewController (modal).

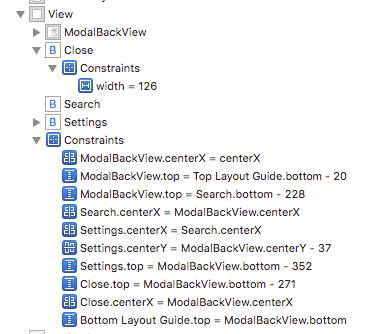


|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import UIKit | | import db | | class ModalViewController: UIViewController { | | @IBAction func onClose(sender: UIButton) { | | self.dismissViewControllerAnimated(true, completion: {}); | | } | | override func viewDidLoad() { | | super.viewDidLoad() | | // Do any additional setup after loading the view. | | } | | override func didReceiveMemoryWarning() { | | super.didReceiveMemoryWarning() | | // Dispose of any resources that can be recreated. | | } |   } |
|  |

**Kod: ModalViewController klasa**

Na dodani modal ćemo postaviti tri gumba (Search, Settings, Close). Iz gumba Search ćemo povući segue prema Navigation Controller koji je povezan sa Table View Controllerom. Iz gumba Settings ćemo povući segue prema Navigation Controlleru koji je povezan s Settings TableView Controllerom.

Definirati ćemo ograničenja (eng. *constraints*) kao na slici:



Da bi modalni pogled funkcionirao na način da on zblja bude modalni prilikom kreiranja treba uraditi sljedeće:

1. Segue s controllera koji ide na controller koji želimo da se prikaže na modalni način (popUp) mora biti definiran kao *present modally* (atribut *kind*)
2. Presentation postavite na Over Current Context
3. Background atribut View-a unutar ModalViewControllera (inicijalni koji je kreiran s ViewControllerom) mora biti postavljen na **Clear Color**
4. Dodajte UIImageView u ModalViewController koji pokriva cijeli ekran i postavite mu neku sliku koja će zamutiti prvi ekran s kojeg smo došli u ModalViewController: slika je dostupna na sljedećem linku: <http://lh5.ggpht.com/_QLwms0mVa4w/S_FsI-DMghI/AAAAAAAAAp8/-4Uwv-yDhVY/behind_alert_view.png?imgmax=800>
5. Nabacajte gumbove unutar nekog UIViewa te akciju na close button definirajte kako je opisano dolje na stranici



Za detaljnije proučavanje postavljanje auto layout-a i definiranje constrainta proučite sadržaj sljedećih poveznica:

<http://www.raywenderlich.com/115440/auto-layout-tutorial-in-ios-9-part-1-getting-started-2>

<http://www.raywenderlich.com/115444/auto-layout-tutorial-in-ios-9-part-2-constraints>

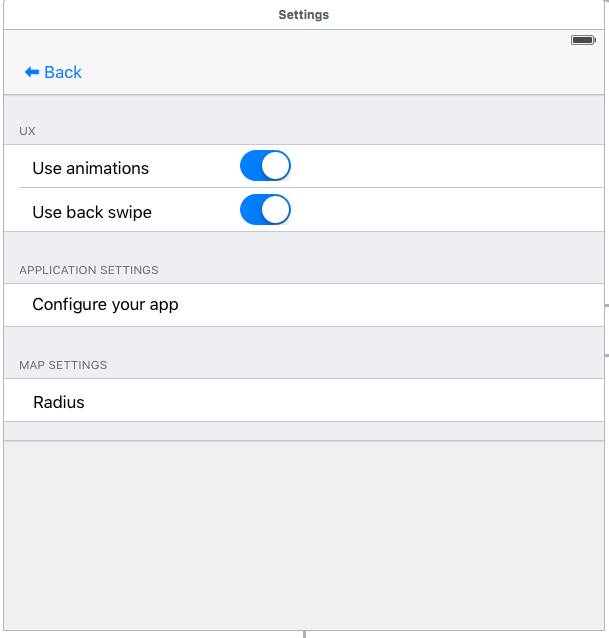
Na ranije opisani način ćemo odvući gumb close koji se nalazi na modalu u klasu te ćemo definirati metodu koja će se pozvati pri pritisku na gumb. Sve što moramo napraviti u toj metodi jest maknuti iz pogleda ViewController (modal). U ModalViewController dodajemo sljedeći dio koda:

|  |
| --- |
| @IBAction func onClose(sender: UIButton) { |
| self.dismissViewControllerAnimated(true, completion: {}); |
| } |

* 1. Dodavanje kontrolera za postavke

Sljedeća klasa je SettingsTableViewController te ju povezati sa TableViewController (settings). Odabrati ćemo TableView u hijerarhiji te u attributes inspectoru postaviti content na static cell i postaviti 3 sekcije. Nakon toga će se unutar TableViewa kreirati sekcije. Odabrati ćemo prvu sekciju te definirati da sadrži dva redka i u header staviti naziv 'UX'. U toj sekciji se nalaze postavke vezane za korisničko iskustvo. Sljedeća sekcija će imati jedan redak i header joj je 'application settings'. Iz ćelije druge sekcije ćemo povući segue prema TableViewControlleru (application settings) tipa show. Treća sekcija također ima jednu ćeliju te će se sekcija nazivati 'map settings'. Iz ćelije treće sekcije ćemo povući segue prema ViewController koji sadrži Modal Radius Picker tipa present modally.

Nakon što ste uradili prethodno SettingsTableViewController bi trebao izgledati ovako:



Slijedi kod SettingsTableViewControllera:



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | import UIKit | | class SettingsTableViewController: UITableViewController, UIGestureRecognizerDelegate { | | let prefs = NSUserDefaults.standardUserDefaults() | | @IBOutlet var useBackSwipeSwitch: UISwitch! | | @IBOutlet var useAnimationsSwitch: UISwitch! | | @IBAction func OnUseAnimationsChanged(sender: UISwitch) { | | prefs.setBool(sender.on, forKey: "UseAnimations") | | Settings.changeAnimationSetting(sender.on) | | } | | @IBAction func OnBackSwipeChanged(sender: UISwitch) { | | prefs.setBool(sender.on, forKey: "UseBackSwipe") | | backSwipeCheck() | | } | | func loadFromUserPrefs(){ | | useAnimationsSwitch.setOn(prefs.boolForKey("UseAnimations"),animated:false) | | useBackSwipeSwitch.setOn(prefs.boolForKey("UseBackSwipe"), animated:false) | | } | | func handleGesture(gestureRecongizer: UISwipeGestureRecognizer){ | | if gestureRecongizer.state == .Ended{ | | performSegueWithIdentifier("settingsToRevealSegue", sender: self) | | } | | } | | override func viewDidLoad() { | | super.viewDidLoad() | | loadFromUserPrefs() | | backSwipeCheck() | | //dodavanje handlera za swipe | | self.navigationController?.interactivePopGestureRecognizer?.delegate = self | | self.navigationController?.interactivePopGestureRecognizer | | self.navigationController?.interactivePopGestureRecognizer?.addTarget(self, action: "handleGesture:") | | } | | override func didReceiveMemoryWarning() { | | super.didReceiveMemoryWarning() | | // Dispose of any resources that can be recreated. | | }  } | |  | |
|  |

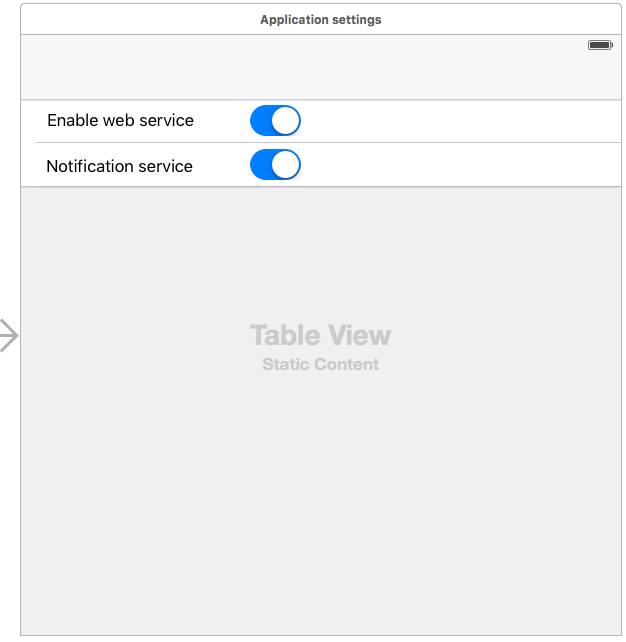
**Kod: SettingsTableViewController klasa**

Klasa sadrži atribut 'prefs' koji sadržava UserDefaults. NSUserDefaults je klasa koja sadržava programsko sučelje za interakciju sa sustavskim zadanim postavkama. Zadane postavke sustava omogućuju aplikaciji da prilagodi svoje ponašanje korisničkim preferencijama.

Sa atributom prefs ćemo manipulirati kada korisnik promijenit vrijednost prekidača. Promjenom vrijednosti pojedinog prekidača promijeniti će se korisničke zadane postavke. Metoda loadFromUserPrefs provjera korisničke postavke te na temelju toga postavlja vrijednost prekidača na true ili false.

Budući da smo u SettingViewController došli iz modal controllera nije moguće vratiti se nazad uobičajenim načinom. Naime, svaki puta kada definiramo push segue u sljedećem ViewControlleru se pojavi back button kojim se vraćamo na prethodni ViewController. Zbog toga je potrebno definirati vlastiti gumb za povratak nazad i vlastite metode za upravljanje back swipeom. Za to nam služi metoda handleGesture. Također smo definirali i segue settingsToRevealSegue koji je povezan na RevealViewController odnosno početni zaslon.

ApplicationSettingsTableViewController također se kreira na isti način kao i SettingsTableViewController samo što ima jednu sekciju. Trebao bi izgledati ovako:



Sljedeća klasa koju ćemo dodati je ApplicationSettingsTableViewController. Navedenu klasu ćemo povezati sa ViewControllerom (application settings).

Slijedi kod klase ApplicationSettingsTableViewController:



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import UIKit | | class ApplicationSettingsTableViewController: UITableViewController { | | @IBOutlet var enableWSSwitch: UISwitch! | | @IBOutlet var notificationServiceSwtich: UISwitch! | | var prefs = NSUserDefaults.standardUserDefaults() | | @IBAction func onEnableWSChange(sender: UISwitch) { | | prefs.setBool(sender.on, forKey: "EnableWebService") | | } | | @IBAction func onNotificationServiceChange(sender: UISwitch) { | | prefs.setBool(sender.on, forKey: "EnableNotificationService") | | } | | func loadFromUserPrefs(){ | | enableWSSwitch.setOn(prefs.boolForKey("EnableWebService"),animated:false) | | notificationServiceSwtich.setOn(prefs.boolForKey("EnableNotificationService"), animated:false) | | } | | override func viewDidLoad() { | | super.viewDidLoad() | | loadFromUserPrefs() | | backSwipeCheck() | | } | | } | |
|  |

**Kod: ApplicationSettingsTableViewController klasa**

Navedena klasa sadrži dva outleta koja predstavljaju prekidače. Kada se učita ViewController stanje prekidača se postavlja u odnosu na stanje u korisničkim postavkama. Također, postoje dvije akcije za svaki prekidač, prva je za uključivanje web servisa, a druga za uključivanje obavijesti. Ovisno o vrijednosti prekidača postavljaju se korisničke postavke.

Za implementaciju pretraživanja potrebno je u main storyboardu ubaciti novi objekt TableViewController i Navigation Controller. U TableView pod attributes inspector treba postaviti Content na Dynamic Prototypes. Time definiramo tablicu koja će se dinamički generirati. Potrebno je kao u prethodnom primjeru dodati back button i definirati metodu za swipe gesture na početni zaslon.

Kreiramo klasu SearchDiscountsTableViewController:

class SearchDiscountsTabeViewController: UITableViewController, UISearchResultsUpdating, UIGestureRecognizerDelegate {

|  |
| --- |
| var discounts: [Discount] = [] |
| var resultSearchController = UISearchController () |
| var filteredDiscounts = [Discount]() |

//… ostatak koda klase

}

Metoda viewDidLoad dohvaća podatke iz lokalne baze. Prema protokolu UISearchResultsUpdating potrebno je definirati objekt tipa UISearchController te postaviti njegovo svojstvo searchResultsUpdater na klasu koja implementira taj protokol.

