Razvoj cross-platform native mobilnih aplikacija u programskom jeziku C#

Native cross-platform mobile application development in C# programming language

Master rad

Nemanja Milošević

Mentor: dr Miloš Racković

2016.

Sadržaj

1. Predgovor

Ovaj master rad se bavi proučavanjem načina na koji je moguće stvarati mobilne aplikacije koje rade na svim popularnim mobilnim platformama. U trenutku pisanja ovog rada, to su: Android, Windows Phone i iOS. U današnje vreme veoma je važno da mobilne aplikacije rade nezavisno od operativnog sistema koji se nalazi na samom telefonu. Ovaj rad pruža pregled u načine na koji se mogu razvijati aplikacije za sve navedene operativne sisteme uz veliki stepen deljenog koda. Važno je na početku napomenuti da se ovaj rad fokusira na razvoj nativnih (eng. native) aplikacija, a ne na aplikacije koje koriste web wrapper komponentu kako bi mogle da se izvršavaju na svim operativnim sistemima. U poslednjem delu rada opisana je implementacija jedne ovakve aplikacije na čijem primeru su opisani koncepti i najbolje prakse pri razvoju mobilnih aplikacija za više različitih operativnih sistema.

Ovaj rad je podeljen u više delova na sledeći način:

1. Poređenje nativnih i aplikacija koje koriste web-wrapper
2. Pregled raznih dostupnih biblioteka za razvoj cross-platform mobilnih aplikacija
3. Pregled Xamarin biblioteke, opis prednosti koje donosi i zašto je autor ovog rada odabrao baš ovu biblioteku
4. Pregled Xamarin.Forms biblioteke za definisanje grafičkog interfejsa aplikacija pomoću markup jezika XAML
5. Opis implementacije aplikacije za studentske servise razvijene pomoću Xamarin biblioteke u programskom jeziku C#

2. Cross-platform mobilne aplikacije sa Web tehnologijama

Veoma popularan način razvoja cross-platform mobilnih aplikacija je korišćenjem takozvanih web-wrapper framework-a. Glavna prednost ovog pristupa je jednostavnost: napravi se web aplikacija koja se zatim lokalno učitava sa mobilnog telefona. Posebne mogućnosti telefona kao što su kamera, kompas, GPS i slično se mogu koristiti sa nekim ograničenjima. Glavna mana ovakvog pristupa razvoju mobilnih aplikacija je to što se aplikacija grafički ne uklapa sa izgledom operativnog sistema. Takođe, performanse su ograničene, pristup senzorima je ograničen, i često nije moguće u potpunosti realizovati aplikaciju kako je zamišljena.

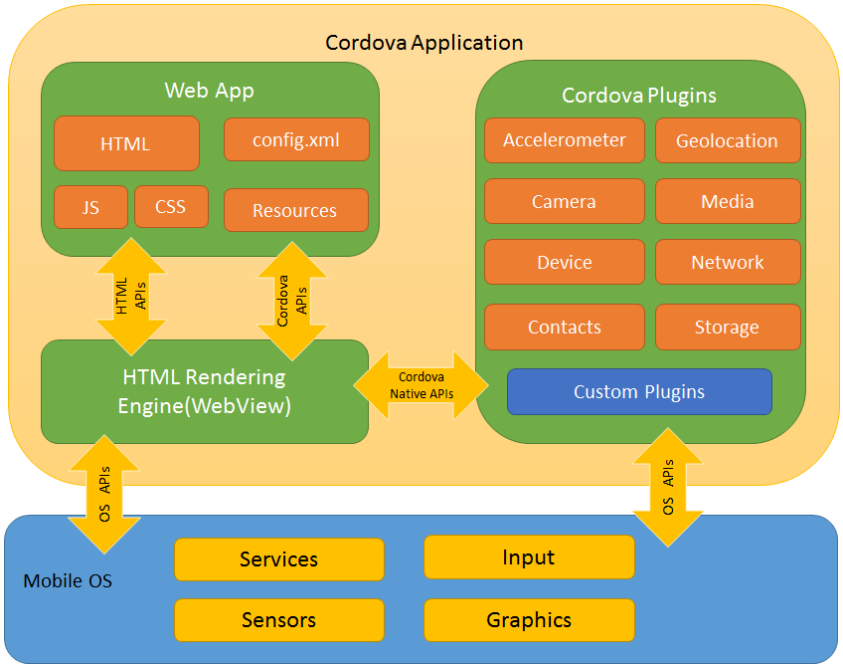
Ipak, ovaj pristup razvoju mobilnih aplikacija ima smisla ukoliko se razvija mala, specifična aplikacija koja ne zahteva pristup senzorima i u kojoj nije važan izgled komponenti.

Postoji nekoliko veoma popularnih biblioteka za razvoj web-wrapper aplikacija. Uglavnom se sve razvijaju koristeći jezike HTML5, CSS i JavaScript ili neki od njegovih derivata koji se može konvertovati u JavaScript – TypeScript, Dart i slični.

Ovde će biti navedene tri najpopularnije i najrazvijenije biblioteke u trentuku pisanja ovog rada.

2.1. Apache Cordova

Verovatno najpoznatija biblioteka za razvoj mobilnih aplikacija uz pomoć Web tehnologija. Apache Cordova je besplatna, open-source biblioteka koja programerima pruža načine za pakovanje aplikacija u pakete za mobilne operativne sisteme, kao i interfejse za pristup senzorima, kameri i drugim komponentama mobilnih telefona. Apache Cordova je veoma modularna biblioteka, te se u osnovnom paketu dobijaju samo osnovne funkcionalnosti, dok se sve ostalo nalazi u Apache Cordova dodacima (plugin-ovi). Apache Cordova podrazumeva da se aplikacije pišu korišćenjem HTML/CSS/JS razvojnih alata. Arhitekturu Apache Cordova projekta najlakše je objasniti pomoću sledećeg prikaza:



Kao što se na slici vidi, Apache Cordova implementira WebView komponentu koja služi za prikaz korisničkog interfejsa, kao i posebne interfejse kojima se pristupa komponentama mobilnog operativnog sistema na kom se aplikacija izvršava. U gornjem desnom uglu su prikazani neki od dodataka, a važno je napomenuti da je moguće implementirati i druge dodatke.

Velika prednost Apache Cordova biblioteke (i drugih sličnih biblioteka) je jednostavnost upotrebe. Ona ne zahteva nikakve posebne alate za razvoj, dovoljno je samo poznavanje HTML/CSS/JS jezika. Takođe, moguće je koristiti i popularne JavaScript biblioteke kao što su Angular ili KnockoutJS koje doprinose čistoći i preglednosti koda.

2.2. NativeScript

NativeScript je nova biblioteka koju je razvila američka kompanija Telerik, koja se dugi niz godina bavi razvojem izuzetno kvalitetnih grafičkih komponenti (između ostalog) koje se mogu koristiti u nekoliko programskih jezika.

NativeScript je takođe potpuno besplatna i open-source biblioteka koja kao i Apache Cordova koristi JavaScript/HTML i CSS za pravljenje cross-platform mobilnih aplikacija. NativeScript trenutno podržava Android verziju 4.2 i više i iOS verziju 7.1 i više, dok je podrška za Windows telefone u razvoju. Razvoj je moguć na bilo kojem Desktop operativnom sistemu (Windows/Linux/OS X).

Glavna razlika i prednost NativeScript-a je što ne koristi WebView za prikaz grafičkih komponenti u aplikacijama, već koristi nativne komponente operativnog sistema. Takođe NativeScript pored funkcija koje jednostavno mapiraju pozive na funkcije operativnog sistema sadrži i funckije koje olakšavaju razvoj kao što su funkcije za prikaz poruka, mrežnu komunikaciju i

slično. Ovo omogućava programeru da može da koristi ove funkcionalnosti bez da nužno zna kako one funkcionišu na nivou mobilnog operativnog sistema.

NativeScript je još uvek u ranoj fazi razvoja, i ne postoji mogućnost korišćenja ni jednog drugog jezika za razvoj osim JavaScript-a i njegovih derivata, što je velika mana ukoliko se razvijaju komplikovanije, enterprise aplikacije.

Ipak, ova biblioteka je za kratko vreme stekla veliku popularnost, i već se može pronaći dosta kvalitetnih i popularnih aplikacija koje koriste ovu biblioteku.

