iOS Tutorial - aplikacija Discount Locator

**Tim T20:**

Darijan Vertovšek

Danijel Vincijanović

Hrvoje Hrvoić

**Poveznica na GitHub repo:**

<https://github.com/davinci2015/DiscountLocatorIOS>

1. ARHITEKTURA IOS 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

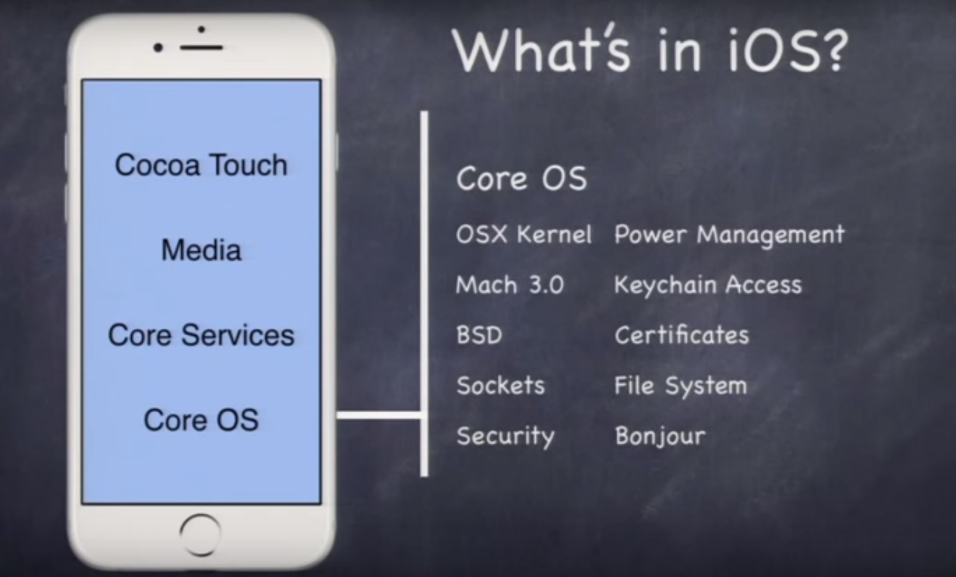


GITHUB LINK

<https://goo.gl/WCNkeX>

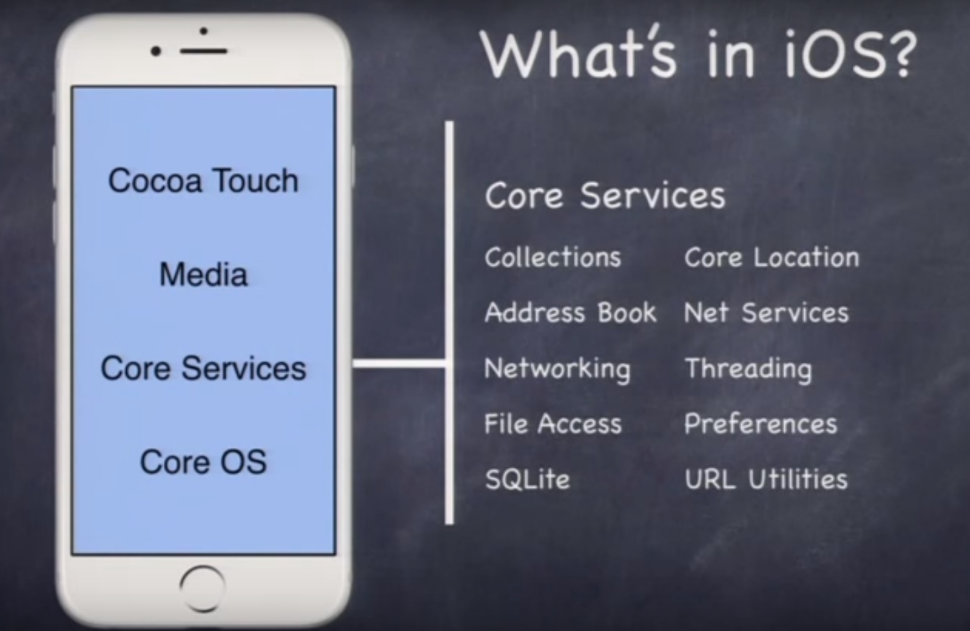
**RAZRADA**

* 1. Arhitektura iOS sustava

****

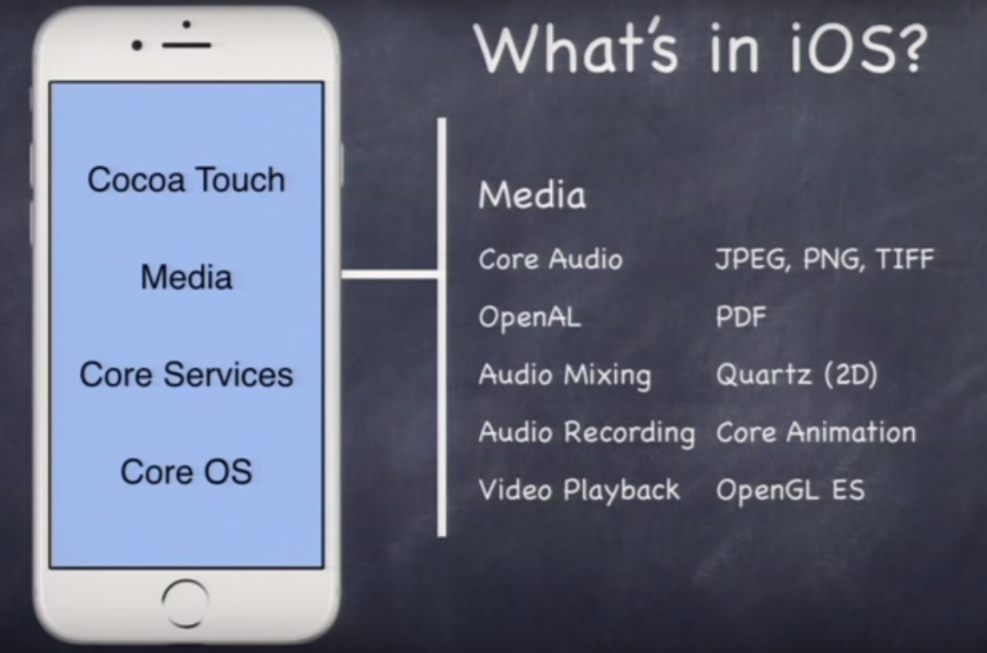
Slika : Arhitektura iOS-a: Core OS

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:

****

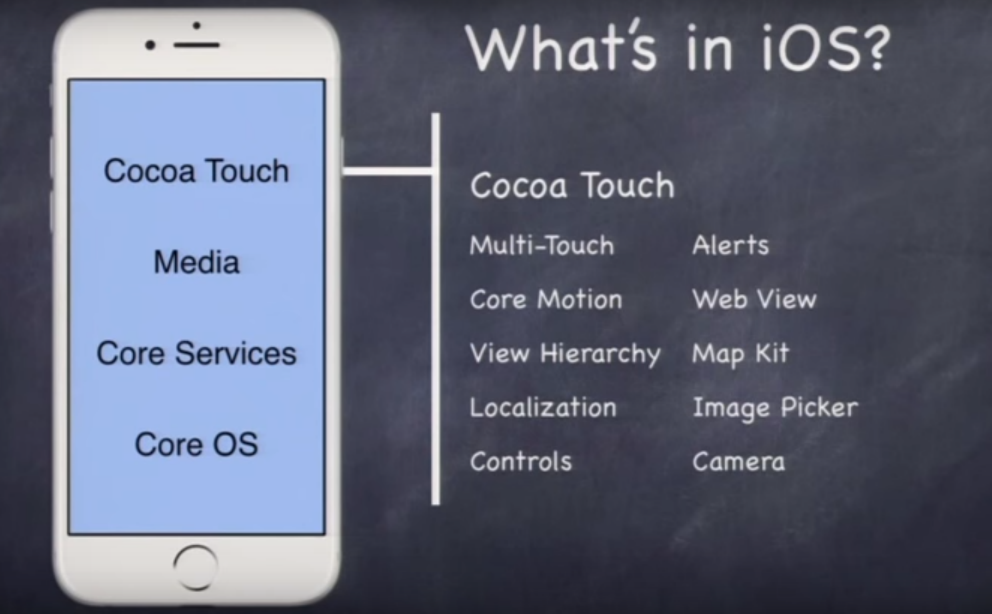
Slika : Sloj Core Services