Prilikom unosa termina pretraživanja rezultat istoga prikazuje podatke na koje nije moguće kliknuti i postaje neaktivan. To je potrebno onemogućiti kako bismo mogli kliknuti odnosno odabrati željeni popust i saznati dodatne informacije o istome. Search bar postavljamo u zaglavlje.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | override func viewDidLoad() { | | super.viewDidLoad() | | discounts = DbController.sharedDBInstance.realm.objects(Discount).reverse() | | //custom backswipe pošto smo došli iz modalviewa na | | self.navigationController?.interactivePopGestureRecognizer?.delegate = self | | self.navigationController?.interactivePopGestureRecognizer?.addTarget(self, action: "handleGesture:") | | self.backSwipeCheck() | | self.resultSearchController = UISearchController(searchResultsController: nil)  //novi searchbar | | self.resultSearchController.searchResultsUpdater=self | | self.resultSearchController.dimsBackgroundDuringPresentation = false  //inače bi dimmao/disableo cijeli tableview dolje  //(zapravo cijeli viewcontroller) - nebi mogo kliknut na discount | | self.resultSearchController.searchBar.sizeToFit() //preko cijelog ekrana | | self.tableView.tableHeaderView = self.resultSearchController.searchBar  //stavljamo searchbar u zaglavlje tableviewa | | self.resultSearchController.searchBar.placeholder="search discounts..." | | definesPresentationContext=true  //ako odeš na neki drugi  //ViewController search bar nece ostati tu | | self.tableView.reloadData() //učitaj podatke | | } | |
|  |

**Kod: viewDidLoad metoda**

Kada korisnik unosi tekst u search bar tada se poziva metoda protokola UISearchResultsUpdating - updateSearchResultsForSearchController koja će zatim pozvati metodu filterContentForSearchText i proslijediti joj uneseni tekst kao argument.

|  |
| --- |
| func updateSearchResultsForSearchController(searchController: UISearchController) { |
| filterContentForSearchText(searchController.searchBar.text!) |
| } |

Metodu filterContentForSearchText koju ćemo koristiti za filtriranje popusta. Popuste ćemo pretraživati prema imenu i opisu. Isprva filtriramo prema imenu te dodajemo u polje, a prilikom filtriranja uvijek metodom alreadyExists provjeravamo da li se već traženi popust nalazi u polju. Zatim nakon filtriranja prema imenu, nastavljamo pretraživati prema opisu i koristiti se opet alreadyExists metoda kako bi provjerili jesmo li već ubacili traženi popust u polje. Na kraju ažuriramo table view sa filtriranim podacima.



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | func alreadyExists(keyDiscount:Discount)->Bool{ | | for discount in filteredDiscounts { | | if(discount.name==keyDiscount.name) { | | return true; | | } | | } | | return false; | | } | |
|  |

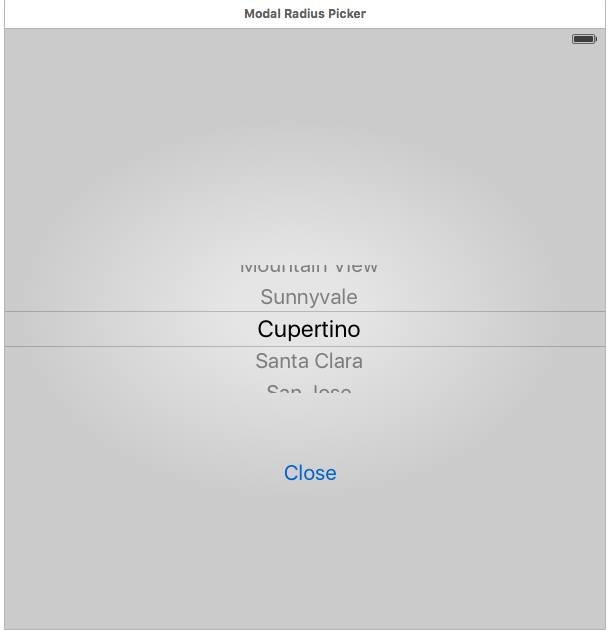
**Kod: alreadyExists metoda**



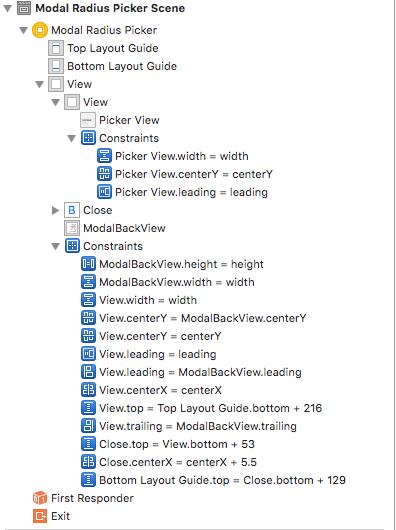
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | func filterContentForSearchText(searchText: String, scope: String = "All") { | | filteredDiscounts = discounts.filter { discount in | | return discount.name.lowercaseString.containsString(searchText.lowercaseString) | | } | | filteredDiscounts.appendContentsOf(discounts.filter { discount in | | if(!self.alreadyExists(discount)){ | | return discount.description.lowercaseString.containsString(searchText.lowercaseString) | | } | | else { | | return false | | } | | } | | ) | | tableView.reloadData() | | } | |
|  |

**Kod: fileContentForSearch metoda**

Nadalje, ostaje nam objasniti modalni prozor za odabir radiusa koji se pokreće dodirom na ćeliju unutar SettingsTableViewControllera.



Prvo dodajte UIView te unutar njega odajte pickerView objekt te gumb koji će služiti za zatvaranje trenutnog modalRadiusPickerViewControllera u kojem se nalazimo. Slika koja prikazuje hijerarhiju i definirana ograničenja između objekata slijedi u prilogu:



Izrada modalnog pogleda te interakcija je ista kao i prethodno opisani način za ModalViewController (pravila koja trebate slijediti).

Slijedi kod za ModalRadiusPickerViewController:



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | // | | import UIKit | | | |  | | | | class ModalRadiusPickerViewController: UIViewController, UIPickerViewDelegate,UIPickerViewDataSource { | | | |  | | | | var prefs = NSUserDefaults() | | | |  | | | @IBOutlet var pickerView: UIPickerView! | | |  | | | var pickerDataSource = ["500m":500,"1Km":1000,"2Km":2000,"10Km":10000,"100Km":100000] | | | @IBAction func onClose(sender: UIButton) { | | | let index = pickerDataSource.startIndex.advancedBy(pickerView.selectedRowInComponent(0)) //da se dobije index tipa DictionaryIndex, ne moze se direktno preko indexa tipa int | | | prefs.setInteger(pickerDataSource.values[index], forKey: "StoreRadius") | | | print("setano je:",prefs.integerForKey(("StoreRadius"))) | | | self.dismissViewControllerAnimated(true, completion: nil); | | | } | | | func loadDefaultPickerValue(){ | | | let radiusInt=prefs.integerForKey("StoreRadius") //vraca 0 ako ne postoji pref StoreRadius | | | var selectedRow=0  if radiusInt != 0 { | | for (\_,value) in pickerDataSource { | | if(value==radiusInt){ | | break | | } | | selectedRowInt++ | | } | | } | | self.pickerView.selectRow(selectedRowInt, inComponent: 0, animated: false) | | } | | override func viewDidLoad() { | | super.viewDidLoad() | | self.pickerView.dataSource=self | | self.pickerView.delegate=self | | loadDefaultPickerValue() | | // Do any additional setup after loading the view. | | } | | func numberOfComponentsInPickerView(pickerView: UIPickerView) -> Int { | | return 1 | | } | |  | | func pickerView(pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int { | | return pickerDataSource.count; | | } | |  | | func pickerView(pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String? { | |  | let index = pickerDataSource.startIndex.advancedBy(row) //vraca index | | | |  | return pickerDataSource.keys[index] | | | |  | } | | | |  | //format the pickerView | | | |  |  | | | |  | override func didReceiveMemoryWarning() { | | | |  | super.didReceiveMemoryWarning() | | | |  | // Dispose of any resources that can be recreated. | | | |  | } | | | |  | } | | | |
|  |

**Kod:** **ModalRadiusPickerViewController**

Nadalje, povukli smo outlet pickerView iz viewControllera unutar storyboarda u kod. Kako bi trenutni pogled mogao implementirati metode pickerViewa treba implementirati protokole UIPickerViewDelegate i UIPickerViewDataSource te trenutni viewController postaviti kao delegat za view i za izvor podataka (datasource) u metodi viewDidLoad što znači da će ova klasa davati izvor podataka za taj view i ova klasa koristi taj view.

Zatim smo definirali riječnik sa svim vrijednostima udaljenosti. Riječnik je tipa [String:Int]. String je ključ koji će se prikazivati unutar pickerViewa dok su Int stvarne vrijednosti koje će spremati u klasu NSUserDefaults sa funkcijom setInteger. Odvukli smo akciju za gumb close te ju nazvali onClose. Nakon zatvaranja se sprema vrijednost koja je ostala unutar pickerViewa te se zatvara trenutni modalni pogled. Funkcija loadDefaultPickerValue će se pozvati u metodi viewDidLoad te će namjestiti vrijednost pickerViewa na onu spremljenu u korisničkim postavkama (NSUserDefaults).

Funkcije koje smo morali implementirati su numberOfComponentsInPickerView te dvije preopterećene funkcije pickerView iz prethodno spomenutih protokola.

Prva funkcija vraća broj komponenti pickerViewa (broj kliznih komponenti pickerViewa – npr. kad imamo picker za datum broj komponenti je 3 – za dan, mjesec i godinu) koja vraća 1 zato jer imamo samo jednu kliznu komponentu.

Druga funkcija vraća broj opcija unutar klizne komponente (broj opcija za radius). Treća funkcija umeće ključeve iz rječnika u pickerView iz scene.

* + 1. Vizualizranje radiusa na mapi

Radius koji je spremljen u modalnom pogledu za biranje radiusa prikazati ćemo na mapi. Prvo što trebate izmjeniti u klasi MapViewController jest dodavanje sljedećih svojstava:

var storeRadiusInMeters: Double = Double(NSUserDefaults().integerForKey("StoreRadius"))

var currCircle = MKCircle()

Prvo svojstvo služi za dohvaćanje radiusa iz klase NSUserDefaults dok drugo svojstvo je tipa MKCircle te je to krug koji će se renderirati oko korisničke lokacije u radiusu specificiranom u prvom svojstvu.

Trebalo bi dodati metodu koja će stvoriti krug te ga nekako specificirati mapi da ga prikaže. To se radi na sljedeći način:



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | func addRadiusCircle(location: CLLocation){ | | let circle = MKCircle(centerCoordinate: location.coordinate, radius: Double(NSUserDefaults.standardUserDefaults().integerForKey("StoreRadius"))  as CLLocationDistance) | | self.currCircle = circle | | self.mapView.addOverlay(circle) | | } | |
|  |

**Kod: addRadiusCircle metoda**

Referenciranjem metode addOverlay svojstva mapView i pridodavanjem mu kruga stvorenog na temelju radiusa iz postavki se poziva preopterećenja metoda mapView koja služi za prikaz preklapajućih (overlay) objekata na mapi (u našem slučaju to je krug) stoga je trebamo implementirati. Kod je sljedeći:



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | func mapView(mapView: MKMapView, rendererForOverlay overlay: MKOverlay) ->  MKOverlayRenderer { | | let circle = MKCircleRenderer(overlay: overlay) | | circle.strokeColor = UIColor.blueColor() | | circle.fillColor = UIColor(red: 0, green: 255, blue: 0, alpha: 0.2) | | circle.lineWidth = 1 | | return circle | | } | |
|  |

**Kod: mapView metoda**

Ova metoda će se pokrenuti svaki put kada se pozove lokacija korisnika te svaki put crta krug radiusa. Metoda će vratiti krug koji će se prikazati preko mape. Unutar nje smo definirali neka svojstva poput boje linije i ispune te širine linije.