2.3. React Native

React Native je još jedna biblioteka koja koristi Web tehnologije za razvoj cross-platform mobilnih aplikacija. Razvila ju je kompanija Fejsbuk, i još uvek je u ranoj fazi razvoja. React Native se bazira na React biblioteci koja se koristi u razvoju Web aplikacija. Kao i NativeScript, React Native se ne izvršava u WebView komponenti već se grafički interfejs mapira na nativne komponente mobilnog operativnog sistema. Takođe, React Native, kao i NativeScript, definiše svoje interfejse koji objedinjuju elemente kao što su kontrola senzora, kamere i slično na različitim operativnim sistemima, što omogućava programeru da implementira ove delove aplikacije bez da nužno zna kako su oni definisani u samom mobilnom operativnom sistemu. React Native podržava iOS i Android.

Glavna prednost i novina koju React Native donosi je mogućnost da se komponente implementiraju direktno za specifičnu mobilnu platformu u programskim jezicima Java (za Android) i Swift ili Objective-C (za iOS). Ovo znači da programer ima veliku fleksibilnost u poređenju sa drugim JavaScript bibliotekama, kao i da je moguća implementacija kritičnih komponenti u pravom nativnom smislu ukoliko postoji takva potreba.

Mobilna aplikacija za društvenu mrežu Fejsbuk implementirana je na upravo ovakav način.

3. Native biblioteke

Native biblioteke za razvoj mobilnih aplikacija su dosta komplikovanije od gore navedenih biblioteka. One se zasnivaju na korišćenju apstrakcija i moćnih oruđa modernih programskih jezika kao što su delegati, lambda izrazi, generički tipovi i slično, kako bi se napravio sloj koji prevodi kod koji programer piše u više različitih verzija koda za različite platforme.

Native biblioteke takođe zahtevaju često veoma zahtevne alate i emulatore za testiranje.

U trenutku pisanja ovog rada, postoje tri dovoljno razvijene biblioteke: CodenameOne, Kivy i Xamarin.

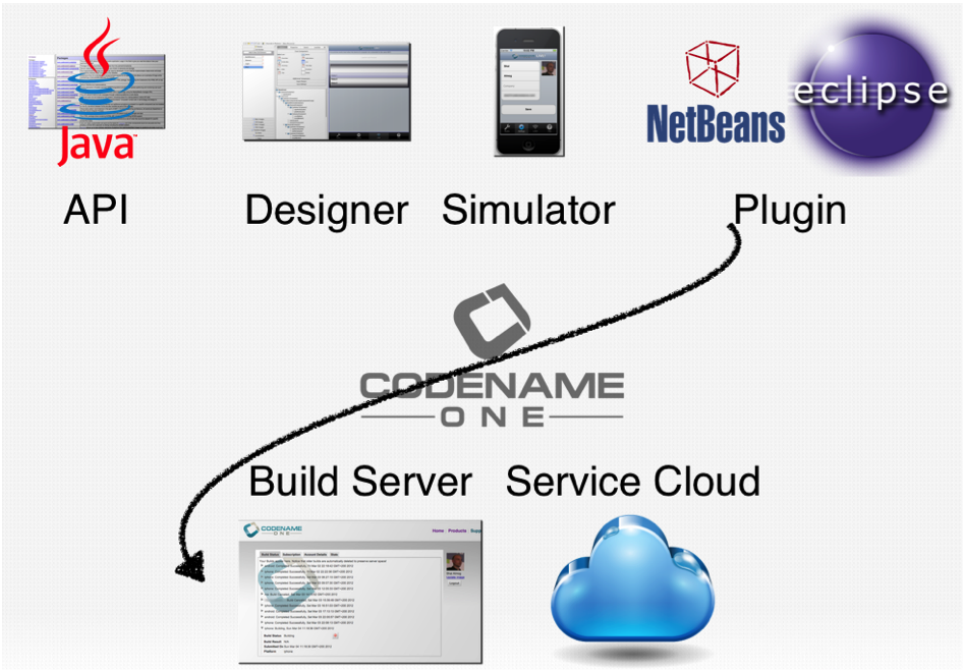
3.1. CodenameOne

CodenameOne je biblioteka koja koristi Java kod koji se onda kompajlira i pakuje za razne mobilne uređaje. Postoje dodaci za popularna razvojna okruženja Eclipse i NetBeans, koji olakšavaju razvoj cross-platform nativnih aplikacija.

CodenameOne se zasniva na nekoliko koncepata:

- definisanje interfejsa nezavisnih od uređaja i operativnog sistema na kojem se aplikacija izvršava
- definisanje načina pravljenja dodataka (plugin-ova) za razna razvojna okruženja
- postavljanje Build servera koji kompajliraju aplikaciju

Pošto su razvojni alati za moderne mobilne platforme veliki i zahtevni, CodenameOne pruža svoj servis za testiranje i kompajliranje aplikacija, što znači da je moguće kompajlirati na primer za iOS bez alata instaliranih na OS X računaru, što je inače slučaj.

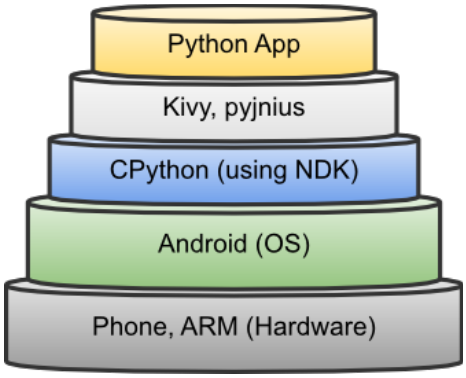


CodenameOne je zreo i relativno popularan projekat. Takođe je potpuno besplatan i open-source. Međutim, pošto se kompajliranje vrši na udaljenim serverima, potrebno je platiti mesečnu pretplatu za njihovo korišćenje. Instalacija lokalnog build servera u trenutku pisanja ovog rada koštala bi preko 20000 dolara. Ovaj podatak, i činjenica da je razvoj CodenameOne projekta veoma usporen i daleko manje aktivan nego pre, odvratilo je autora ovog rada od korišćenja.

3.2. Kivy

Kivy je potpuno besplatni, open-source alat koji omogućava programeru da pravi cross-platform nativne aplikacije u programskom jeziku Python. Kivy se takođe može koristiti i za razvoj Desktop aplikacija što je velika prednost u odnosu na sve do sada navedene biblioteke. Podržava Windows, Linux, OS X, Android i iOS. Apsolutno isti kod se može izvršavati na bilo kojoj od navedenih platformi. Takođe, Kivy ima podršku i za grafički intezivne aplikacije što znači da se uz pomoć ove biblioteke čak mogu praviti i igrice.

Kivy ne koristi nativne grafičke komponente, već definiše svoje. Grafički interfejs je takođe moguće definisati posebnim Kv jezikom. Python kod se prevodi u C kod (CPython) koji se zatim koristi na mobilnim uređajima.



Od svog začeća 2011. godine do danas, Kivy je u veoma aktivnom razvoju i sada je već stabilna i zrela platforma za rad koju koriste mnoge aplikacije kako desktop tako i mobilne. Kivy razvija neprofitna organizacija Kivy Organization koju čine veoma iskusni Python programeri.

Autor ovog rada ima iskustva sa Python programskim jezikom i sa Kivy bibliotekom ali se ipak odlučio na korišćenje druge biblioteke zbog daleko bolje razvijenosti, daleko većih mogućnosti i daleko većih performansi.