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 : 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 . 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.

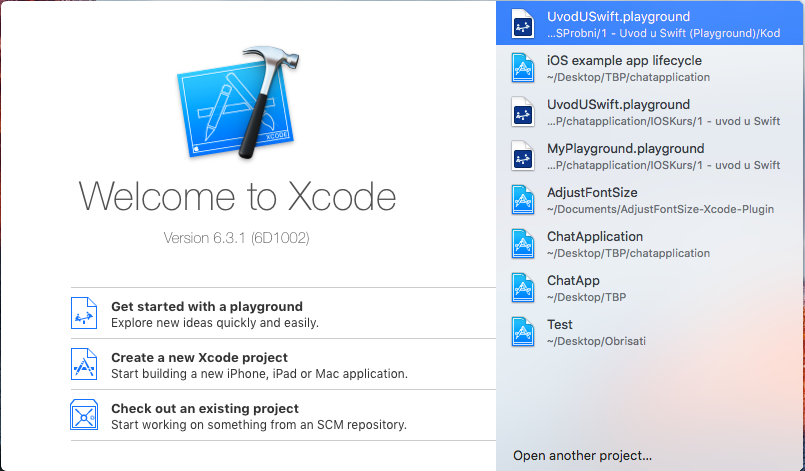
Get started with a playground - Playground jest alat u sklopu Xcode-a koji služi za real-time kompajliranje Swift programa. Svrha Playgrounda je učenje i isprobavanje programskog jezika Swift.

Create a new Xcode project - ovom opcijom pokrećemo novi projekt

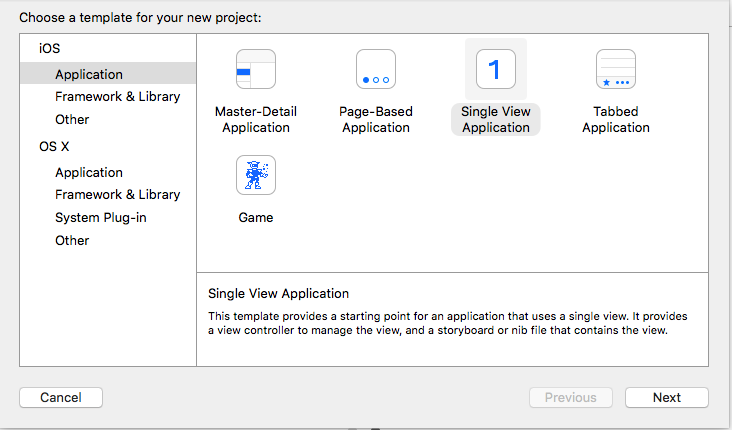
Check out an existing project - pokretanje postojećeg projekta iz repozitorija sa sustava za verzioniranje koda

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.