Kako bi dobro implementirali stvari koje se trebaju desiti prilikom promjene lokacije korisnika dodati ćemo 3 preopterećene locationManager metode unutar proširenja koje implementira CLLocationManagerDelegate protokol. Kod i objšanjenje slijede:

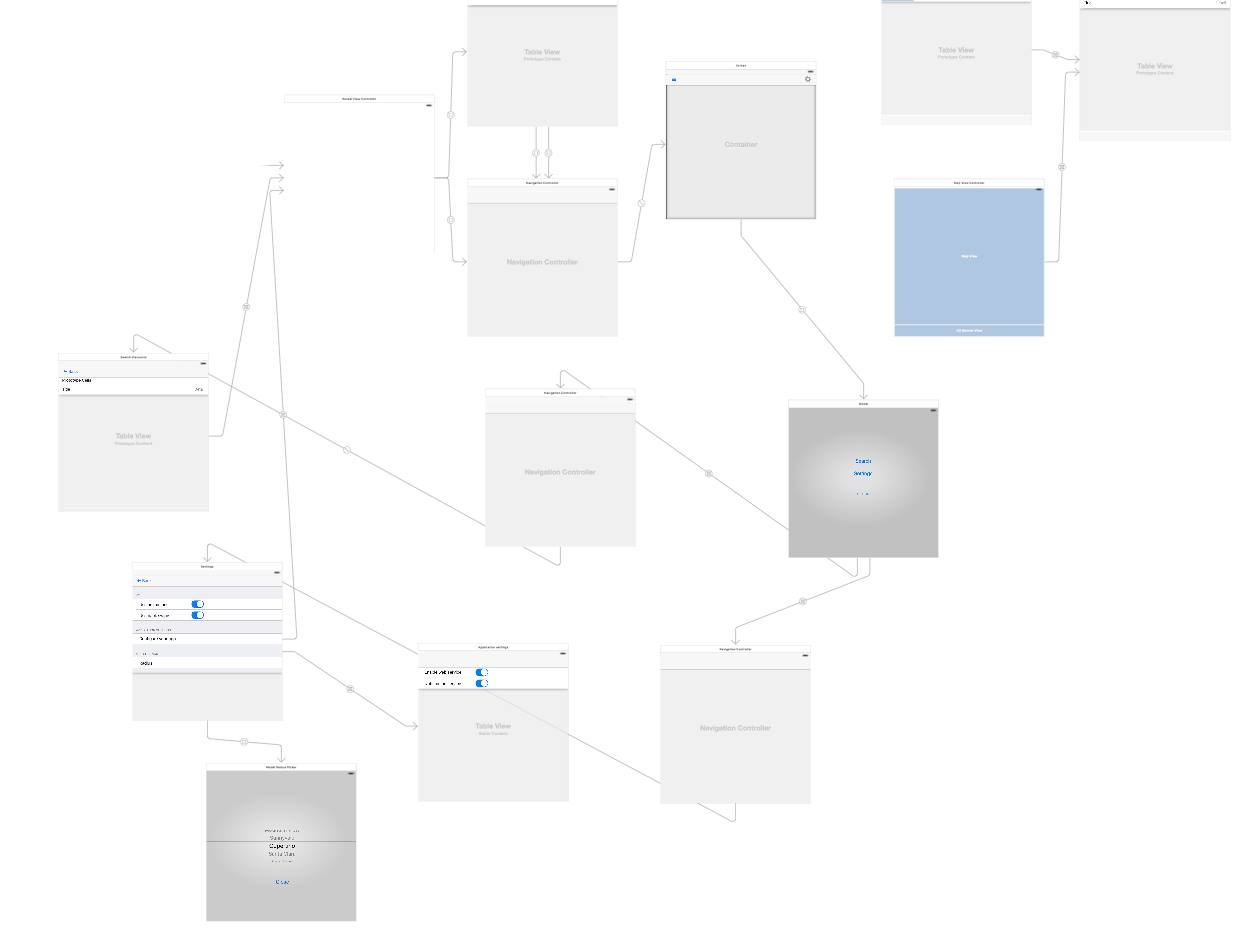


|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | extension MapViewController: CLLocationManagerDelegate | | { | | |  | | | func locationManager(manager: CLLocationManager, didUpdateToLocation newLocation:  CLLocation, fromLocation oldLocation: CLLocation) { | | | self.locationManager.startUpdatingLocation() | | | } | | |  | | | func locationManager(manager: CLLocationManager, didUpdateLocations locations:  [CLLocation]) { | | |  | | | self.locationManager.stopUpdatingLocation() | | |  | | let location = locations.last! as CLLocation | |  | | let center = CLLocationCoordinate2D(latitude: location.coordinate.latitude, longitude: location.coordinate.longitude) | | let region = MKCoordinateRegion(center: center, span: MKCoordinateSpan(latitudeDelta: 0.01, longitudeDelta: 0.01)) | |  | | currentLocation = location | | self.mapView.removeOverlay(self.currCircle) | | addRadiusCircle(currentLocation) | |  | | self.mapView.setRegion(region, animated: true) | | } | |  | | func locationManager(manager: CLLocationManager, didFailWithError error: NSError) { | |  | | self.locationManager.stopUpdatingLocation()  }  } | |  | } | | |  | } | | |
|  |

**Kod: locationManager metode**

Prva metoda će se pozvati kad se promjeni lokacija korisnika te treba ažurirati informacije o lokaciji koje se šalju u drugu metodu koja se poziva stalno i šalje informacije o lokaciji zato smo na početku te metode pozvali metodu stopUpdatingLocation kako se nebi pozivala stalno već samo nakon što korisnik promjeni lokaciju. Tu se još implementirao dio za dohvaćanje lokacije u svojstvo currentLocation te brisanje trenutnog kruga (za prošlu lokaciju) i dodavanje novog kruga za novu lokaciju. Treća istoimena funkcija će se pozvati kada se dogodi greška pri učitavanju lokacije te se tada zaustavlja ažuriranje podataka o lokaciji. Primjetite da se to moglo sve u prvoj metodi napraviti ali u svrhu demonstracije smo to uradili možda na malo teži način kako bi pokazali implementaciju obiju funkcija.

Na kraju bi cijeli storyboard trebao izgledati ovako:



1. Monetizacija

**SAŽETAK**

Sa samo nekoliko linija koda moguće je integrirati reklame unutar iOS aplikacije i početi zarađivati. Apple developer službena stranica nudi odličan opis postupka uvođenja monetizacije.

**KLJUČNE RIJEČI**

Monetizacija, reklame, iAd, banner

**POTREBNA PREDZNANJA**

Constraintovi, outleti i služenje objekt bibliotekom

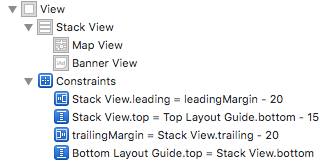
**ISHODI UČENJA**

Nakon ove vježbe student bi trebao moći znati implementirati osnovni banner unutar pogleda.

**RAZRADA**

U ovoj lekciji implementirati ćemo jedan od oblika iAd monetizacije. U naš map viewController dodati ćemo takozvani ad banner view odnosno reklamu u podnožje viewControllera.

Ad banner view je objekt koji se nalazi u objekt biblioteci pa ćemo ga povući u map viewController. Kako bismo ga postavili na dno viewControllera potrebno je dodati StackView iz objekt biblioteke koji ćemo raširiti preko cijeloga viewControllera. U StackView ćemo ubaciti map view i ad banner view. Definirajte constrainte kao na slici.



Nakon umetanja ad banner viewa važno je umetnuti outlet u MapViewController inače će se aplikacija rušiti pri pokretanju. Također, potrebno je importati iAd

import iAd

Dodajte još dvije linije u viewDidLoad metodu:

|  |
| --- |
| bannerView.delegate = self |
| bannerView.hidden = true |

Banner view ćemo sakriti u slučaju da reklame nisu učitane. Zbog toga ćemo također implementirat dvije metode koje će provjerati da li je reklama učitana, te ako je prikazat će banner.

|  |
| --- |
| extension MapViewController: ADBannerViewDelegate |
| { |
| func bannerViewDidLoadAd(banner: ADBannerView!) { |
| bannerView.hidden = false |
| } |
|  |
| func bannerView(banner: ADBannerView!, didFailToReceiveAdWithError error: NSError!) { |
| bannerView.hidden = true |
| } |
| } |

Prethodna implementacija reklame je samo za development verziju aplikacije.

Reklame su tek vidljive nakon što se aplikacija stavi na App Store, a u development fazi se prikazuje predefinirana reklama koja nam daje do znanja da smo spojeni na iAd uslugu.

1. Push notifikacije

**SAŽETAK**

U ovoj vježbi student će naučiti kreirati app id i kreirati sve certifikate koji su potrebni za slanje i primanje push notifikacija.

**KLJUČNE RIJEČI**

Certifikati, app id, push notifikacije, openssl, x509, php, apns, pem

**POTREBNA PREDZNANJA**

Kako funkcioniraju push zahtjevi

**ISHODI UČENJA**

Nakon obrade vježbe student će znati kreirati certifikat i poslati push notifikacije na uređaj

**RAZRADA**

* 1. Zahtjev za certifikatom

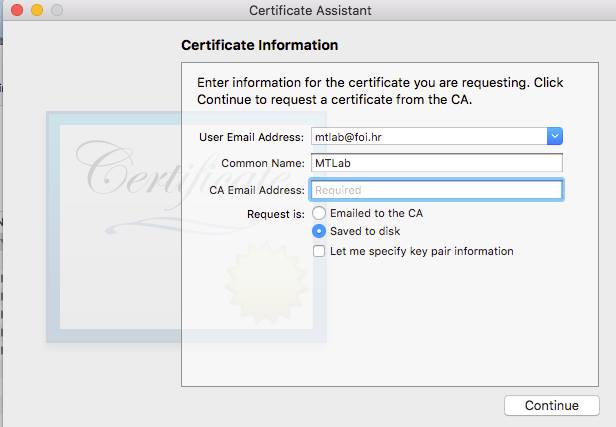
Push notifikacije ili tzv. Server push notifikacije dostavljaju informaciju klijentu bez specifično zahtjeva klijenta prema serveru. S druge strane postoje pull notifikacije za koje klijent radi zahtjev. Krajnji korisnik mora dozvoliti aplikaciji slanje push notifikacije.

Za potrebe slanja push notifikacija na iOS platformi potrebno je imati Apple developer id koji se naplaćuje.

Prije svega je potrebno otići na KeyChain Access koji je već instaliran na Mac računalu. KeyChain Access nam služi za generiranje zahtjeva za certifikatom koji ćemo podnijeti Apple-u na Apple developer portalu. Odabrati ćemo sljedeću opciju u KeyChainu.



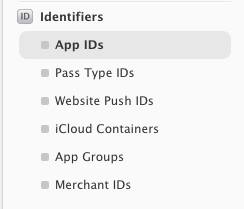
Nakon toga će se otvoriti prozor u kojemu ćemo unijeti email i obavezno odabrati Saved to disk:



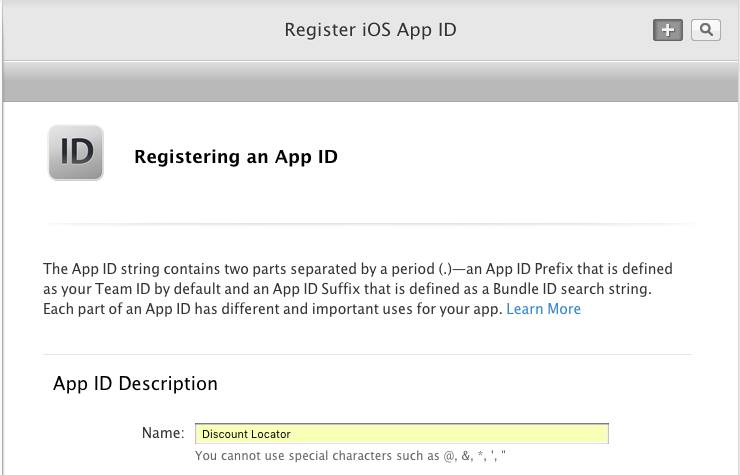
Tako dobivenu datoteku ćemo spremiti na računalo koju ćemo kasnije koristiti pri generiranju certifikata.