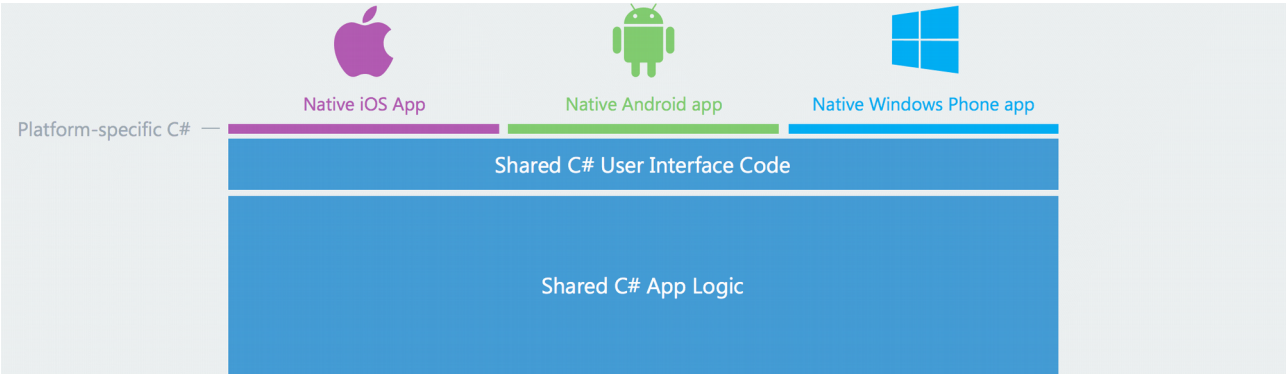
3.3. Xamarin

Xamarin je (od nedavno) besplatna i open-source platforma za razvoj mobilnih aplikacija, vlasništvo kompanije Microsoft. Xamarin koristi nativne komponente mobilnih operativnih sistema, nativne sistemske pozive mapira u svoje interfejse, i postiže izuzetno visoke performanse na svim platformama.

Xamarin takođe podržava integraciju sa bibliotekama za razvoj moblnih igara (Unity).

Xamarin koristi programski jezik C# za kod i markup jezik XAML za definisanje korisničkog interfejsa. Korisnički interfejs je takođe moguće definisati uz pomoć C#.

Xamarin projekte odlikuje izuzetno visok stepen deljenog koda.



Ovo je postignuto korišćenjem veoma velikog niza modernih osobina programskog jezika C# (koji je odabran baš iz tog razloga) kao što su kompleksne strukture, asinhrone funkcije, lambda izrazi, delegati i slično.

Xamarin trenutno podržava Android (mobilni telefoni, tableti, pametni satovi), iOS (telefoni, tableti), Windows (Universal Windows Platform – desktop i mobilne aplikacije) a dostupna razvojna okruženja su Visual Studio (na Windows operativnom sistemu) i Xamarin Studio (na OS X operativnom sistemu). Razvojno okruženje za Linux (MonoDevelop) postoji, ali nije u potpunosti podržano.

Za kompajliranje Xamarin aplikacija za iOS potreban je računar sa OS X operativnim sistemom. Xamarin tim je razvio alate koji olakšavaju proces slanja kompajliranja i simuliranja izvršavanja aplikacije, tako da OS X računar može biti negde daleko na mreži, i programer ne mora da ima fizički pristup istom.

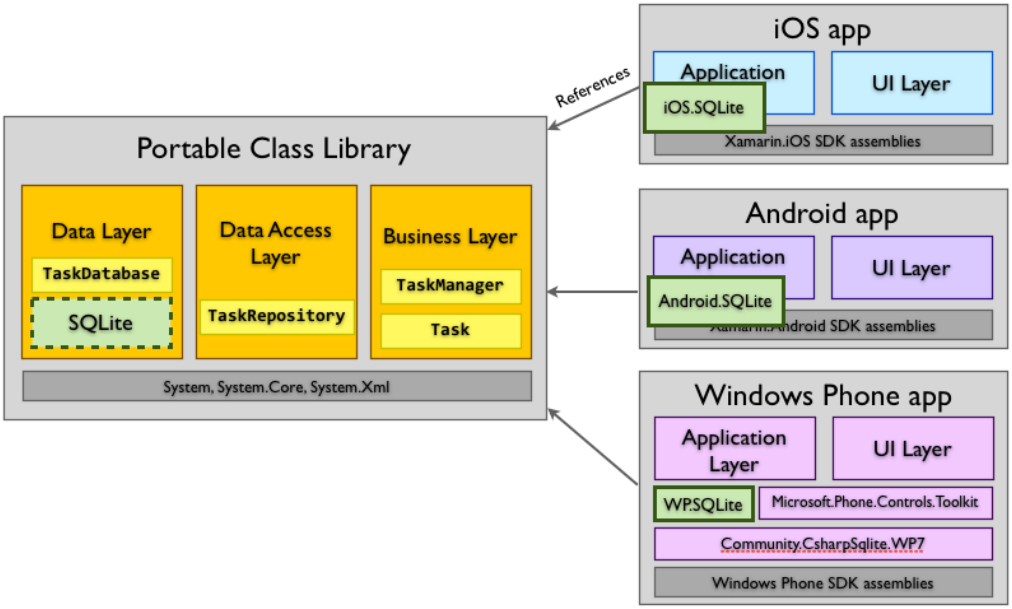
Xamarin.Forms je novi, podrazumevani način razvoja grafičkih okruženja u Xamarin biblioteci. Definišu se XAML fajlovi (XAML je markup jezik sličan XML-u) koji se zatim transformišu u nativne komponente na svim platformama za koje se kompajlira.

Xamarin projekat koristi Mono (open-source implementaciju .NET Framework-a) za izvršavanje C# kod-a na Android i iOS operativnim sistemima. To znači da nema prevođenja koda koji programer direktno piše u različit kod za različite platforme, već da se zaista isti C# kod izvršava na više različitih platformi.

Postoje dva načina postizanja deljivosti koda u definiciji samog projekta:

1. korišćenjem takozvanih Shared projekata
2. i korišćenjem PCL (Portable Class Library) projekata

Način dalje korišćen u ovom radu (i preporučeni način) je korišćenjem PCL projekata prikazanim na sledećoj slici.



Ovakav koncept izrade aplikacija je veoma interesantan. Naime, programer piše klasičnu .NET C# biblioteku koristeći Xamarin interfejse koju zatim drugi projekti (za specifične platforme) koriste. Dakle, programer nema nikakvu potrebu na menja projekte za specifične platforme, postoji samo jednostavna zavisnost između projekta za specifičnu platformu i PCL projekta.

Naravno, postoji mogućnost izmena projekata za različite platforme, čak i predefinisani načini u Xamarin biblioteci uz čiju pomoć je ovaj proces znatno olakšan.

Xamarin biblioteka ima odličnu dokumentaciju i veoma veliku podršku pošto je mnogo programera koristi. Iako je razvojno okruženje glomazno, stabilnost, podrška za statički pisan jezik – C#, performanse i mogućnosti biblioteke bile su presudan faktor u odluci autora ovog rada koju će biblioteku koristiti za razvoj aplikacije opisanu u ovom radu.

Za implementaciju aplikacije koja je prikazana u ovom radu i čija je svrha da prikaže mogućnosti Xamarin biblioteke, kao i da prikaže neke koncepte koje je neophodno koristiti u razvoju cross-platform native mobilnih aplikacija korišćena je Xamarin biblioteka, Xamarin.Forms biblioteka za razvoj grafičkog interfejsa, programski jezik C# 6 i razvojno okruženje Microsoft Visual Studio 2015.

Sav kod ovde prikazane aplikacije dostupan je na http://github.com/nmilosev/PMF

4. Motivacija za ovakav način razvoja mobilnih aplikacija

Važno je napomenuti koje su prednosti razvoja hibridnih tj. cross-platform nativnh mobilnih aplikacija, u poređenju sa razvijanjem posebnih aplikacija za sve platforme u za to predviđenim alatima.

Prva prednost je očigledno deljenje koda. Ukoliko bismo razvijali aplikaciju za tri ovde navedene mobilne platforme na tradicionalan način, tj. posebno za svaku platformu, morali bismo tri puta da pišemo kod za svaku funkcionalnost. Takođe, morali bismo da koristimo tri različita programska jezika, što je druga prednost kod korišćenja biblioteke kao što je Xamarin – koristi se samo jedan programski jezik. Treća prednost je sam rad sa interfejsima aplikacije. Biblioteke kao što su Xamarin pružaju jedinstveni pristup mogućnostima uređaj za koji se razvija tako što definišu jedan interfejs za pristupanje nekoj komponenti (npr. Kompasu) čija implementacija zavisi od platforme za koju se kompajlira aplikacija. Ukoliko bismo se odlučili da našu aplikaciju razvijamo na tradicionalan način – morali bismo da se upoznamo sa definicijom biblioteke za kompas (u ovom slučaju) za sve tri različite mobilne platforme – što nam oduzima dodatno vreme. Prednosti ima još, ovo su samo neke – najvažnije.