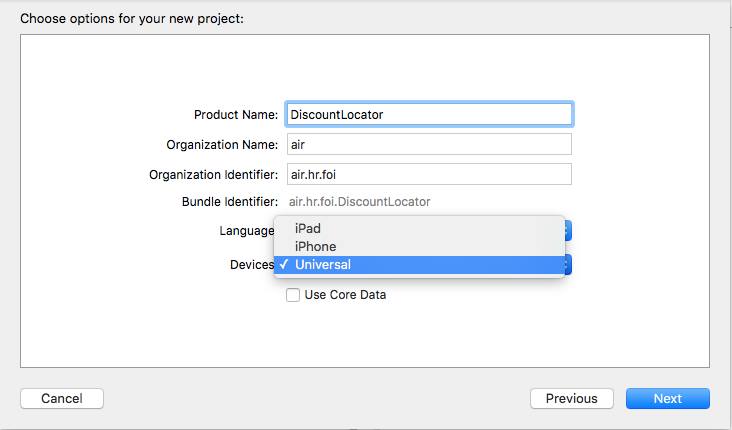
S lijeve strane se nalazi ponuđeni izbornik gdje biramo koju vrst projekta izrađujemo. Mi ćemo uvijek odabirati aplikaciju za iOS platformu. S desne strane imamo odabir predloška aplikacije. U svrhu upoznavanja sa sučeljem, za sada ćemo odabrati Single View Application. Radi se o jednostavnoj MVC aplikaciji.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



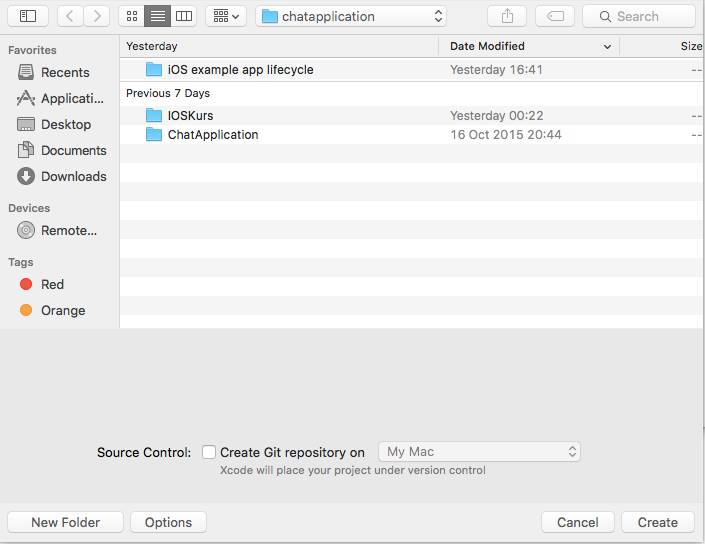
Sada odabiremo ime proizvoda, organizaciju te identifier. Koja je svrha ovoga? Na temelju organization identifiera, generira se Bundle identifier te, kada budemo objavljivali aplikaciju za Apple store, to predstavlja jedinstveni identifikator aplikacije. Može postojati više aplikacija sa istim imenom, 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.

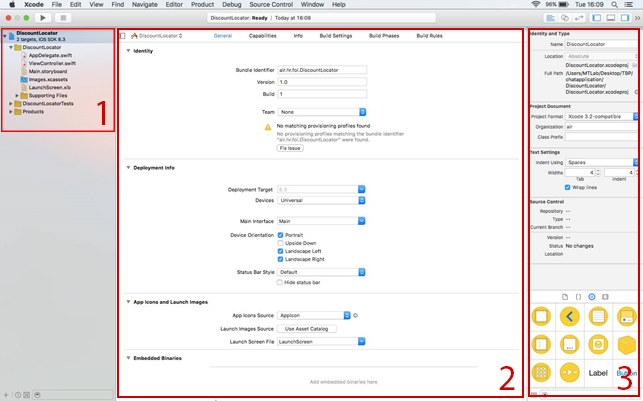
Osim toga, moguće je odabrati za koje uređaje razvijamo aplikaciju (vidi sljedeću sliku).



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.