* 1. Registriranje aplikacije

Logirajte se na Apple developer portal. Nakon toga u lijevom izborniku pod Identifiers otiđite na App IDs



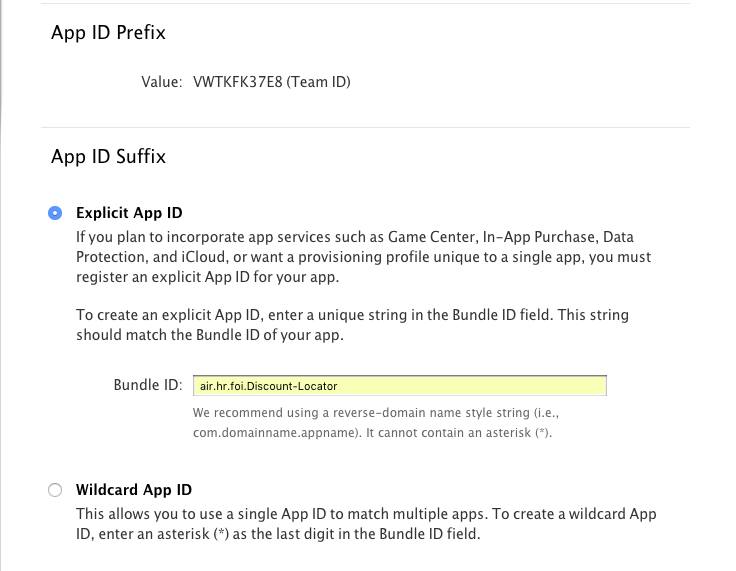
Zatim ćemo pod name unijet ime aplikacije



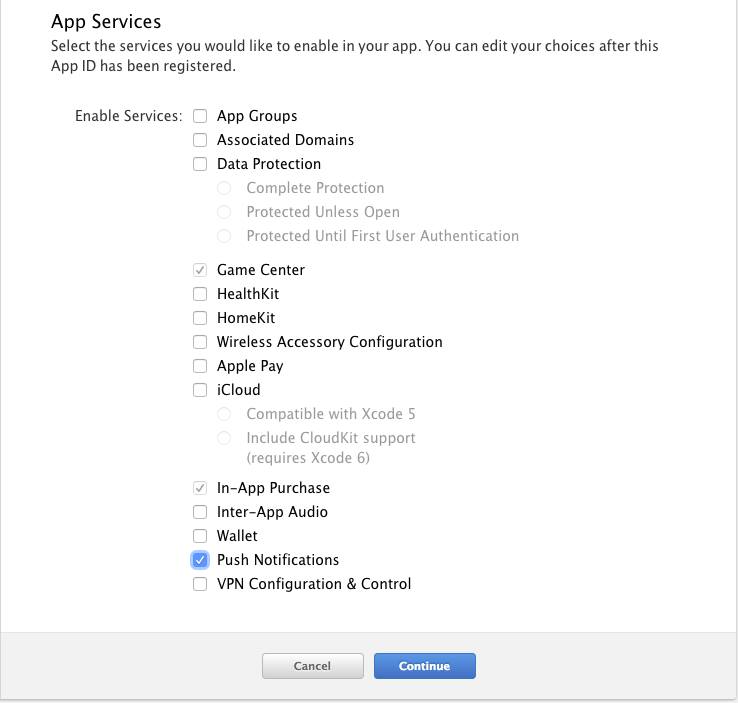
Dodati ćemo Bundle ID. Ukoliko ne znate koji je vaš Bundle ID otiđite u projekt i u projektu otiđite na General opcije i odaberite pod Targets Discount Locator kao što je prikazano na slici:



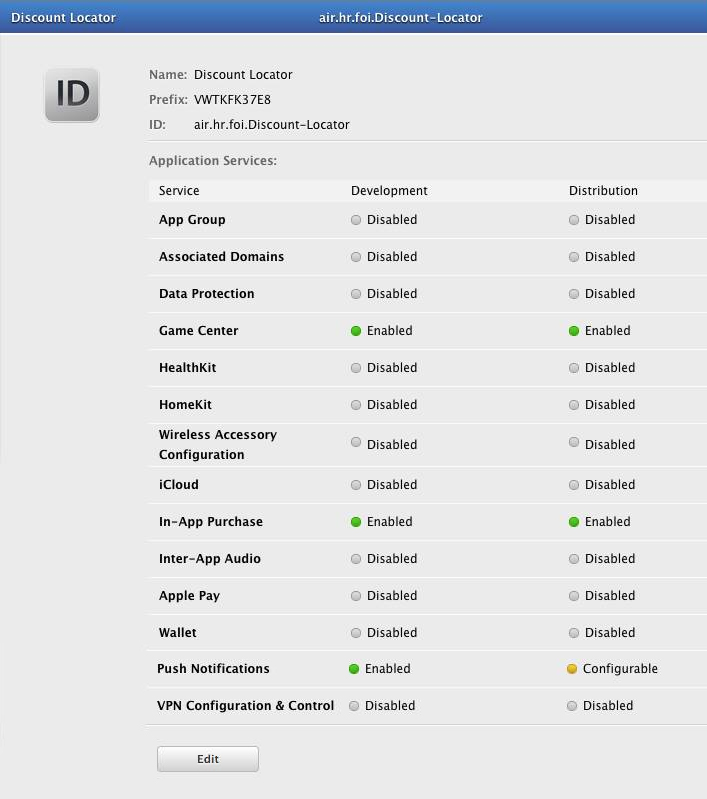
Sada ispunimo polje Bundle ID:



Zatim ćemo odabrati koje servise trebamo. Unaprijed su odabrani Game Center i In-App Purchase servisi, a mi ćemo odabrati još Push Notifications servis.

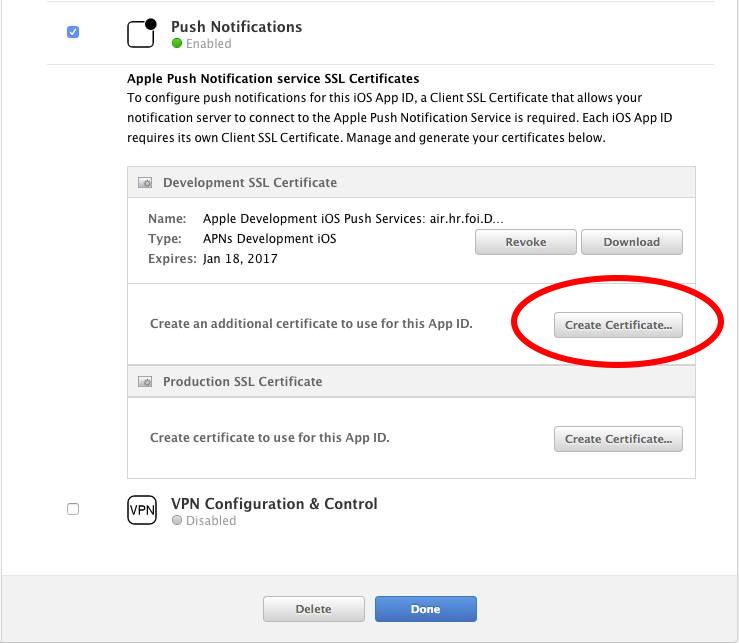


Nakon što je uspješno kreiran App ID. Otići ćemo ponovno u lijevom izborniku pod Identifiers na App IDs i odabrati naš projekt te će se pojaviti sljedeći zaslon na kojemu ćemo stisnuti gumb Edit:

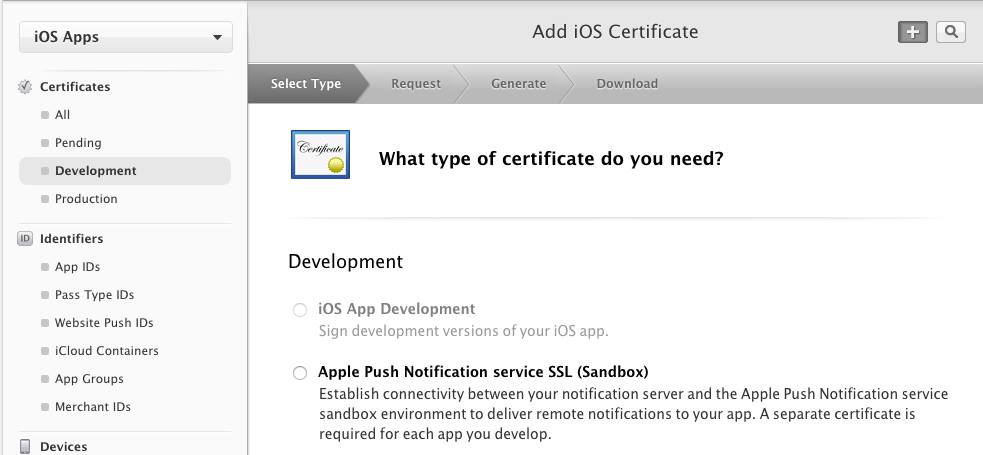


* 1. Preuzimanje certifikata

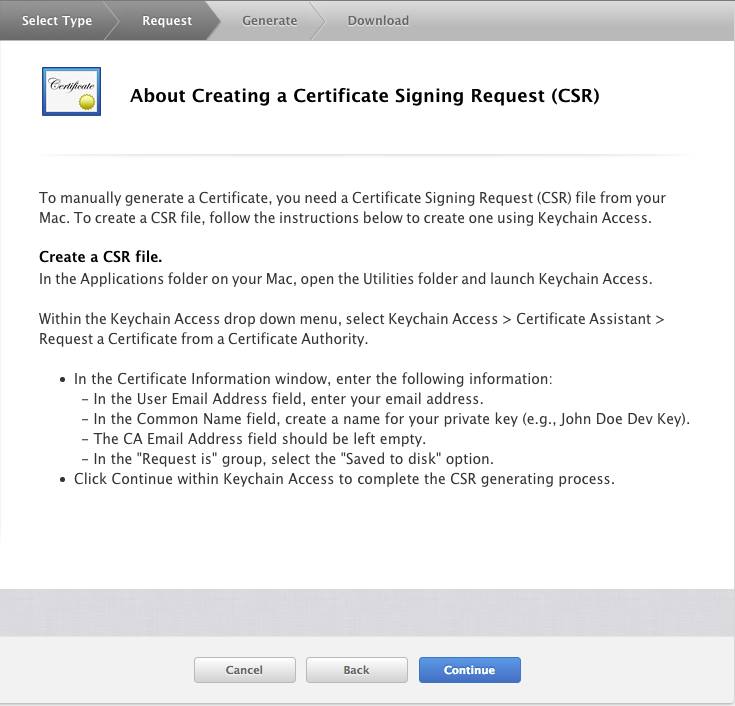
Nakon toga će se pojaviti novi zaslon, otiđite na dno i pod sekcijom Development SSL Certificate treba odabrati Create Certificate:



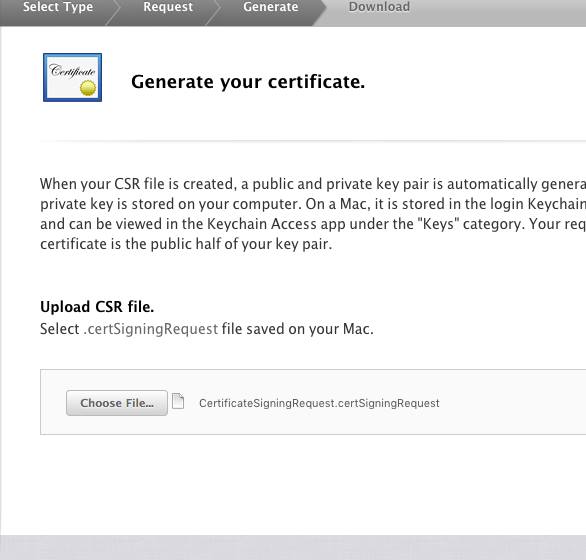
Odabirom Create Certificate pojaviti će se sljedeći zaslon gdje ćemo odabrati iOS App Development. Budući da smo mi već kreirali certifikat ova opcija nam je onemogućena.



Na sljedećem zaslonu odaberemo Continue



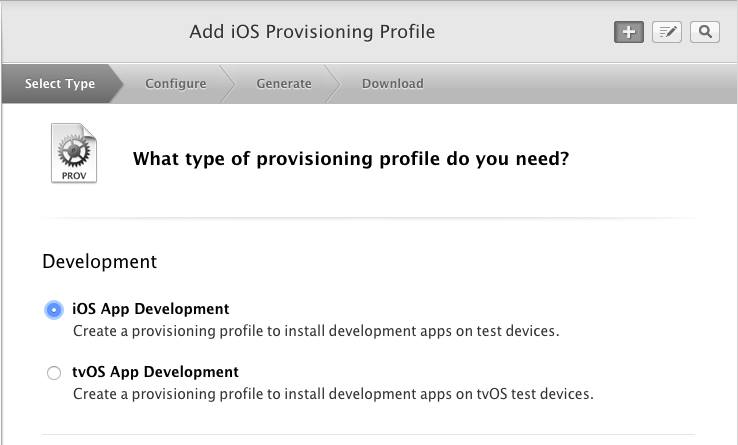
Kod Generate your cerficate ćemo odabrati datoteku koju smo generirali pomoću KeyChain Access te odabrati Generate. Nakon klika na Generate na sljedećem zaslonu ćemo preuzeti certifikat na računalo.