S druge strane, ovakav pristup razvoju ima i mana:

- razvojno okruženje je veoma glomazno pošto mora da ima mogućnosti kompajliranja za mnogo različitih platformi
- interfejsi koje definišu biblioteke za cross-platform razvoj često kasne za nativnim bibliotekama mobilnih operativnih sistema, što znači da nekada može proći dosta vremena od uvođenja nove opcije operativnog sistema do mogućnosti upotrebe te opcije u samoj biblioteci
- gubi se potpuna kontrola nad kompajliranim kodom što može dovesti do gubitka performansi
- pošto se biblioteka mora spakovati u paket mobilne aplikacije, često taj paket bude dosta veći u poređenju sa istom aplikacijom napisanom na tradicionalan način, što može odvratiti korisnike od preuzimanja
- vreme kompajliranja je često duže
- iako je nivo apstrakcije veoma veliki često je potrebno znanje detalja implementacije specifične platforme za koju se razvija

Ipak, i pored ovih mana, u trenutku pisanja ovog rada dosta biblioteka za cross-platform mobilne aplikacije se aktivno razvija što svedoči da je sve veći broj programera koji se odlučuju na ovakav pristup.

5. Kratka istorija Xamarin biblioteke

Kada je u junu 2000. godine kompanija Microsoft objavila prvu verziju .NET framework-a nekoliko programera je počelo na razvoju implementacije koja bi radila na Linux operativnom sistemu. Projekat je nazvan Mono i zvanično je objavljen 19. jula 2001. godine. Mono projekat je kasnije bio razvijan od strane Novell i Attachmate kao delimično komercijalni alat, koje su 2011. godine napustile projekat.

U maju 2011. godine Miguel de Icaza glavni Mono programer i osnivač projekta je osnovao novu kompaniju Xamarin i objavio da ponovo preuzima održavanje Mono projekta. Takođe je naglašeno da je fokus nove kompanije na mobilnim platformama. U tom trenutku biblioteke za razvoj Android i iOS aplikacija su se zvale MonoAndroid i MonoTouch.

Veliki napredak u razvoju i verziju 2.0 Xamarin objavljuje u februaru 2013. godine. U isto vreme pojavljuje se i razvojno okruženje Xamarin Studio (zasnovano na MonoDevelop razvojnom okruženju) kao i svi potrebni alati za razvoj Android i iOS aplikacija u razvojnom okruženju Microsoft Visual Studio.

U sledećem periodu broj programera koji koriste Xamarin biblioteku znatno raste i biblioteka postaje sve zrelija i moćnija.

U verziji 3 koja je objavljena u maju 2014. godine pojavljuje se i biblioteka Xamarin.Forms za definisanje grafičkog interfejsa aplikacije na jedinstven način za sve platforme.

Jedan od glavnih problema Xamarin biblioteke bila je cena. Cena korišćenja ove biblioteke bila je oko $3000 godišnje po programeru, što mnogi programeri nisu mogli da opravdaju pogotovo što su čak i u tom trenutku postojala alternativna besplatna open-source rešenja.

U februaru 2016. godine u zvaničnom objavljenju navodi se da je kompanija Microsoft novi vlasnik kompanije Xamarin i svih prava na Xamarin biblioteke. Tačan iznos za koji je Microsoft kupio kompaniju Xamarin nije poznat.

Ubrzo nakon toga na Microsoft Build 2016 konferenciji, Microsoft objavljuje da će cela Xamarin biblioteka biti potpuno besplatna za korišćenje i u potpunosti open-source pod MIT licencom. Takođe, Mono (open-source implementacija .NET framework-a) koji je takođe bio vlasništvo kompanije Xamarin, postaje ponovo u potpunosti open-source posle skoro 20 godina.

U trenutku pisanja ovog rada Xamarin je u potpuno besplatan i open-source alat koji se lako može preuzeti sa sajta http://xamarin.com ili uz besplatno razvojno okruženje Microsoft Visual Studio 2015 Community

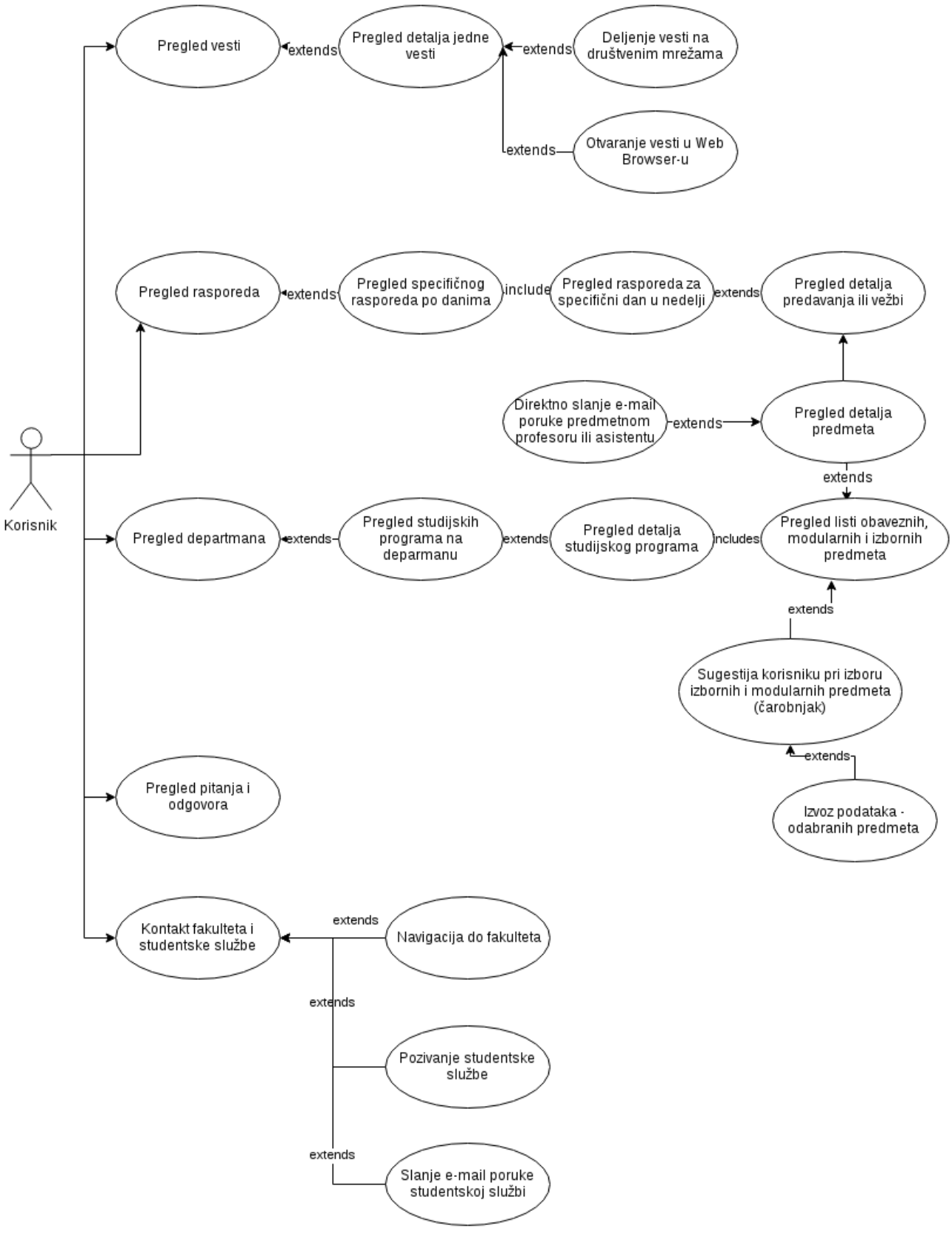
Aplikacije pisane korišćenjem Xamarin biblioteke mogu se bez naknade objavljivati na servisima kao što su Google Play Store, Apple AppStore ili Windows Store.