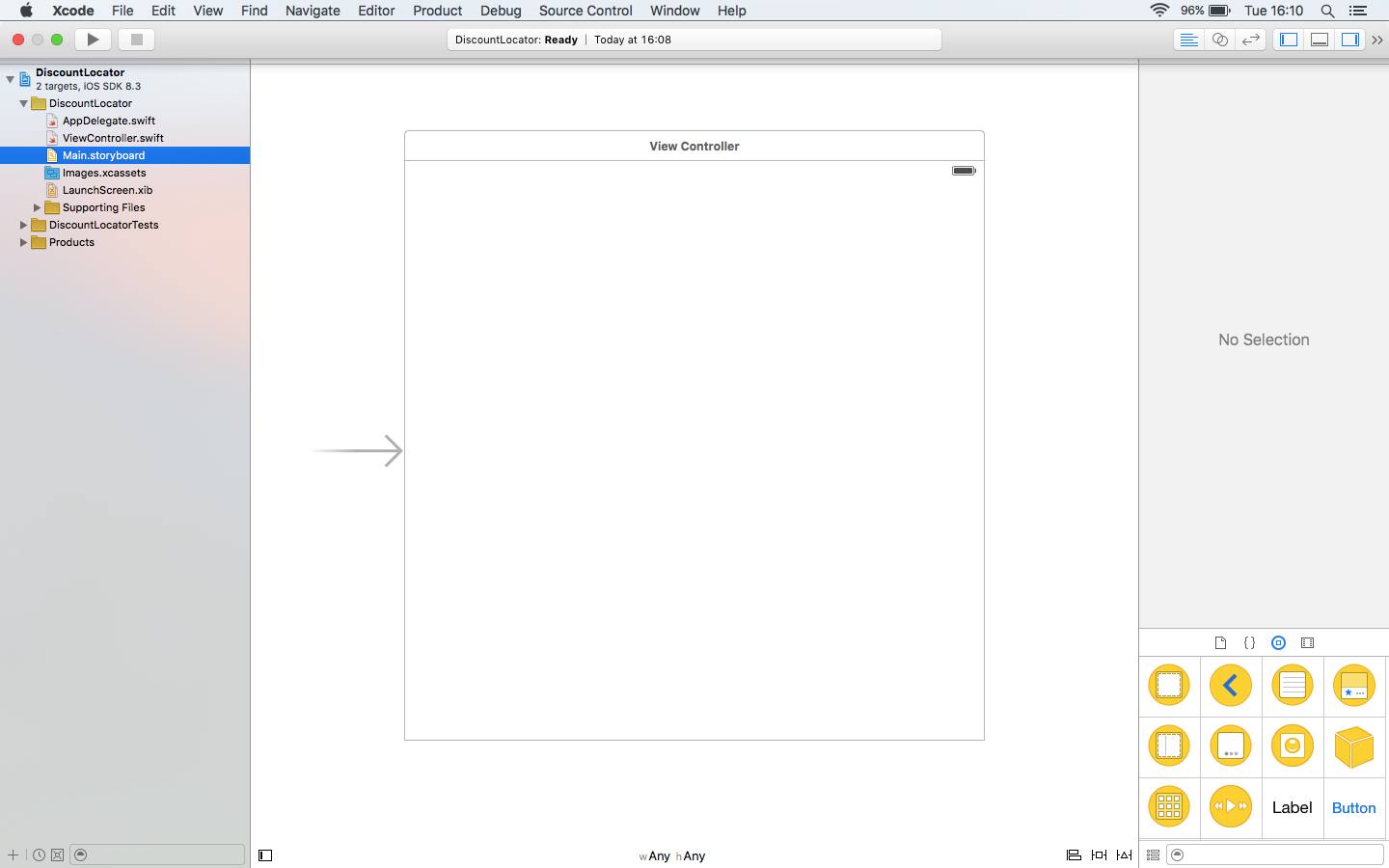


Potrebno je zamijetiti sljedeće dijelove IDE-a:

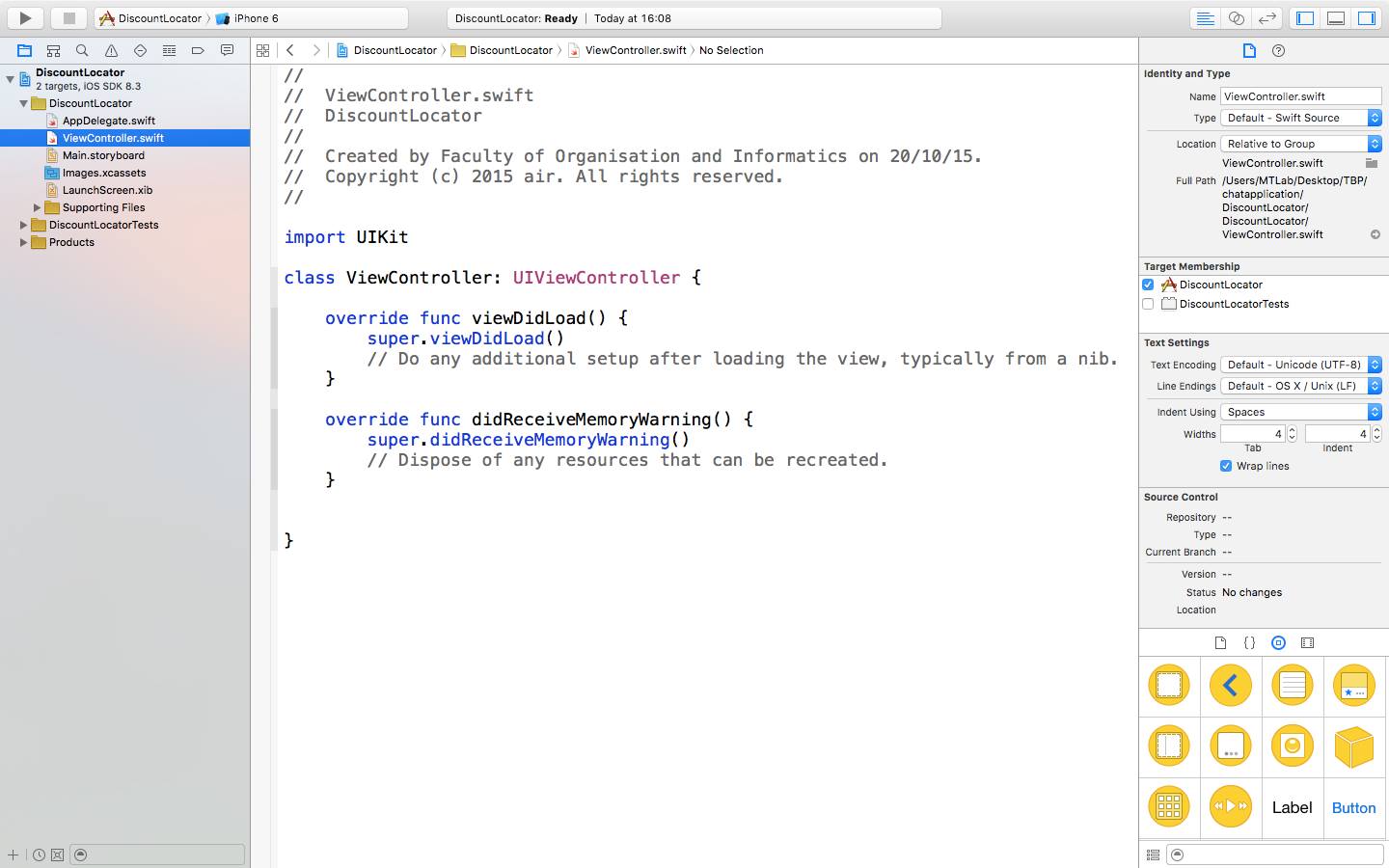
[1] – Struktura projekta koja prikazuje hijerarhiju foldera i njemu pripadnih datoteka unutar projekta. Dvije najbitnije datoteke su Main.storyboard te ViewController.swift. U MVC svijetu Main.storyboard je view, a ViewController.swift kontroler koji upravlja tim view-om (Apple forsira MVC arhitekturu).