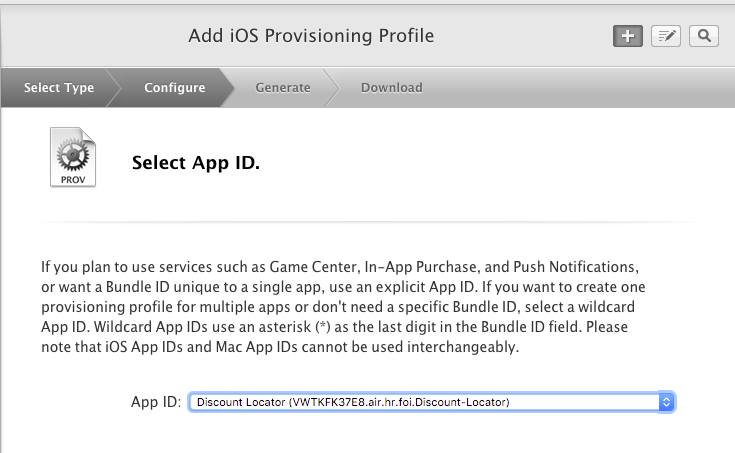
Sljedeći korak je kreiranje Provision profila. U lijevom izborniku pod Provisioning Profiles odaberite Development:



Odabiremo iOS App Development i idemo dalje



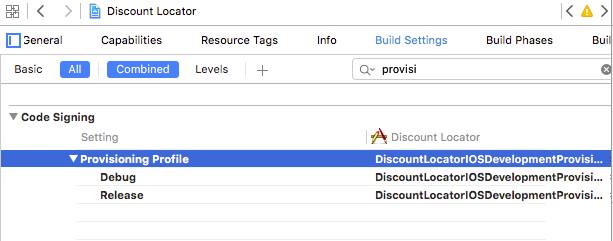
Pod App ID odaberemo našu aplikaciju:



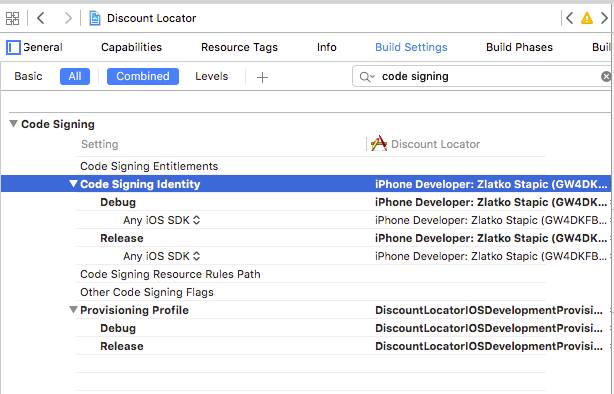
Na kraju generiramo i preuzmemo datoteku.

* 1. Podešavanje provision profila u XCodeu

Otvorite generirani Provision Profile u XCodeu. Zatim ćemo otići na projekt Build Settings i pronaći Provisioning Profile koji ćemo promijeniti na upravo dodani Provision Profile:



Zatim ćemo isto u Build Settingsima pronaći code signing i odabrati developera kojega smo kreirali na Apple developer portalu.



Još je samo preostalo dodati sljedeći kod u AppDelegate klasu:



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool { | | // Override point for customization after application launch. | | let notificationTypes : UIUserNotificationType = [.Alert, .Badge, .Sound] | | let notificationSettings : UIUserNotificationSettings =  UIUserNotificationSettings(forTypes: notificationTypes, categories: nil) | | UIApplication.sharedApplication()  .registerUserNotificationSettings(notificationSettings) | | return true | | } | | func application(application: UIApplication,  didRegisterUserNotificationSettings notificationSettings: UIUserNotificationSettings) | | { | | UIApplication.sharedApplication().registerForRemoteNotifications() | | } | | func application(application: UIApplication,  didRegisterForRemoteNotificationsWithDeviceToken deviceToken: NSData) { | | print(deviceToken) | | } | | func application(application: UIApplication,  didFailToRegisterForRemoteNotificationsWithError error: NSError) { | | print(error.localizedDescription) | | } | |  | | func application(application: UIApplication, didReceiveRemoteNotification userInfo:  [NSObject : AnyObject]) { | | } | |
|  |

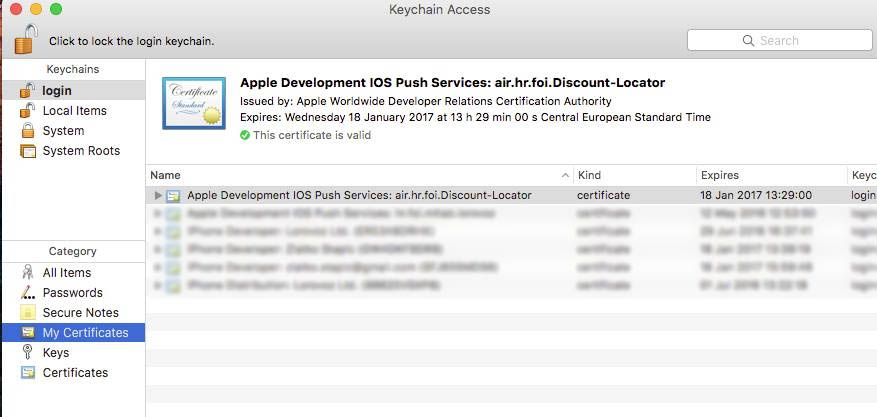
**Kod: metoda za primanje push notifikacija**

Prilikom pokretanja aplikacije prihvatiti ćemo primanje push notifikacija. U konzoli će se ispisati Device Token ID koji će nam trebati za izradu serverske strane za push notifikacije.

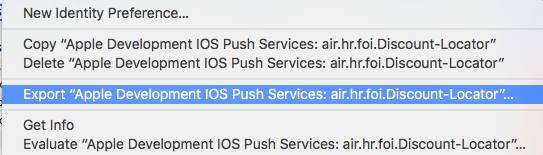


* 1. Implementacija serverske strane

Certifikat koji smo preuzeli će imati ekstenziju .cer. Potrebno ga je otvoriti u KeyChain Accessu. Potrebno je odaberi kategoriju My Certificates, a zatim Keychains - Login



Desnim klikom na certifikat otvara se sljedeći prozor na kojemu ćemo odabrati export…:



Time ćemo exportati certifikat sa ekstenzijom p12 i nazovite ga Certificates. Ovako treba izgledati naziv datoteke – Certificates.p12. Prilikom exportanja certifikata nećemo dodjeljivati nikakvu lozinku. Sada otvorite terminal u datoteci gdje se nalaze svi preuzeti certifikati. Potrebno je izvršiti prvu naredbu kako bismo iz certifikata preuzetog sa Apple developer portala generirali .pem certifikat:

$ openssl x509 -inform der -in aps\_development.cer -out certificate.pem

Sljedećom naredbom ćemo generirati .pem certifikat iz .p12 generiranog certifikata. Prilikom izvršavanja naredbe u terminalu ćemo morati unijeti lozinku, ali budući da nismo postavili lozinku prilikom generiranja p12 certifikata samo ćemo pritisnuti enter za nastavak. Također će nas pitati za dodjelu passphrase koju ćemo koristiti u skripti.

$ openssl pkcs12 -nocerts -in Certificates.p12 -out p12Certificates.pem

Zadnja naredba povezuje oba certifikata u jedan konačni:

$ cat certificate.pem p12Certificates.pem > apns\_cert.pem

Tako dobiveni certifikat potrebno je staviti na server zajedno sa php skriptom dostupnoj na poveznici: <https://github.com/sivaganeshsg/PHP-Push-Notification-File/blob/master/apns_php.php>

Unutar skripte je potrebno promijeniti device token i passphraseu koju smo unijeli prilikom generiranja p12 certifikata. Pokrenite skriptu i push notifikacija će biti poslana na uređaj.

Napomena: Moguće je provjeriti valjanost SSL certifikata izvršavanjem naredbe u terminalu.

$ openssl s\_client -connect gateway.sandbox.push.apple.com:2195 -cert apns\_cert.pem -key apns\_cert.pem

Ukoliko dobijemo poruku connected znači da je SSL certifikat valjan.

1. Modularnost

**SAŽETAK**

U ovoj kratkoj vježbi ćemo naučiti koncept slijepe komunikacije(eng. *Blind comunication)* između modula na primjeru dvije klase iz dva različita modula. To je arhitekturalni pristup izgradnji iOS aplikacija koristeći dva bitna svojstva – protokole I delegate.

**KLJUČNE RIJEČI**

delegat, protokol, modularnost, design pattern, arhitektura

**POTREBNA PREDZNANJA**

Poznavanje OOP paradigme.

**ISHODI UČENJA**

Izgradnja modularne aplikacije koju je jednostavno nadograđivati.

**RAZRADA**

* 1. Koncept slijepe komunikacije modula

Koncept slijepe komunikacije između modula zapravo označava način komunikacije gdje se klasi dodaje protokol u kojem se navode metode koje će biti potrebno implementirati u drugoj klasi nekog drugog modula (delegiranje). Na taj način klasa u modulu ne mora znati ništa o samoj implementaciji I ne zna koja klasa implementira taj protokol već samo navodi da se u svrhu ostvarenja željene funkcionalnosti taj dio mora implementirati. U nastavku će biti pokazani primjer takve komunikacije.



|  |
| --- |
| import Alamofire |
|  |
| public protocol WebServiceResultDelegate{ |
| func getResult(json:AnyObject) |
| } |
|  |
| public class HTTPRequest |
| { |
|  |
| public var delegate:WebServiceResultDelegate? |
|  |
| public init(){} |
|  |
| public func httprequest(url: String, params: [String:String]) |
| { |
| Alamofire.request(.POST, url, parameters: params) |
| .responseJSON { response in |
| if let json = response.result.value{ |
|  |
| self.delegate?.getResult(json) |
|  |
| } |
| } |
| } |
| } |

**Kod: Slijepa komunikacija – 1. (slijepi) dio – klasa HTTPRequest**

U klasi HTTPRequest definiramo protokol WebServiceResultDelegate koja definira funkciju getResult koja će biti implementirana u bilokojoj klasi koja će implementirati taj protokol (u našem slučaju to je klasa WebServiceDataLoader iz glavnog modula DiscountLocator). Funkcija getResult bi trebala vratiti odgovor web servisa. Također, definirali smo varijablu tipa tog protokola u koju će se postaviti ona klasa koja ga implementira. Također u httprequest funkciji se zove funkcija getResult ali se zapravo ne zna unutar kojeg je ona objekta, odnosno ta funkcija je unutar objekta koji je implementira (to je zapravo bit slijepe komunikacije). Slijedi drugi dio, a to je implementacija samog protokola u klasi WebServiceDataLoader. Ovakav način je zapravo uzorak dizajna Observer gdje se na promjenu/zahtjev unutar jednog objekta aktivira neka akcija unutar drugog objekta.



|  |
| --- |
|  |
| import Foundation |
| import db |
| import core |
| import ws |
|  |
| public class WebServiceDataLoader:DataLoader |
| { |
| public var stores: [Store] = [] |
| public var discounts: [Discount] = [] |
| public var storesTableView: UITableView? |
| private var discountsLoaded: Bool = false  private var prefs = NSUserDefaults() |
| **private var storesLoaded: Bool = false** |
|  |
| **var httpRequest = HTTPRequest()** |
| |  | | --- | | public func LoadData() { | |  | | if(NetConnection.Connection.isConnectedToNetwork()&&prefs.boolForKey("EnableWebService")) { | | print("loadm sa web servisa") | | **httpRequest.delegate = self** | | httpRequest.httprequest("http://cortex.foi.hr/mtl/courses/air/stores.php", params: ["method": "getAll"]) | | } | | else { | | print("loadam lokalno") | | self.showDataFromLocalDB() | | } | | } | |

…

}

**Kod: Slijepa komunikacija – 2. (neslijepi - implementacijski) dio, pt.1**

Dodali smo atribut (objekt) klase HTTPRequest te ga inicijalizirali. Nadalje, u funkciji LoadData smo postavili delegat u “self” odnosno, klasa je postavila samu sebe kao delegata te smo pozvali funkciju httprequest iz objekta httpRequest.



|  |
| --- |
| public func getResult(json: AnyObject) { |
| if(!storesLoaded){ |
| storesLoaded = true |
| self.stores = JsonAdapter.getStores(json) |
| self.storesLoaded = true |
| self.showLoadedData() |
| httpRequest.httprequest("http://cortex.foi.hr/mtl/courses/air/discounts.php", params: ["method":"getAll"]) |
| }else{ |
| self.discounts = JsonAdapter.getDiscounts(json) |
| self.discountsLoaded = true |
| self.showLoadedData() |
| } |
| } |
|  |
| } |

**Kod: Slijepa komunikacija – 2. (neslijepi - implementacijski) dio, pt.2**

Dio koji se brine za implementiranje funkcije getResult je stavljen unutar ekstenzije klase WebServiceDataLoader. To je moglo I bez ekstenzije (funkcionalnost je ista), ali se preporuča radi preglednosti koda, ali isto tako I za identificiranje točaka komunikacije da se one izoliraju u ekstenzijama (proširenja funkcionalnosti klase). Sama implementacija (pomalo generičke) funkcije getResult sadrži provjeru jesu li dućani dohvaćeni, tek kada su dućani dohvaćeni okida se drugi http zahtjev I postavlja varijabla storesLoaded u true tako da se u sljedećem zahtjevu dohvate popusti (else dio selekcije).