6. Aplikacija za studentske servise

Cilj ovog rada je da prikaže razvoj jedne moderne cross-platform nativne mobilne aplikacije. Za ovu svrhu osmišljena je mobilna aplikacija za studentske servise Prirodno-matematičkog fakulteta u Novom Sadu. Aplikacija ima sledeće mogućnosti i funkcionalnosti:

- prikaz vesti sa predviđenog web servisa
- prikaz rasporeda časova po departmanima/smerovima/studijskim programima/semestrima – po danima
- detaljni prikaz studijskih programa osnovnih, master i doktorskih studija
- detaljni prikaz detalja studijskih programa i njihovih obaveznih i izbornih predmeta i predmetnih nastavnika
- pomoćnik pri odabiru izbornih i obaveznih predmeta kako bi se uklopili u prethodno određeni broj ESPB (eng. ECST)
- detaljan prikaz detalja i kontaktnih informacija predmetnih nastavnika i asistenata
- prikaz čestih pitanja i odgovora sa predviđenog web servisa
- automatski kontakt studentske službe iz aplikacije sa mogućnosti navigacije do Prirodno-matematičkog fakulteta pomoću eksterne aplikacije (Google Maps na Android platformi, Apple Maps na iOS platformi, Bing Maps na Windows platformi)

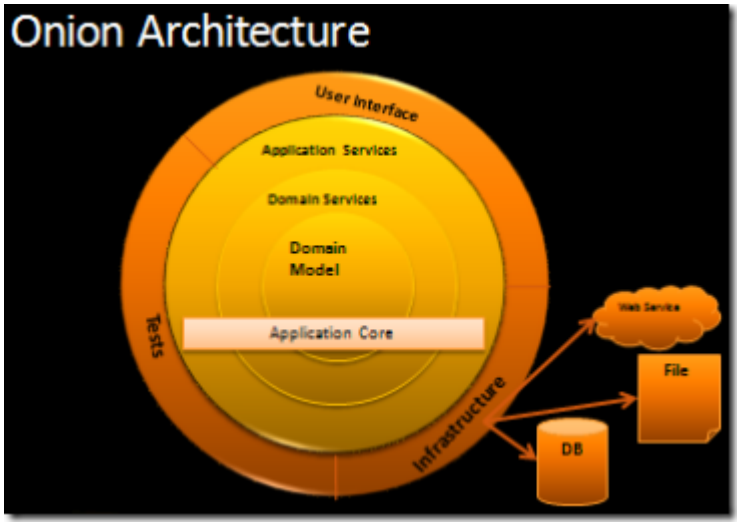
Na sledećem UML dijagramu vide se mogućnosti aplikacije:



6.1. Korišćene tehnologije, biblioteke i dizajn paterni

U ovom poglavlju biće opisane tehnologije, open-source biblioteke i dizajn paterni koji su korišćeni u razvoju aplikacije.

6.1.1. Onion arhitektura



Za razvoj ovde prikazane aplikacije korišćena je Onion arhitektura. Onion arhitektura je relativno nov pristup struktuiranju aplikacije koji je veoma sličan tradicionalnim slojevitim aplikacijama. Ovakav pristup odlikuje visok nivo apstrakcije a glavna razlika u odnosu na tradicionalni slojeviti pristup je što najniži sloj aplikacije nije sloj baze podataka kao što je često slučaju u tradicionalnom slojevitom pristupu, već je osmišljen nov koncept najnižeg sloja koji je nazvan jezgro (eng. Core) aplikacije.

U ovom sloju aplikacije nalaze se klase koje opisuju tipove podataka sa kojima aplikacija radi i svi interfejsi aplikacije, dok je baza podataka u najvišem sloju (spoljašnjem) aplikacije i lako se može zameniti drugom bazom podataka ili drugom vrstom servisa za skladištenje podataka. Klase i interfejsi u ovom sloju su veoma jednostavni, na primer, klasa koja opisuje predmetnog nastavnika ili asistenta:

```
public class Staff
{
    public string Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public string Vocation { get; set; }

    public string URL { get; set; }
    public string Email { get; set; }

    public string ImageURL { get; set; }

    public string FullName => string.Join(" ", Title, FirstName, LastName);

    public List<Subject> Subjects { get; set; }
}
```

U ovom sloju su definisani i interfejsi aplikacije kao što je na primer interfejs za studijske programe:

```
public interface IProgramsSource : InterfaceBase
{
    Task<List<Program>> ForDepartment(int departmentId, string langCode);

    Task<Program> ForId(int programId, string langCode);
}
```

Takođe, svi interfejsi aplikacije koji rade sa podacima nasleđuju interfejs InterfaceBase:

```
public interface InterfaceBase
{
    bool RequireConnection { get; }

    bool IsAvailable { get; }

    bool IsDataValid { get; }
}
```

koji definiše neke zajedničke osobine kao što su da li je potrebna internet konekcija za funkcionisanje, da li je servis dostupan, da li su podaci trenutno u memoriji validni.

Sloj koji je oko sloja jezgra aplikacije je sloj sa servisima koji rade nad tim podacima. Važno je napomenuti da se u ovom sloju ne nalazi poslovna logika aplikacije već samo operacije koje rad nad „sirovim“ objektima jezgra aplikacije

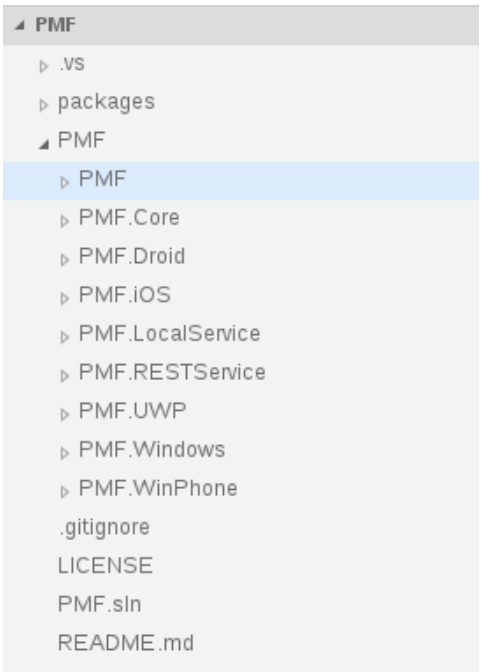
i koje ne znaju ništa o logici aplikacije. Svrha ovog sloja je da omogući programerima još jedan stepen apstrakcije između podataka i sloja poslovne logike.

Sloj koji je oko sloja servisa domena aplikacije je sloj sa poslovnom logikom. U ovom sloju su implementirane operacije koje rade sa podacima kao i sve druge operacije koje nekako imaju veza sa njima ili nekako manipulišu podatke.

U najvišem sloju su moduli za korisnički interfejs, bazu podataka, testove i drugo. Svrha ovakve raspodele je da se ove komponente mogu lako zameniti bez potrebe da se menjaju delovi iz jezgra aplikacije.

6.2. Struktura koda aplikacije

U ovom poglavlju biće opisana struktura projekta aplikacije. Za ovu aplikaciju korišćen je jedan od predviđenih načina razvoja Xamarin aplikacija kao PCL (Portable Class Library). U ovakvom načinu, sav deljeni kod aplikacije se kompajlira u jednoj portabilnoj biblioteci koja se zatim koristi u projektima specifičnim za razne platforme. Ovaj veoma interesantan pristup pisanju aplikacije kao biblioteke znači da programer sav kod piše kao jednu biblioteku koju zatim jednostavno prosleđuje različitim aplikacijama za različite platforme. U tim aplikacijama često je samo jedna linija koda koja učitava biblioteku i pokreće njen predviđeni glavni metod. U skladu sa ovim, i Onion arhitekturom, aplikacija je podeljena na 9 odvojenih projekata:

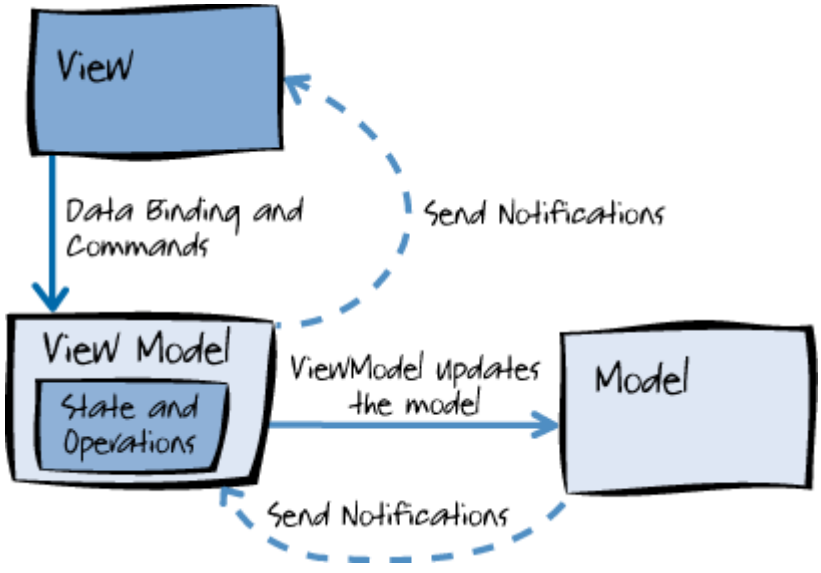


- **PMF.Core** projekat predstavlja jezgro aplikacije i kompajlira se kao portabilna biblioteka. U ovom projektu se nalaze modeli iz domena aplikacija i svi interfejsi koji se koriste kroz aplikaciju