[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.

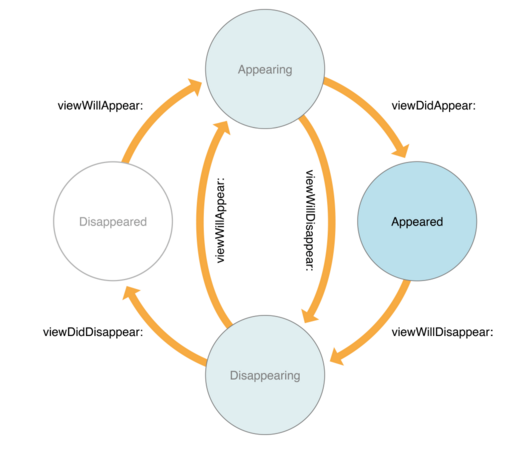
Unutar ViewController-a također postoji par metoda koje se koriste kako bi se upravljalo događajima životnog ciklusa, ali u ovom slučaju ne aplikacije, nego pogleda.

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)

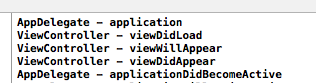
Metoda **viewWillAppear** se poziva prije nego što će se pogled pokazati i koristi se za ažuriranje podataka na formi.

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.

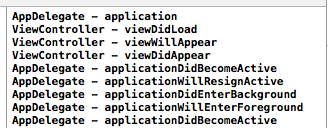


Slika . Životni ciklus pogleda

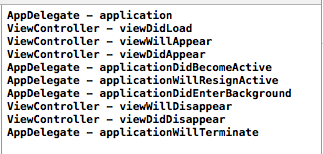
Prilikom pokretanja aplikacije, prvo se pokreće metoda **application**, zatim se pokreće view unutar ViewControllera, te zatim opet unutar klase AppDelegate metoda **appliationDidBecomeActive**.



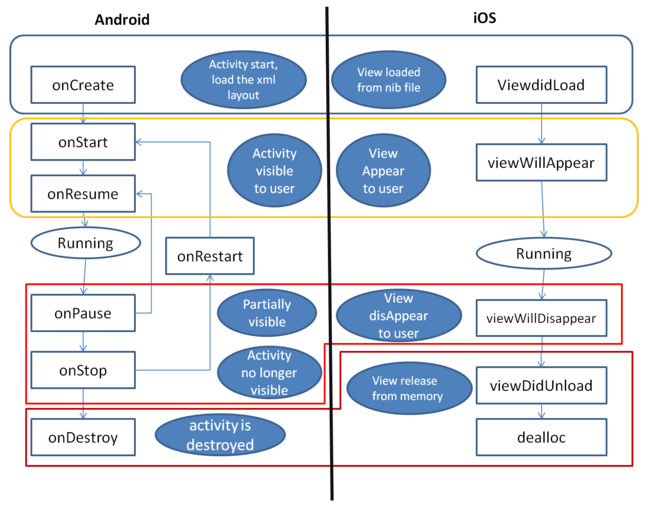
Kada pauziramo aplikacija (pritisak na tipku 'Home'), pozivaju se dvije metode (applicationWillResignActive i applicationDidEnterBackground), te prilikom ponovnog vraćanja aplikacije u fokus također dvije metode (applicationWillEnterForeground te ponovno metoda application DidBecomeActive)



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 . 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:



|  |
| --- |
|  |

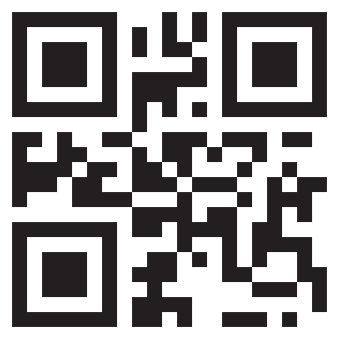
|  |
| --- |
| enum Planet { |
| case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune |
| } |
|  |
| var planet = Planet.Earth //var planet je tipa Planet |
|  |
| switch planet { |
| case .Earth: print ("Pozdrav Zemljanine") |
| default: print ("Pozdrav Izvanzemaljče o.O") |
| } |

**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 : UPC-A kod



Slika : 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.