* 1. Gradnja modularnijeg koda

Za sada aplikacija nije modularna jer ima mogućnost dohvata podataka s web servisa i iz lokalne baze ali ako bi htjeli nadograditi spektar funkcionalnosti dohvaćanja podataka, npr. s SD kartice trebali bi se mnogo pomučiti s refaktoriranjem koda. Cilj modularnosti je da da bilo koja klasa koja će dohvaćati podatke naslijedi neku apstraktniju klasu te da implementira svoj način dohvaćanja. Trenutno u klasi WebServiceDataLoader imamo implementiran dio za dohvaćanje iz baze ukoliko nema interneta/korisnik ne želi dohvaćati podatke s web servisa. To baš i nije dobar prisup. Dobar pristup bi da se svaka funkcionalnost odvoji u zasebnu klasu (princip jednostruke odgovornosti) te da se kod prikaza podataka (npr. kod početnog viewControllera) ispita logika za način dohvaćanja te neovisno o načinu se ti isti podaci dohvate uz minimalnu promjenu u kodu prilikom nadograđivanja novih načina dohvaćanja. Te implementacijske klase se mogu zvati WebServiceDataLoader (postojeća), DBDataLoader, SearchDataLoader, SDKDataLoader itd.

Krenimo od definiranja općenitije klase (u modulu Core): DataLoader.



|  |
| --- |
|  |
| import Foundation |
| import db  //this protocol will be implemented by some controller which shows the data/knows what to do with data |
| public protocol OnDataLoadedDelegate { |
| func onDataLoaded(stores : [Store], discounts: [Discount]) |
| }  //“abstract“ class |
| public class DataLoader |
| { |
| public var stores:[Store]? |
| public var discounts:[Discount]? |
| public var **onDataLoadedDelegate:OnDataLoadedDelegate?** |
|  |
| func LoadData() {} //implemented by subclass Loaders (ws/db/sdk/whatsover) |
| public init(){} |
| public func **dataLoaded()** -> Bool { //called by loaders when they are done with its job |
| if (stores==nil || discounts == nil) { |
| return false |
| } |
| else { |
| **onDataLoadedDelegate?.onDataLoaded(stores!, discounts: discounts!) //trigger the controller to show the data** |
| return true |
| } |
|  |
| } |
| } |

**Modularniji Kod: klasa DataLoader**

Na početku je definiran protokol OnDataLoadedDelegate koji sadrži funkciju onDataLoaded koja prima polje trgovina i popusta. Instanca tog protokola je uz polje trgovina i popusta definirana u klasi DataLoader te će se bilo koja klasa koja implementira dohvaćanje podataka postaviti u tu instancu. Navedena je funkcija (samo definicija) LoadData koju će bilo koja klasa koja dohvaća podatke i implementirati. Također, funkcija dataLoaded će forsirati bilo koju klasu koja će prikazivati podatake da izvrši funkciju onDataLoaded. Nadalje, klasa WebServiceDataLoader (i bilo koja klasa kasnije koja dohvaća podatke) ima svoju implementaciju dohvaćanja podataka putem funkcije LoadData() koja sada ima kod koji se tiče implementacije podataka putem dohvaćanja s web servisa (httpRequest). Kod koji će se pozvati na odgovor web servisa jest isti (funkcija getResult u ekstenziji) samo što se podaci spremaju u svojstva DataLoader klase a ne WebServiceDataLoader (tu više ne postoje).



|  |
| --- |
|  |

public func LoadData(){

|  |
| --- |
| httpRequest.wsResultDelegate = self |
| httpRequest.httprequest("http://cortex.foi.hr/mtl/courses/air/stores.php", params: ["method": "getAll"]) |
| } |
|  |
|  |
| private func showLoadedData() |
| { |
|  |
| if(self.storesLoaded && self.discountsLoaded){ |
| self.bindData() |
| **self.dataLoaded()//**from DataLoader class |
| } |
| } |

**Modularniji Kod: klasa WebServiceDataLoader**

Nakon prihvata rezultata kao i prije funkcija getResult će pospremiti trgovine i popuste te će se pozvati funkcija showLoadedData koja će okinuti prethodno spomenutu **dataLoaded()** funkciju iz DataLoader klase koja će forsirati bilo koju klasu zaduženu za plasiranje podataka na screen da to i uradi (putem funkcije onDataLoaded). Možemo primjetiti da će svaka Loader klasa koja nasljeđuje DataLoader uraditi isto te da će se neovisno o načinu podaci dohvatiti. Nadalje, zadnji korak je da kontroler koji će prikazivati trgovine i popuste implementira sučelje OnDataLoadedDelegate odnosno metodu onDataLoaded. Kod nas je to klasa „ViewController“. Klasa ViewController će sadržavati objekt klase WebServiceDataLoader (ili bilo koje klase koja implementira dohvat podataka), pozvati njegovu funkciju LoadData te se postaviti se kao implementatora sučelja OnDataLoadedDelegate te u ekstenziji odraditi posao s primljenim podacima (narediti prikaz).



|  |
| --- |
|  |

|  |
| --- |
| public class ViewController: UITableViewController { |
|  |
| @IBOutlet weak var storesTableView: UITableView! |
| var stores = [Store] () |
| var discounts = [Discount] ()  var webServiceDataLoader = WebServiceDataLoader() |
| //…  public override func viewDidLoad() {     |  | | --- | |  | | if(NetConnection.Connection.isConnectedToNetwork() && NSUserDefaults.standardUserDefaults().boolForKey("EnableWebService")){ | | **webServiceDataLoader.onDataLoadedDelegate = self //nested property from DataLoader class** | | webSeerviceDataLoader.LoadData() | | }  //else…bilo koji drugi način dohvaćanja podataka | |  |   } |

**Modularniji Kod: klasa ViewController –deklariranje svojstava te metoda viewDidLoad**



|  |
| --- |
|  |

|  |
| --- |
| extension **ViewController:OnDataLoadedDelegate** { |
| public func onDataLoaded(stores: [Store], discounts: [Discount]) { |
| self.stores=stores |
| self.discounts=discounts |
| tableView.reloadData() |
| } |
| } |

**Modularniji Kod: klasa ViewController – implementacija prihvata podataka iz podatkovnog sloja (DataLoader)**

Dakle, kada su podaci primljeni (u argumentu nam ih šalje DataLoader klasa) metodom reloadData se osvježe podaci tableViewa koji služi za prikaz trgovina. Zgodno je to što ViewController klasu „nije briga“ tko joj dohvaća podatke već ona samo implementira ono što će se dogoditi kada su podaci dohvaćeni. Ono što ostvaruje modularnost jest zapravo generalnija klasa DataLoader koju možemo smatrati „middle-manom“ između loader klasa koje rade posao dohvaćanja podataka te kontrolera koji će prihvatiti podatke jer objekt bilo koje klase koja naslijeđuje DataLoader (**implementatori** dohvaćanja) će pozvati dataLoaded metodu iz generalnije klase te će se u istoj metodi proslijediti ti podaci **k implementatoru prikaza** (ViewController dostupan preko delegate svojstva protokola za implementaciju prikaza) pozivanjem metode koju je on implementirao.

Na isti način ćemo implementirati komunikaciju prilikom dohvaćanja iz lokalne baze.

Unutar klase DBController u modulu db stvoriti ćemo novi protokol:

|  |
| --- |
| code.png |

|  |
| --- |
| import Foundation |
| import RealmSwift |
|  |
| public protocol DBResultDelegate{ |
| func getStores(result:[Store]) |
| func getDiscounts(result:[Discount]) |
| }   |  | | --- | | public class DbController | | { | |  | | public static var sharedDBInstance = DbController() //singleton object | |  | | **public var dbResultDelegate:DBResultDelegate?** | | public let realm = try! Realm() | |  | | //… | | public func realmFetchStores() | | { | | let data = self.realm.objects(Store) | | dbResultDelegate?.getStores(data.reverse()) //calls getStores in the implementator class | | } | | public func realmFetchDiscounts() | | { | | let data = self.realm.objects(Discount) | | dbResultDelegate?.getDiscounts(data.reverse()) | | } | | } | |

**Modularniji Kod: Klasa DBController - definiranje protokola i njegovog svojstva**

Deklarirali smo da klasa koja želi implementirati dohvaćanje iz lokalne baze (loader klasa) mora imati implementirane funkcije za dohvat trgovina i popusta te ko i uvijek dodali svojstvo dbResultDelegate u koje će se implementator postaviti. Nadalje, kreirati ćemo klasu DBDataLoader koja će naslijediti općenitiju klasu DataLoader i implementirati konkretno dohvaćanje podataka odnosno implementirati metodu LoadData() i dvije prethodno navedene metode iz DBResultDelegate protokola te pozivanje funkcije *dataLoaded* (iz generalnije klase DataLoader) nakon što su podaci dohvaćeni.

|  |
| --- |
| code.png |

|  |
| --- |
| import Foundation |
| import core |
| import db |
| import Realm |
| import RealmSwift |
|  |
| public class DBDataLoader:DataLoader { |
|  |
| public func LoadData(){ |
| DbController.sharedDBInstance.dbResultDelegate = self |
| showDataFromLocalDB() |
| } |
|  |
| private func showDataFromLocalDB() |
| { |
| DbController.sharedDBInstance.realmFetchStores() |
| DbController.sharedDBInstance.realmFetchDiscounts() |
| } |
| } |
|  |
| extension DBDataLoader:DBResultDelegate { |
| public func getDiscounts(result: [Discount]) { |
| self.discounts = result |
| self.dataLoaded() |
| } |
| public func getStores(result: [Store]) { |
| self.stores = result |
| self.dataLoaded() |
| } |
| } |

**Modularniji Kod: Klasa DBDataLoader – implementacija LoadData() i protokola DBResultDelegate u ekstenziji**

Dakle, u metodi LoadData se klasa DBDataLoader postavlja kao implementator dohvaćanja podataka iz baze te zove metodu showDataFromLocalDB koja će putem singletone objekta (statičko svojstvo) sharedDBInstance pozvati dohvaćanje trgovina i popusta u klasi DbController. Nadalje, ako pogledate kod unutar DBController klase realmFetch metode će implementatoru proslijediti podatke te će se one u njemu okinuti (konkretno u našoj klasi DBDataLoader). Na kraju će se pozvati funkcija generalnije klase dataLoaded koja će opet forsirati implementatora koji koristi te podatke da radi s njima što hoće (konkretno, kod nas je to klasa ViewController koji će te podatke prikazati u svojem pogledu). Ostaje još samo dodati kod u klasi ViewController koji će implementirati prikaz podataka ukoliko se oni dohvaćaju iz lokalne baze = nadogradnja funkcionalnosti dohvaćanja podataka. Konkretno, potrebno je dodati svojstvo klase koja je implementator dohvaćanja podatka za lokalnu bazu – DBDataLoader te u istom opet postaviti sebe (ViewController) klasu kao implementatora prikaza podataka.

|  |
| --- |
| code.png |

|  |
| --- |
| public class ViewController: UITableViewController { |
|  |
| //… |
| var webServiceDataLoader = WebServiceDataLoader() |
| **var dbDataLoader = DBDataLoader()** |
| public override func viewDidLoad() { |
| super.viewDidLoad() |
| //… |
| if(NetConnection.Connection.isConnectedToNetwork() && NSUserDefaults.standardUserDefaults().boolForKey("EnableWebService")){ |
| webServiceDataLoader.onDataLoadedDelegate = self |
| webServiceDataLoader.LoadData() |
| }**else{** |
| **dbDataLoader.onDataLoadedDelegate = self** |
| **dbDataLoader.LoadData()** |
| **}** |
|  |
| } |
|  |
| //…. |
| } |

**Modularniji Kod: Klasa ViewController: dodavanje načina dohvaćanja podataka iz lokalne baze**

Kod koji će prikazati te podatke (prethodno definiran unutar ekstenzije) ostaje isti.