- **PMF.LocalService** je takođe portabilna biblioteka u kojoj se nalaze lokalne implemetacije web servisa koje služe za testiranje aplikacije na test podacima. U ovom projektu se nalaze servisi za vesti, studijske programe, predmete i drugi a implementirani su pomoću asinhronih operacija koje mogu da simuliraju zastoje na mreži i gubitak mreže u potpunosti.
- **PMF.RESTService** je takođe implementacija interfejsa servisa za pristup podataka koji koristi prave RESTful web servise za pristup bazi podataka.
- **PMF.Droid, PMF.iOS, PMF.UWP, PMF.WinPhone, PMF.Windows** su specifični projekti za različite platforme. U ovim projektima se nalaze aplikacije koje se kompajliraju za različite platforme. Uglavnom sadrže resurse koji moraju biti specifično napravljeni za specifične platforme (npr. Slike) i samo jednu glavnu klasu aplikacije koja učitava glavnu portabilnu biblioteku:
- **PMF –** glavna bilbioteka aplikacije u kojoj se nalazi sva poslovna logika i logika korisničkog interfejsa koju dele sve aplikacije za različite platforme.

6.3. MVVM patern

Za implementaciju glavne PMF biblioteke koju koriste sve druge platforme korišćen je MVVM patern.

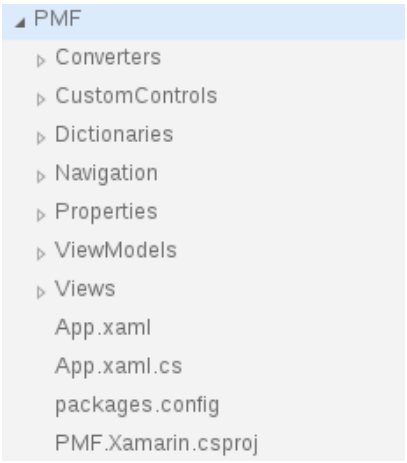


MVVM patern je veoma sličan MVC paternu sa glavnom razlikom da uvodi koncep Data Binding-a. MVVM aplikacije pružaju dobru podelu između različitih slojeva aplikacije i drže se principa da jedan deo aplikacije treba da radi samo jednu stvar i da je radi nezavisno od drugih delova. MVVM aplikacije se sastoje od tri dela:

- Model – sve klase domena

- View – klase korisničkog interfejsa, često pisane u markup jeziku (uglavnom XAML) koji podržava sintaksu za lak Data Binding
- ViewModel – klase koje služe za povezivanje Model i View klasa. Česta implementacija je da jedna View klasa ima tačno jednu ViewModel klasu i prikazuje podatke koje ta ViewModel klasa sadrži. ViewModel klase takođe sadrže detalje kao što su navigacija između drugih View klasa i slično. ViewModel klase podatke čuvaju u C# property-ima koji imaju mogućnost da obaveste View klasu ukoliko dođe do izmene njihove vrednosti kao i da automatski obrade promene vrednosti na View klasi. Ovo se postiže implementacijom .NET INotifyPropertyChanged interfejsa.

Za primenu MVVM paterna uglavnom se koristi već postojeća biblioteka. Za ovaj projekat odabrana je MVVMLight biblioteka zbog jednostavnosti upotrebe i kompatibilnosti sa Xamarin projektima. Takođe, biblioteka je besplatna i open-source. U skladu sa MVVM paternom, struktura projekta glavne PMF biblioteke je sledeća:



Očigledno, ne postoji deo sa domenskim klasama pošto se one koriste iz PMF.Core biblioteke. Svi elementi korisničkog interfejsa se nalaze u Views paketu (namespace) i svi su implementirani u XAML markup jeziku koji će detaljniji biti opisan kasnije.

Za razmenu podataka između View i ViewModel klasa koristi se Data Binding. Data Binding je koncept vezivanja elemenata korisničkog interfejsa direktno sa property-ima C# klase. Na primer u delu aplikacije sa često postavljenim pitanjima i odgovorima u ViewModel klasi postoji kolekcija objekata koji sadrže pitanja i odgovre i koja se zove FAQ:

```
private ObservableCollection<QA> _faqItems;
public ObservableCollection<QA> FAQ
{
    get
    {
        if (_faqItems == null)
            Refresh();
        return _faqItems;
    }
    set
    {
        _faqItems = value;
        RaisePropertyChanged();
    }
}
```

Ovaj property koristi XAML klasa FAQPage.xaml kako bi prikazala ove objekte (samo deo prikazan):

```
<ListView VerticalOptions="FillAndExpand" SeparatorVisibility="None" ItemsSource="{Binding FAQ}"
  <ListView.ItemTemplate>
    <DataTemplate>
      <ViewCell>
        <StackLayout Padding="10" VerticalOptions="FillAndExpand" HorizontalOptions="FillAndExpa
          <Grid Padding="2" RowSpacing="2" ColumnSpacing="2" VerticalOptions="FillAndExpand" Hor
            <Grid.BackgroundColor>
              <OnPlatform x:TypeArguments="Color" Android="{x:Static i18n:Colors.AndroidDarkGray
            </Grid.BackgroundColor>
            <Grid.GestureRecognizers>
              <TapGestureRecognizer />
            </Grid.GestureRecognizers>
```

Važno je napomenuti da se na ovakav način ne mešaju detalji implementacije korisničkog interfejsa (koji se nalaze u XAML kodu) i same logike aplikacije koja se nalazi u ViewModel klasi.

Prednosti korišćenja MVVM paterna u dizajnu aplikacija sa bogatim korisničkim interfejsom su mnoge:

- lakše odvojeno testiranje komonenti – testiranje XAML stranica bez pravih podataka kao i testiranje ViewModel klasa bez korisničkog interfejsa
- potpuno odvajanje zaduženja klasa
- lak redizajn interfejsa za različite platforme, ukoliko je potrebno
- lakše održavanje aplikacije zbog dobro struktuiranog koda
- mogućnost podele programerskih timova na front-end i back-end razvoj

ali postoje i mane:

- neisplativo za male aplikacije, pošto se piše dosta dodatnog koda
- sporiji razvoj

Ipak, za ozbiljnije i kompleksnije aplikacije, MVVM je standardni pristup struktuiranju i razvoju aplikacija.

Takođe neke delove aplikacije bi bilo nemoguće implementirati pomoću XAML i Data Binding mogućnosti jer Xamarin biblioteka ne predviđa da neke komponente budu korišćene na takav način. Zbog toga je bilo potrebno implementirati dodatne komponente koje se zasnivaju na Xamarin komponentama – biće prikazana jedna ovakva komponenta.

6.4. Inversion of Control (IoC)

Kada se implementira cross-platform nativna mobilna aplikacija, veoma je važno da komponente budu slabo povezane i što apstraktnije. Zbog ovoga je veoma interesantan pristup inverzije kontrole (Inversion of Control) i Dependency Injection dizajn patern.

Koncept je veoma jednostavan. Umesto da imamo objekte koji kreiraju druge objekte i njihove zavisne objekte, dizajniramo klase tako da ne zavise od mesta na kojem su napravljene i same definišu svoje potrebe. Na ovaj način se kontrola kreiranja objekata premešta iz nadobjekata koji su tradicionalno pravili sve objekte aplikacije u same objekte koji se instanciraju. Ovo omogućava čistiji dizajn i mnogo uredniju strukturu samog koda.