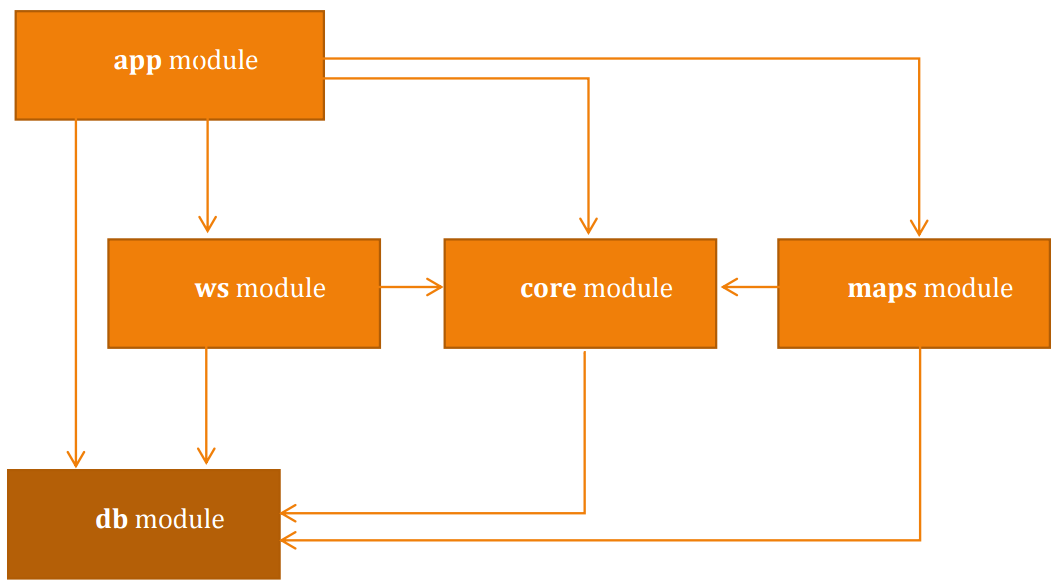
**ISHODI UČENJA**

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'

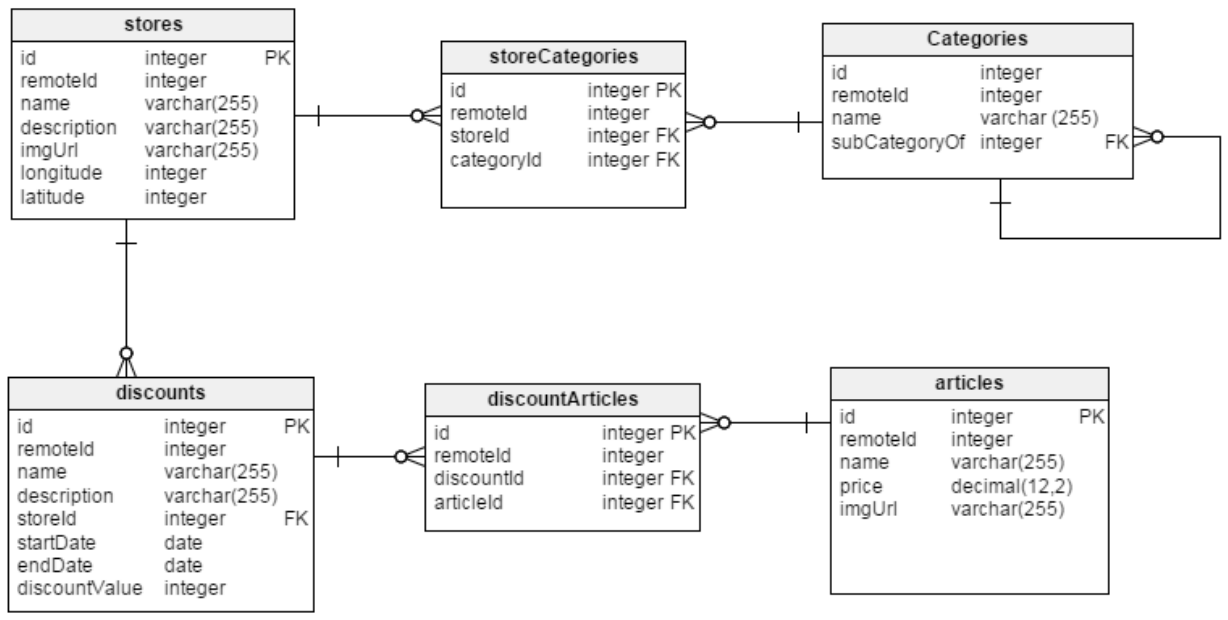
**Podfile** je tekstualna datoteka u kojoj 'opisujemo' koji kod odnosno ovisnosti želimo povući u projekt. Ovisnosti su sada spremne za instalaciju pa u konzoli pokrenemo sljedeću naredbu