Za vježbu probajte implementirati klasu SearchDataLoader koja će na isti način poput DBDataLoadera dohvaćati podatke iz lokalne baze te ih proslijeđivati SearchDiscountTableViewControlleru na prikaz. Trenutno je vćeina logike za dohvaćanje/filtriranje podataka implementirana u kontroleru, razmslite o pristupu gdje se cijela logika implementira u klasi SearchDataLoader (vraćanje podataka na upit) dok se kontroler oslobađa od logike te samo služi za prikaz podataka.

1. Testiranje

**SAŽETAK**

U ovoj vježbi ćemo naučiti koncept testiranja unutar XCodea. Vježba obuhvaća Unit testove te UI testove. Za testiranje bit će korišten XCodeov framework za testiranje iOS aplikacija – XCTest.

**KLJUČNE RIJEČI**

Testiranje, Unit test, UI test, XCTest.

**POTREBNA PREDZNANJA**

Poznavanje koncepata testiranja neovisno o programskom okruženju i jeziku.

**ISHODI UČENJA**

Čitatelj/korisnik ove skripte bi trebao znati primijeniti koncepte testiranje u Xcodeu te poznavati XCTest FW.

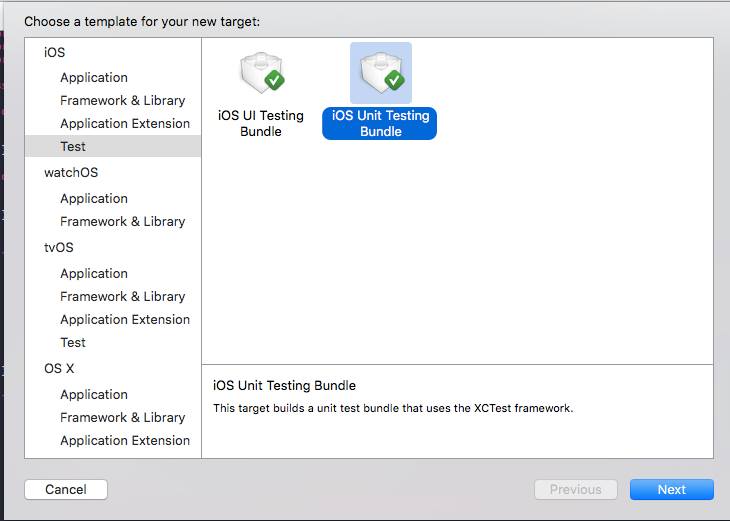
**RAZRADA**

Xcode pruža mogućnosti za iscrpno testiranje softvera. Testiranje projekata povećava mogućnost razvijanja robustnosti, smanjuje bugove te ubrzava prihvatljivost proizvoda za distribuciju i prodaju. Dobro testirane aplikacije koje se ponašaju očekivano povećavaju zadovoljstvo korisnika. Xcode pruža XCTest framework koji omogućuje unit testove te UI testove.

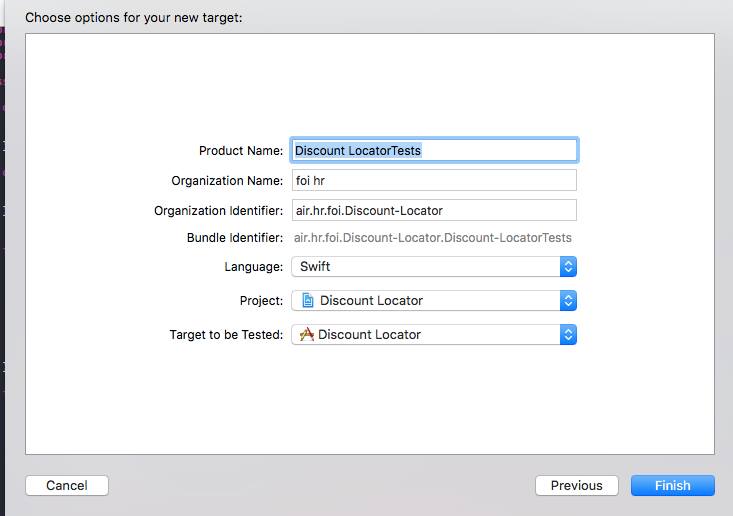
* 1. Unit testovi

Što se tiče unit testova, oni služe za testiranje funkcionalnosti pojedinih objekata. Takvi testovi su dobar način za pisanje modularnog koda koji se lako održava. Stara programerska izreka kaže: „ako vaš kod nije lako testirati tada također neće biti lagan za održavanje i otkrivanje grešaka (clean code)“. Često se klase trebaju oponašati (mock) odnosno pružati lažne funkcionalnosti kako bismo izolirali specifičnu funkcionalnost koju tada možemo testirati. Prionimo.

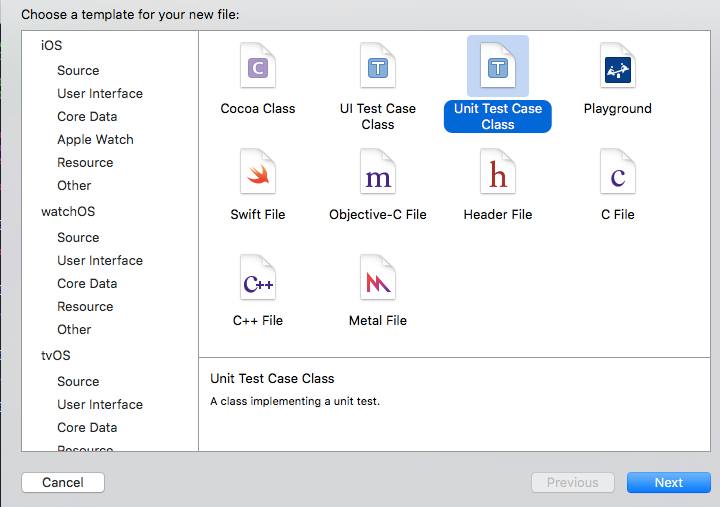
Kao prvo, treba kreirati novi modul (u žargonu Xcodea target) kao na sljedećoj slici (File->new->target):



Odaberemo ios UnitTesting bundle i odaberemo naziv modula i ostale potrebne podatke -> finish…



U tom trenutku će nam biti stvoren modul za testiranje i sadržavati će jednu klasu inicijalno kreiranu prilikom kreacije targeta. Dodajmo datoteku u modul tipa Unit Test Case Class i nazovimo je CoreUnitTests. Tu ćemo testirati Core modul.¸



Unit Test klasa inicijalno dolazi sa 4 metode. Zadnje dvije možemo izbrisati, dok ćemo metode setUp i tearDown ostaviti. Radi se o metodama koje se pozivaju prije i nakon svake funkcije testiranja. Dodajmo novu funkciju testiranja u klasu koju ćemo nazvati testParsing. Ne prima nikakve parametre niti ne vraća ništa.



|  |
| --- |
| import XCTest |
| import core |
| import db |
| import SwiftyJSON |
|  |
| class CoreUnitTests: XCTestCase { |
|  |
| override func setUp() { |
| super.setUp() |
| // Put setup code here. This method is called before the invocation of each test method in the class. |
| } |
|  |
| override func tearDown() { |
| // Put teardown code here. This method is called after the invocation of each test method in the class. |
| super.tearDown() |
| } |
| func testParsing(){ |
|  |
| let jsonStores: [String: AnyObject] = [ |
| "items": "[{\"id\": \"1\",\"name\": \"1\",\"description\": \"1\",\"imgUrl\": \"1\",\"longitude\": \"1\",\"latitude\": \"1\"},{\"id\": \"1\",\"name\": \"1\",\"description\": \"1\",\"imgUrl\": \"1\",\"longitude\": \"1\",\"latitude\": \"1\"}]", |
| ] |
| let stores = JsonAdapter.getStores(jsonStores) |
| XCTAssertTrue(stores.count==2, "nije vratio 2 dućana") |

}

**Kod: Klasa CoreUnitTests, metoda testParsing**

Prvo smo inicijalizirali testni json objekt koji bi klasa trebala inače dobiti od web servisa. Zatim smo parsali json objekt pomoću statičke metode getStores klase JsonAdapter.

Provjerili smo da li smo dobili točno dvije trgovine, odnosno polje sa točno dva objekta tipa Store pomoću metode XCTAssertTrue. To je metoda koja provjerava uvjet. Ukoliko uvjet nije zadovoljen test neće proći. Sljedeća funkcija koju ćemo napisati jest funkcija koja provjerava metodu getStores klase JsonAdapter.



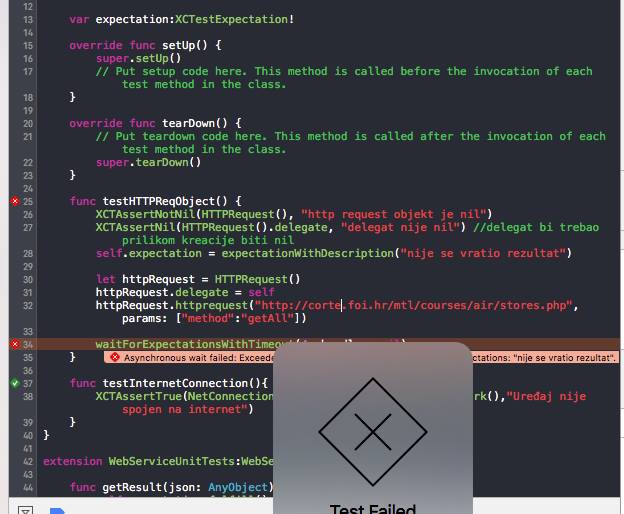
|  |
| --- |
| import XCTest |
| import core |
| import db |
| import SwiftyJSON |
|  |
| class CoreUnitTests: XCTestCase {  //…kod prije |
| |  | | --- | | func testJsonAdapterStores(){ | | let resultEmptyString = JsonAdapter.getStores("") | | let wrongObject: [String: AnyObject] = [ | | "test": "1", | | ] | | let resultWrongObject = JsonAdapter.getStores(wrongObject) | | let normalObject: [String: AnyObject] = [ | | "items": "[{\"id\": \"1\",\"name\": \"Super nova\",\"description\": \"Veliko blagdansko snizenje\",\"imgUrl\": \"cortex.foi.hr/slika2.png\",\"longitude\": \"16\",\"latitude\": \"46\"}]", | | ] | | let resultNormal = JsonAdapter.getStores(normalObject) | |  | | XCTAssertEqual(resultEmptyString.count, 0, "treba biti nula") | | XCTAssertEqual(resultWrongObject.count, 0, "treba biti nula") | |  | | XCTAssertEqual(resultNormal[0].name, "Super nova", "naziv ducana nije tocan") | |  | | } | |

}

**Kod: Klasa CoreUnitTests, metoda testJsonAdapterStores**

Tu smo pokušali parsati prvo prazni string, zatim krivi JSON objekt, zatim normalni JSON objekt. Funkcija XCTAssertEqual provjerava je li sve dobro odnosno jesu li prvi i drugi argumenti jednaki. Ukoliko nisu jednaki test ne prolazi i prikazuje se poruka koju smo naveli kao treći argument.

Kreirajmo novu klasu i nazovimo je WebServiceUnitTests, tu ćemo testirati modul ws. Dakle, dodati ćemo prvu metodu testHttpReqObject kako bi pokazali rad s funkcijom XCTAssertNotNil/XCTAssertNil. Još smo dodali funkciju koja testira spajanje na Internet. Ono što je zanimljivo jest klasa XCTestExpectation koja omogućava programerima da testiraju funkcionalnost u kojoj se ne zna kada će se točno vratiti rezultat (npr. vraćanje podataka od strane servera). U tom slučaju mogu definirati varijablu tipa XCTestExpectation te pozvati funkciju *waitForExpectationsWithTimeout* koja čeka određeni broj sekundi te ukoliko se čeka više od tog broja test će pasti. U konkretnom primjeru (vidi kod dolje) smo poslali zahtjev serveru te čekamo deset sekundi na odgovor. Ono što ostvaruje ovakav način testiranja jest funkcija *fulfill* klase *XCTestExpectation* koja određuje moment u kojem je naše očekivanje zadovoljeno te se tada poziva. U našem slučaju to je funkcija *getResult* unutar ekstenzije *WebServiceUnitTests* koja implementira dohvaćanje rezultata koji mora biti po protokolu *WebServiceResultDelegate*. Broj sekundi ovisi o programerovim očekivanjima. Ukoliko se čeka više od navedenog broja sekundi test pada te dobivamo poruku : „Asynchronous wait failed“ sa popratnom porukom da je prijeđen broj sekundi te našom personaliziranom porukom (vidi sliku i kod dolje).