Implementacija Dependency Injection dizajn paterna koji je samo jedan od načina za postizanje inverzije kontrole je takođe veoma jednostavna, sve klase po ovom dizajn paternu koriste isključivo interfejse za vezu sa drugim objektima koji se lako mogu zameniti po potrebi.

Za implementaciju Dependency Injection dizajn paterna uglavnom se koristi spoljašnja biblioteka. U ovoj aplikaciji korišćena je SimpleIoc biblioteka kao deo MVVMLight biblioteke.

Ovde je prikazan deo implementacije koji koristi ovu biblioteku.

```
private void SetupIoc()
{
    ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);

    SimpleIoc.Default.Register<Navigation.Navigator>(true);

    SimpleIoc.Default.Register<INewsSource, LocalNewsSource>();
    SimpleIoc.Default.Register<IScheduleSource, LocalScheduleSource>();
    SimpleIoc.Default.Register<ISubjectsSource, LocalSubjectsSource>();

    SimpleIoc.Default.Register<IFAQSource, LocalFAQSource>();

    SimpleIoc.Default.Register<IDepartmentsSource, LocalDepartmentSource>();

    SimpleIoc.Default.Register<IProgramsSource, LocalProgramsSource>();
}
```

U metodu SetupIoc se povezuju interfejsi sa konkretnim implementacijama. Ovo znači da kada bilo koja klasa u aplikaciji zatraži, npr. IFAQSource implementaciju, dobiće instancu LocalFAQSource klase.

```
public FAQViewModel()
{
    _faq = SimpleIoc.Default.GetInstance<IFAQSource>();
}
```

Na primer u konstruktoru FAQViewModel klase nam je potrebna instanca servisa za česta pitanja i odgovore i tu koristimo IoC kontejner da dobijemo konkretnu implementaciju. Na ovaj način nikada sami ne kreiramo objekte, već puštamo kontejner da to uradi za nas i da se stara da uvek dobijamo objekat koji želimo da dobijemo. Zbog jednostavnosti upotrebe SimpleIoc biblioteka i generalno koncept inverzije kontrole je veoma koristan za kreiranje singleton objekata (singleton objekat je objekat koji uvek ima samo jednu instancu), te se u ovoj aplikaciji koristi i za kreiranje ViewModel klasa (samo deo prikazan):

```
public ViewModelLocator()
{
    SimpleIoc.Default.Register<MainViewModel>();
    SimpleIoc.Default.Register<MenuViewModel>();
    SimpleIoc.Default.Register<ContactViewModel>();
    SimpleIoc.Default.Register<NewsViewModel>();
    SimpleIoc.Default.Register<ScheduleViewModel>();
    SimpleIoc.Default.Register<ScheduleDetailsViewModel>();
    SimpleIoc.Default.Register<SubjectViewModel>();
    SimpleIoc.Default.Register<FAQViewModel>();
    SimpleIoc.Default.Register<ProgramsViewModel>();
    SimpleIoc.Default.Register<WizardViewModel>();
}

public MainViewModel Main => ServiceLocator.Current.GetInstance<MainViewModel>();

public ContactViewModel Contact => ServiceLocator.Current.GetInstance<ContactViewModel>();

public MenuViewModel Menu => ServiceLocator.Current.GetInstance<MenuViewModel>();

public NewsViewModel News => ServiceLocator.Current.GetInstance<NewsViewModel>();
```

a takođe i za ViewLocator klasu koja služi za lociranje View objekata:

```
public ViewLocator()
{
    SimpleIoc.Default.Register<MainPage>();
    SimpleIoc.Default.Register<ContactPage>();
    SimpleIoc.Default.Register<AboutPage>();
    SimpleIoc.Default.Register<WelcomePage>();
    SimpleIoc.Default.Register<MenuPage>();

    SimpleIoc.Default.Register<NewsPage>();
    SimpleIoc.Default.Register<NewsArticlePage>();

    SimpleIoc.Default.Register<SchedulePage>();
    SimpleIoc.Default.Register<ScheduleDetailsPage>();

    SimpleIoc.Default.Register<SubjectPage>();

    SimpleIoc.Default.Register<FAQPage>();

    SimpleIoc.Default.Register<DepartmentsPage>();
    SimpleIoc.Default.Register<ProgramsPage>();
    SimpleIoc.Default.Register<ProgramDetailsPage>();

    SimpleIoc.Default.Register<WizardPage>();

    SimpleIoc.Default.Register<StudentServicesPage>();
}
```

```
public WelcomePage WelcomePage => ServiceLocator.Current.GetInstance<WelcomePage>();

public MainPage MainPage => ServiceLocator.Current.GetInstance<MainPage>();
public MenuPage MenuPage => ServiceLocator.Current.GetInstance<MenuPage>();
```

Ovo u mnogome olakšava rad sa kompleksnim View i ViewModel klasama kojih ima mnogo. Takođe, klase ViewModelLocator i ViewLocator su napisane na takav način da se mogu koristiti direktno bilo gde iz koda čak i iz XAML fajlova.

6.5. Resursi aplikacije

Kada se razvija ovakav tip aplikacije, česta je potreba da se definišu globalni resursi aplikacije. Ovo mogu biti razni pomoćni objekti koji se koriste kroz celu aplikaciju. Koncept inverzije kontrole prikazan u prošlom poglavlju nam donekle može pomoći, međutim ostaje problem lociranja samih „lokator“ klasa koje se negde moraju definisati.

Po konvenciji (generalno u .NET aplikacijama koje koriste XAML) ovo je App.xaml fajl.

App.xaml fajl i propratna App.xaml.cs klasa su glavna ulazna tačka aplikacije i mogu sadržati definiciju resursa aplikacije. Pri kompajliranju App.xaml klase, klase koje su navedene kao resursi biće instancirane sa podrazumevanim konstruktorima i smeštene u memoriju. Odatle će im biti veoma lako pristupiti bilo gde iz aplikacije, bilo iz C# klasa ili XAML klasa.

Sadržaj App.xaml fajla:

```
<?xml version="1.0" encoding="utf-8" ?>
<Application xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="PMF.App"
             xmlns:views="clr-namespace: PMF.Views"
             xmlns:vm="clr-namespace: PMF.ViewModels"
             xmlns:conv="clr-namespace: PMF.Converters">

  <Application.Resources>
    <ResourceDictionary>
      <views:ViewLocator x:Key="ViewLocator"/>
      <vm:ViewModelLocator x:Key="ViewModelLocator" />
      <conv:IntToDayNameConverter x:Key="dayNameConverter"/>
      <conv:ScheduleItemTypeToTypeNameConverter x:Key="scheduleItemTypeConverter"/>
    </ResourceDictionary>
  </Application.Resources>

</Application>
```

Resursi se definišu u Application.Resources promenljivi koristeći se posebnim rečnikom sa parovima ključ i vrednost. Naravno ovde se navode samo željeni ključevi, dok će vrednosti biti instance objekata. Ovim instancama objekata kasnije se pristupa na veoma jednostavan način:

```
private Views.ViewLocator _viewLocator = (Application.Current.Resources["ViewLocator"] as Views.ViewLocator);
```

ili iz XAML koda:

```
xmlns:i18n="clr-namespace: PMF.Dictionaries"
BindingContext="{Binding Source={StaticResource ViewModelLocator}, Path=Programs}"
Title="{i18n:Translate PickDepartment}">
```

sa ključnom reči StaticResource i navedenim ključem.

6.6. Navigacija

Kada se razvijaju MVVM aplikacije, bilo mobilne ili desktop, veoma velik problem je navigacija između raznih ekranskih formi. Problem je što po principima MVVM paterna, View klase ne bi trebalo da znaju ništa o postojanju drugih View klasa a ni o ViewModel klasi za koju su vezane. Takođe, česta je diskusija o tome gde bi trebalo smestiti deo koda koji će raditi navigaciju da li u ViewModel klasu ili u View klasu. Implementacija zavisi od programera do programera, često se pojavljuju rešenja gde će cela aplikacija biti implementirana po MVVM principu bez navigacije, koja će biti naknadno dodata kao klasičan event-driven deo aplikacije. Autor ovog rada se odlučio da ne odustane od MVVM principa u ovom slučaju i da implementira (po njegovom mišljenju) dobar način navigacije kroz aplikaciju.