$ 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: Int = 0 |
|  |
| 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: Int = 0 |
|  |
| 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
5. 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>

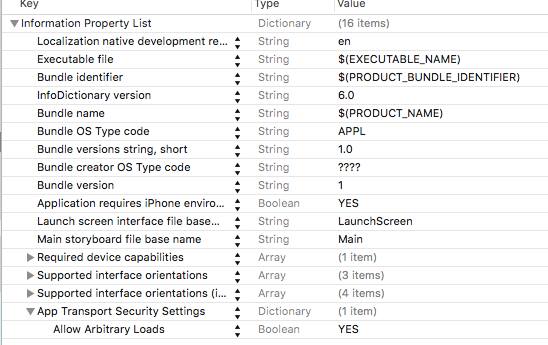
<key>NSAllowsArbitraryLoads</key>

<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, 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 |
| { |
| public static func getStores(json: AnyObject) -> Array<Store> |
| { |
| var stores = [Store]() |
| let jsonStores = JSON(json) |
|  |
| let items = jsonStores["items"] |
|  |
|  |
| for (key, subJson) in items { |
| var s:Store = Store() |
| s.remoteId = subJson["id"].int! |
| s.name = subJson["name"].string! |
| s.desc = subJson["description"].string! |
| s.imgUrl = subJson["imgUrl"].string! |
| s.latitude = subJson["latitude"].float! |
| s.longitude = subJson["longitude"].float! |
|  |
| stores.append(s) |
| } |
|  |
| return stores |
| } |
|  |

**Kod: Klasa JsonAdapter**

Klasa JsonAdapter sadrži dvije statičke metode – *getStores* i *getDiscounts*. Gore je navedena implementacija *getStores* metode, a implementacije *getDiscounts* je identična. Kao parametar prima *AnyObject* odnosno ono što nam vrati servis kao odgovor. Metoda vraća polje *Store-ova*.

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*, puniti ga podacima iz JSON-a i stavljati u listu.

1. **Dodavanje i implementacija WebServiceDataLoader klase (Discount Locator modul)**

Spremni smo za kreiranje i implementaciju klase WebServiceDataLoader.



|  |
| --- |
|  |
|  |
| import Foundation |
| import db |
| import core |
| import ws |
| public class WebServiceDataLoader:DataLoader |
| { |
| private var discountsLoaded: Bool = false |
| private var storesLoaded: Bool = false |
| public var stores: [Store] = [] |
| public var discounts: [Discount] = [] |
|  |
| public func LoadData() { |
| HTTPRequest.sharedWSInstance.httprequest("http://www.json-generator.com/api/json/get/csbvEnjqnC") |
| { |
| (result: AnyObject) in |
| self.stores = JsonAdapter.getStores(result) |
| self.storesLoaded = true |
| self.showLoadedData() |
| } |
| HTTPRequest.sharedWSInstance.httprequest("http://www.json-generator.com/api/json/get/ccWtDCAmRe") |
| { |
| (result: AnyObject) in |
| self.discounts = JsonAdapter.getDiscounts(result) |
| self.discountsLoaded = true |
| self.showLoadedData() |
| } |
| } |
|  |
| private func showLoadedData() |
| { |
| if(storesLoaded && discountsLoaded) |
| { |
| self.bindData() |
| } |
| } |

**Kod: Klasa WebServiceDataLoader I. dio**

…nastavak implementacije



|  |
| --- |
|  |
|  |
| private func bindData() |
| { |
| DbController.sharedDBInstance.realm.beginWrite() |
| DbController.sharedDBInstance.realm.deleteAll() |
| try! DbController.sharedDBInstance.realm.commitWrite() |
|  |
| for store in stores |
| { |
| DbController.sharedDBInstance.realmAdd(store) |
| for discount in discounts |
| { |
| if(discount.storeId == store.remoteId) |
| { |
| discount.storeSet(store) |
| DbController.sharedDBInstance.realmAdd(discount) |
| } |
| } |
| } |
|  |
| print(DbController.sharedDBInstance.realm.objects(Store)) |
| print(DbController.sharedDBInstance.realm.objects(Discount)) |
| } |

**Kod: Klasa WebServiceDataLoader II. dio**

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

Klasa sadrži tri funkcije – *LoadData* (funkcija implementirana iz protokola koja vrši zahtjeve prema servisu), *showLoadedData* (nakon što su izvršena oba zahtjeva 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)

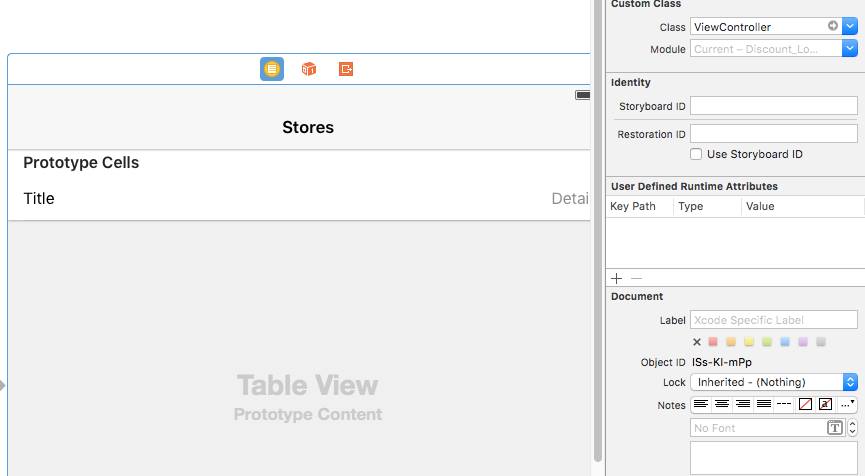


Budući da metoda httprequest iz klase HTTPRequest prima closure kao argument proučite isti na sljedećoj poveznici:

<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() |
| webServiceDataLoader.storesTableView = self.storesTableView |
| webServiceDataLoader.LoadData() |
| } |
|  |
| 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 UITableViewCell |
| var store: Store |
| store = webServiceDataLoader.stores[indexPath.row] |
| cell.textLabel!.text = store.name |
| cell.detailTextLabel!.text = store.desc |
| return cell |
| } |
|  |

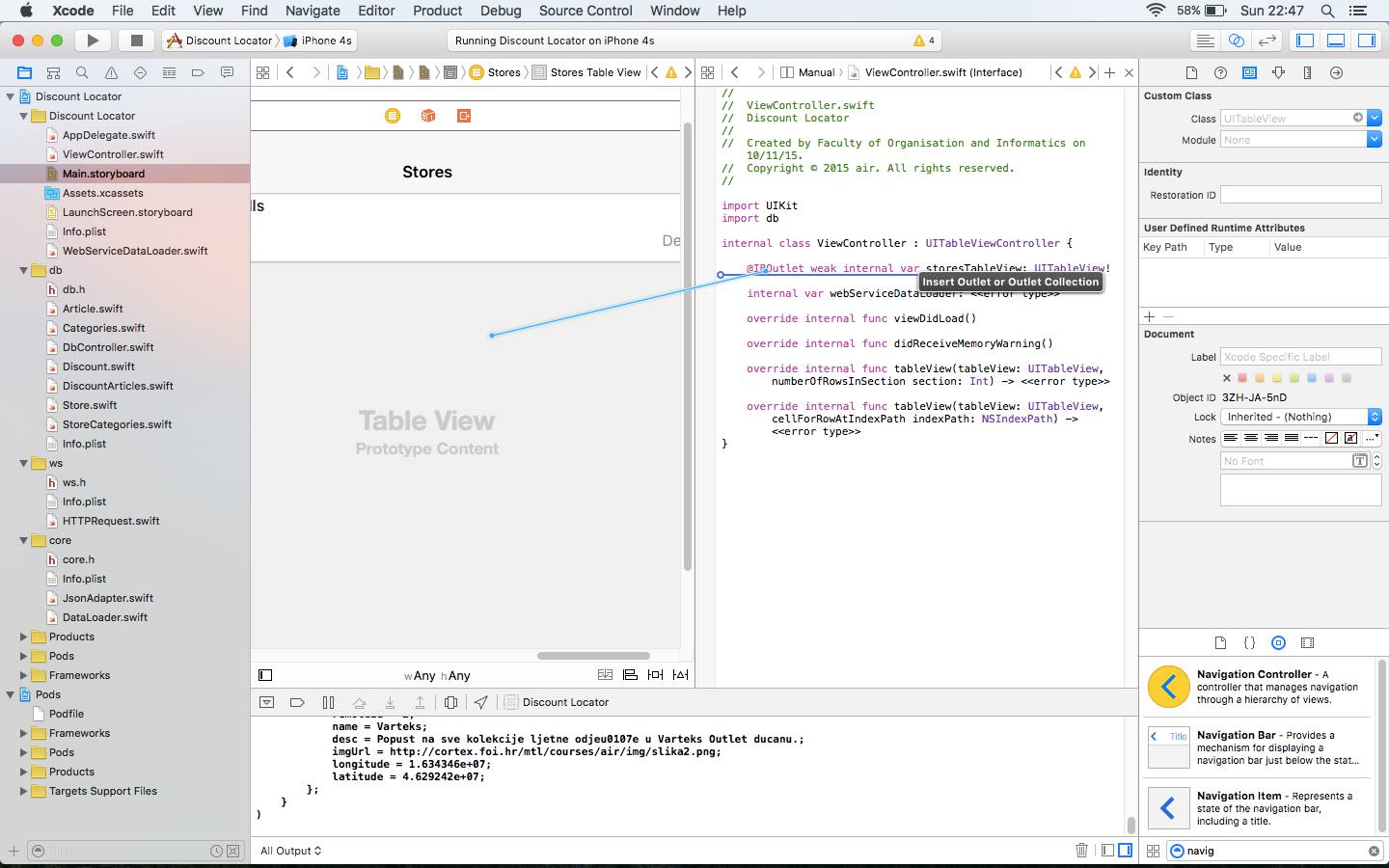
**Kod: Izmjena klase ViewController**

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

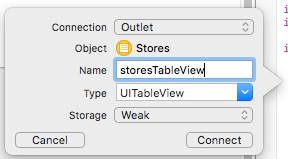
Zatim ćemo otvoriti *main.storyboarda* kako bi povukli table view unutar našeg kontrolera te ih povezali. Kliknimo na *assistant editor* odnosno na srednju ikonu sa slike:

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 servicu u njih učitavamo podatke pa je potrebno osvježiti TableView.

Sada možemo build-ati projekt i naša lista sadrži.