Funkcija XCTAssertNotNil provjerava da li je prvi argument inicijaliziran (nije nil) te ukoliko je onda vraća poruku koja je definirana kao drugi argument, suprotno radi funkcija XCTAssertNil.



|  |
| --- |
| import XCTest |
| import ws | | | |
| class WebServiceUnitTests: XCTestCase { | | | |
|  | | | |
| var expectation:XCTestExpectation! | | | |
|  | | | |
| override func setUp() { | | | |
| super.setUp() | | | |
| // Put setup code here. This method is called before the invocation of each test method in the class. | | | |
| } | | | |
|  | | | |
| override func tearDown() { | | | |
| // Put teardown code here. This method is called after the invocation of each test method in the class. | | | |
| super.tearDown() | | | |
| } | | | |
|  | | | |
| func testHTTPReqObject() { | | | |
| XCTAssertNotNil(HTTPRequest(), "http request objekt je nil") | | | |
| XCTAssertNil(HTTPRequest().delegate, "delegat nije nil") //delegat bi trebao prilikom kreacije biti nil | | | |
| // Dio koda sa expectation fulfill paradigmom  self.expectation = expectationWithDescription("nije se vratio rezultat") | | | |
| httpRequest.delegate = self | | |
| httpRequest.httprequest("http://cortex.foi.hr/mtl/courses/air/stores.php", params: ["method":"getAll"]) | | |
|  | | |
| **waitForExpectationsWithTimeout(10, handler: nil)** | |
| } | |
|  | |
| func testInternetConnection(){ | |
| XCTAssertTrue(NetConnection.Connection.isConnectedToNetwork(),"Uređaj nije spojen na internet") | |
| } |
| } |
|  |
| extension WebServiceUnitTests:WebServiceResultDelegate{ |
|  |
| func getResult(json: AnyObject) { |
| **self.expectation.fulfill()** |
| } | } | | | |
| } |  | | | |

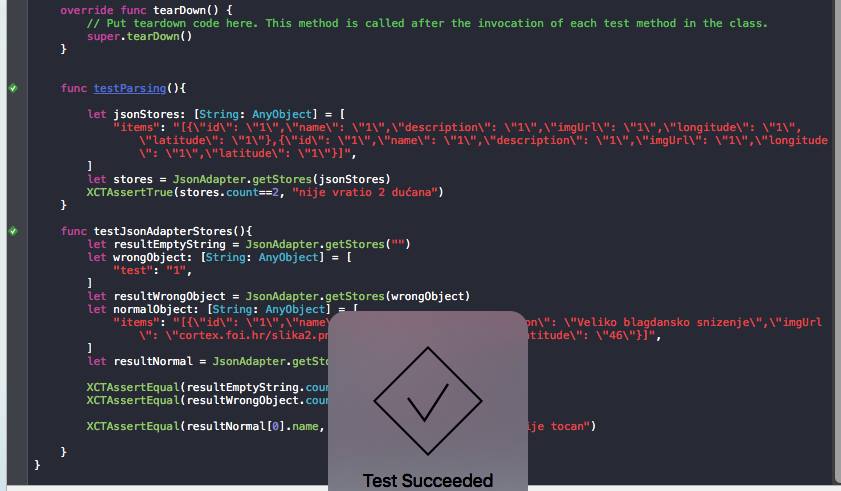
**Kod: Klasa WebServiceUnitTests**

Pokretanje napisanih testnih funkcija može se uraditi na više načina. Može se pokrenuti na način tako da se pokrenu sve metode testne klase pritiskom na gumb pored naziva testne klase ili na način da se pokrene sama metoda također pritiskom na gumb pored naziva metode kojom želite testirati (vidljivo na sljedećim slikama).

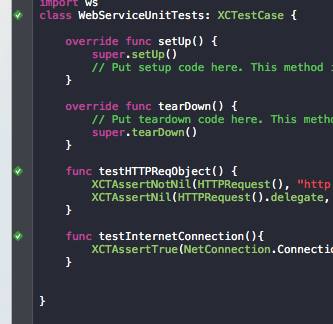


Slika 9: neuspješni test

Ukoliko neki od assertova padne javit će nam poruku koju smo definirali i obilježiti točno taj assert koji je pao.



Slika 10: Uspješni test



Slika 11: Svi testovi uspješni

Ukoliko su svi testovi uspješni pokraj klase će biti zelena kvačica kao i pokraj svih njezinih metoda. Za kraj unit testova slijedi pregled prednosti i nedostataka unit testova.

|  |  |
| --- | --- |
| **Prednosti** | **Nedostaci** |
| Pouzdanost: Možete demonstrirati da vaš kod zbilja radi. | Više koda: U projektima koji zahtjevaju iscrpno testiranje moguće je da ćete imati više testnog koda nego pravog koda. |
| Brzi feedback: Možete testirati vaš kod koji je “zakopan” slojevima daleko od vaše navigacije u aplikaciji. | Veće održavanje: Više koda = više testnog koda. |
| Modularnost: Unit testovi vas potiču da pišete modularniji kod . | Nema garancije: Unit testovi ne osiguravaju da je vaš kod “bug-free” |
| Koncentracija na detalje: pisanje unit testova ispituje najmanje moguće funkcionalnosti | Zahtjeva vrijeme: za vrijeme pisanje unit testova mogli ste raditi na nekim funkcionalnostima aplikacije. |
| Regresija: budite sigurni da bugovi koje ste ispravili ostanu ispravljeni – neće se opet otvoriti s novim ispravcima. |  |
| Refaktoriranje: služe za validaciju refaktoriranja koda. |  |
| Dokumentacija: Unit testovi mogu poslužiti kao pomoć pri pisanju dokumentacije – definiraju točno što vaš kod treba činiti. |  |

* 1. UI Testovi

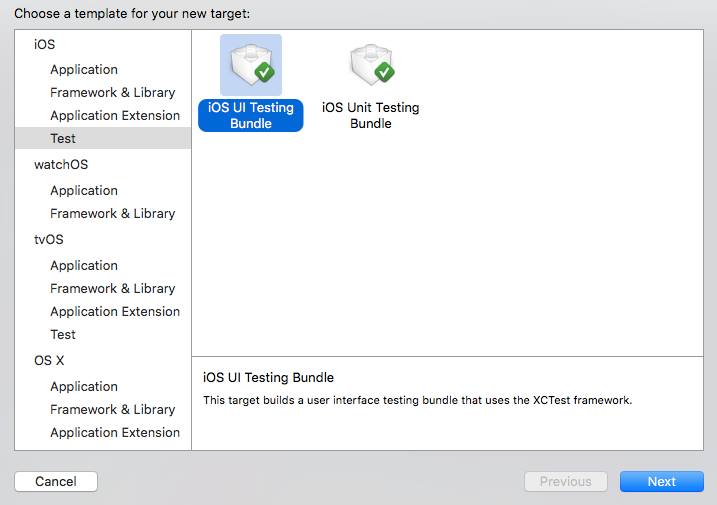
UI testovi ili testovi korisničkog sučelja također se nalaze unutar XCodeovog frameworka **XCTest**. Od Xcodea verzije 7 postoji API ka sučelju naše aplikacije zahvaljujuči XCTest modulu. Novi API omogućuje pisanje UI testova tako da snimimo interakciju s aplikacijom te automatsko generiranje testnog koda na temelju iste. Proces snimanja interakcije ne obuhvaća postavljanje Assertova već to korisnik može sam u kodu dodati. Moramo priznati da trenutna verzija API-a za testiranje ima bugova no nadamo se da će to ubrzo ispraviti pošto je ovo relativno nova opcionalnost unutar XCodea.

Kao i kod unit testova trebamo dodati novi modul (target) koji služiti isključivo za UI testove (vidi sljedeću sliku).



|  |
| --- |
| import XCTest |
| import ws | | | |
| class WebServiceUnitTests: XCTestCase { | | | |
|  | | | |
| var expectation:XCTestExpectation! | | | |
|  | | | |
| override func setUp() { | | | |
| super.setUp() | | | |
| // Put setup code here. This method is called before the invocation of each test method in the class. | | | |
| } | | | |
|  | | | |
| override func tearDown() { | | | |
| // Put teardown code here. This method is called after the invocation of each test method in the class. | | | |
| super.tearDown() | | | |
| } | | | |
|  | | | |
| func testHTTPReqObject() { | | | |
| XCTAssertNotNil(HTTPRequest(), "http request objekt je nil") | | | |
| XCTAssertNil(HTTPRequest().delegate, "delegat nije nil") //delegat bi trebao prilikom kreacije biti nil | | | |
| // Dio koda sa expectation fulfill paradigmom  self.expectation = expectationWithDescription("nije se vratio rezultat") | | | |
| httpRequest.delegate = self | | |
| httpRequest.httprequest("http://cortex.foi.hr/mtl/courses/air/stores.php", params: ["method":"getAll"]) | | |
|  | | |
| **waitForExpectationsWithTimeout(10, handler: nil)** | |
| } | |
|  | |
| func testInternetConnection(){ | |
| XCTAssertTrue(NetConnection.Connection.isConnectedToNetwork(),"Uređaj nije spojen na internet") | |
| } |
| } |
|  |
| extension WebServiceUnitTests:WebServiceResultDelegate{ |
|  |
| func getResult(json: AnyObject) { |
| **self.expectation.fulfill()** |
| } | } | | | |
| } |  | | | |

**Kod: Klasa WebServiceUnitTests**



Potrebno je samo pokrenuti snimanje interakcija klikom na „Record UI Test“ gumb (vidi sljedeću sliku).

https://scontent-vie1-1.xx.fbcdn.net/hphotos-xpf1/v/t34.0-12/12575991_10207355388196722_1240092527_n.jpg?oh=e64ae5abf0031c0b835fef1bca9ff6fd&oe=56A5ECAA

U našem primjeru ćemo testirati:

* otvaranje jedne trgovine
* odabir popusta
* zatvaranje dućana
* otvaranje mape
* odabir jednog popusta preko mape.

Ukoliko želimo da nam aplikacija pričeka neko vrijeme da dođe odgovor s web servisa možemo staviti naredbu *NSThread.sleepForTimeInterval(x)* gdje je x broj sekundi. U protivnom će se test srušiti budući da se nisu stigle generirati trgovine. Kod koji nam je XCode generirao je sljedeći:



|  |
| --- |
| import XCTest |
|  |
| class Discount\_LocatorUITests: XCTestCase { |
|  |
| override func setUp() { |
| super.setUp() |
|  |
| // Put setup code here. This method is called before the invocation of each test method in the class. |
|  |
| // In UI tests it is usually best to stop immediately when a failure occurs. |
| continueAfterFailure = false |
| // UI tests must launch the application that they test. Doing this in setup will make sure it happens for each test method. |
| XCUIApplication().launch() |
|  |
| // In UI tests it’s important to set the initial state - such as interface orientation - required for your tests before they run. The setUp method is a good place to do this. |
| } |
|  |
| override func tearDown() { |
| // Put teardown code here. This method is called after the invocation of each test method in the class. |
| super.tearDown() |
| } |
|  |
| func testExample() { |
|  |
| let app = XCUIApplication() |
| NSThread.sleepForTimeInterval(4) |
| app.tables.staticTexts["Super Nova"].tap() |
| app.tables.staticTexts["For two paid products, the third product (with the lowest price) is given for free in all stores of Super Nova shopping center."].tap() |
| app.alerts["📈 Three for two : 30%"].collectionViews.buttons["Close"].tap() |
| app.navigationBars["Discounts"].childrenMatchingType(.Button).matchingIdentifier("Back").elementBoundByIndex(0).tap() |
|  |
| app.navigationBars["Discount\_Locator.VcNav"].buttons["Image"].tap() |
| app.tables.staticTexts["Map"].tap() |
| app.otherElements["Varteks, Popust na sve kolekcije ljetne odjeu0107e u Varteks Outlet duu0107anu."].tap() |

}

}

**Kod: Klasa DiscountLocatorUITests**

Opaska: ukoliko u naslovima na koje bi simulator testa trebao stisnuti imate emoji/simbole prilikom generiranja koda snimač neće dobro generirati naslov na koji treba stisnuti npr. umjesto znaka 📈 će generirati string „\ud83d\udcc8“ (kod za taj simbol). To će uzrokovati rušenje testa prilikom pokretanja istog stoga je potrebno ručno zamijeniti string koda sa stvarnim simbolom (edit->emojis and symbols). Nažalost, trebamo pričekati da se te neke stvari poslože na svoje budući da je ovo novi sustav za testiranje.