Važno je napomenuti da je aplikacija za studentske servise MasterDetail aplikacija. To znači da postoje dve centralne komponente aplikacije Master strana, koja se nikada ne menja (u ovom slučaju to je meni aplikacije) i Detail strana koja prikazuje sadržaj. Implementacija se lako vidi u fajlu MainPage.xaml:

```
<?xml version="1.0" encoding="utf-8" ?>
<MasterDetailPage xmlns="http://xamarin.com/schemas/2014/forms"
                  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                  x:Class="PMF.Views.MainPage"
                  xmlns:local="clr-namespace:PMF.Views"
                  DetailPage="{Binding WelcomePage, Source={StaticResource ViewLocator}"
                  MasterPage="{Binding MenuPage, Source={StaticResource ViewLocator}">
</MasterDetailPage>
```

Ova implementacija se oslanja na MasterDetailPage klasu iz Xamarin.Forms biblioteke.

Donekle, ovo olakšava navigaciju. Ukoliko imamo referencu na glavnu stranu (koju lako možemo dobiti pomoću ViewLocator klase) možemo da pristupimo DetailPage polju i da ga promenimo instancom željene strane – koju takođe možemo dobiti iz ViewLocator klase. Implementirana je odvojena klasa Navigator čija jedina svrha je da navigira između stranica aplikacije. Na primer, kod za navigaciju sa jedne stranice na drugu:

```
public void Navigate(Type pageType)
{
    var page = SimpleIoc.Default.GetInstance(pageType) as Page;
    _viewLocator.MainPage.Detail = new NavigationPage(page);
}
```

Kao parametar se šalje tip tj. klasa stranice koju želimo da prikažemo korisniku. Ovo je pogotovo korisno jer ne ostavljamo mesta za greške, u poređenju da se na primer ime strane prosledi kao string. Takođe, SimpleIoc biblioteka ima metod koji za prosleđeni tip klase vraća njenu implementaciju, što dovodi do veoma jednostavne implementacije.

Kada neka stranica želi da navigira na drugu stranicu, tj. kada korisnik naznači da želi da ode na neku drugu stranicu na primer klikom na dugme, tada stranica kroz komandu na svom ViewModel-u javlja Navigator klasi da uradi navigaciju. Na primer za otvaranje detalja predmeta u aplikaciji:

```
if (subjectsData.IsDataValid)
{
    SimpleIoc.Default.GetInstance<SubjectViewModel>().Current = s;
    SimpleIoc.Default.GetInstance<Navigator>().NavigateModal(typeof(Views.SubjectPage));
}
else
{
    UserDialogs.Instance.ErrorToast("Error".Localize(), "SubjectLoadError".Localize(), 1500);
}
```

Pre samog poziva za navigaciju, potrebno je i osvežiti odgovarajuću ViewModel klasu kako bi podaci bili ažurni.

6.7. Lokalizacija

Aplikacija za studentske servise je lokalizovana na više jezika: srpski (ćirilica i latinica), engleski, i mađarski. Ovo se postiže ne korišćenjem eksplicitno definisanih string-ova (hard-coded strings) u korisničkom interfejsu već korišćenjem rečnika pojmova za svaki jezik pojedinačno.

Xamarin biblioteka podržava lokalizaciju aplikacija. Koriste se standardni .NET rečnici resursa (.resx) fajlovi koji se zatim kompajliraju u C# klase sa konstantama. Na primer, deo rečnika za engleski jezik u ResX formatu:

```
<data name="DefaultTitle" xml:space="preserve">
  <value>PMF - Student Services</value>
</data>
<data name="Description" xml:space="preserve">
  <value>Application for Student Services</value>
</data>
<data name="DMI" xml:space="preserve">
  <value>Department of Mathematics and Informatics</value>
</data>
<data name="DMILink" xml:space="preserve">
  <value>http://www.dmi.rs</value>
</data>
<data name="Error" xml:space="preserve">
  <value>Error!</value>
</data>
<data name="FAQError" xml:space="preserve">
  <value>Greška prilikom učitavanja pitanja i odgovora. Pokušajte ponovo kasnije...</value>
</data>
<data name="FAQTitle" xml:space="preserve">
  <value>F.A.Q.</value>
</data>
```

i odgovarajuća C# klasa:

```
/// <summary>
///   Looks up a localized string similar to PMF - Student Services.
/// </summary>
internal static string DefaultTitle {
    get {
        return ResourceManager.GetString("DefaultTitle", resourceCulture);
    }
}

/// <summary>
///   Looks up a localized string similar to Application for Student Services.
/// </summary>
internal static string Description {
    get {
        return ResourceManager.GetString("Description", resourceCulture);
    }
}
```

ResX rečnici se mogu definisati u samom XML-u ili kroz vizuelni alat koji se nalazi u razvojnom okruženju. Za ključeve se korišćeni termini na engleskom jeziku pošto se oni zapravo koriste u kodu aplikacije koji je na engleskom jeziku.

Nažalost, osim podrške za kompajliranje ResX rečnika, Xamarin biblioteka nam ne pruža mnogo naprednih mogućnosti za korišćenje istih, te su zbog toga morale biti implementirane dodatne pomoćne klase. Pri implementaciji ovih dodatnih klasa vodilo se računa da budu kratke, brze, i da se mogu ponovo koristiti u budućim projektima.

Autor ovog teksta se ovde odlučio za upotrebu jedne od moćnijih mogućnosti C# programskog jezika – ekstenzione metode. Ekstenzioni metodi (Extension Method) su metodi koji se mogu „prilepiti“ na klasu bez da se ona nasledi i ponovo instancira. Prirodno je bilo napraviti ovakav metod za string klasu koji može da lokalizuje bilo koji ključ, i koji za na koji jezik treba da ga prevede:

```
public static string _translate(string key)
{
    if (string.IsNullOrEmpty(key))
        return string.Empty;

    var translation = ResourceManager.GetString(key, CultureInfo);

    if (translation != null)
        return translation;
    else
        return ERROR;
}

public static string Localize(this string key)
{
    return _translate(key);
}
```

Dakle, sve što je potrebno da bi se lokalizovala vrednost je da se pozove Localize metod nad nekom string promenljivom. Na primer:

```
if (departments.IsDataValid)
    SimpleIoc.Default.GetInstance<Navigator>().Navigate(typeof(Views.DepartmentsPage));
else
    UserDialogs.Instance.ErrorToast("Error".Localize(), "ProgramsError".Localize(), 1500);
```

Ovaj metod se može pozvati i na samoj definiciji string-a, kao što je ovde i prikazano.

Translator klasa se stara da lokalizacija bude na predviđenom jeziku, i da program nastavi svoje izvršavanje čak i ako ne postoji prevedena vrednost ključa.

Naravno, većina elemenata koje je bilo potrebno prevoditi na razne jezike se nije nalazio u C# kodu već u XAML kodu gde je i definisan korisnički interfejs.

Za korišćenje Translator klase iz XAML koda bilo je takođe potrebno implementirati i ekstenziju samog XAML jezika za markiranje. Naravno, moguća je i jednostavnija implementacija, ali ovakav način je ispravan sa aspekta čistoće koda i daleko najfleksibilnija opcija.

Ekstenzije se u XAML kodu definišu implementacijom specijalnog interfejsa IMarkupExtension:

```
[ContentProperty("Value")]
public class Translate : IMarkupExtension
{
    public string Value { get; set; }

    public object ProvideValue(IServiceProvider serviceProvider)
    {
        return Value.Localize();
    }
}
```

Takođe je potrebno naglasiti da ova ekstenzija radi sa vrednostima, a ne sa samim delovima XAML koda. Kada je ovakva ekstenzija napravljena, veoma je lako koristiti naš Translator.Localize metod bilo gde iz XAML koda. Prvo se mora definisati upotreba same ekstenzije, a za to je dovoljno samo da kažemo u kojem se paketu (namespace) nalazi. Ovo se najčešće radi u prvom elementu stranice, jer onda svi podelementi nasleđuju tu definiciju:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:i18n="clr-namespace: PMF.Dictionaries"
             BindingContext="{Binding Source={StaticResource ViewModelLocator}, Path=Menu}"
             x:Class="PMF.Views.MenuPage"
             Icon="menu.png"
             Title="Master">
```

Napomena: svuda je korišćeno ime „i18n“ kao univerzalna programerska oznaka za internacionalizaciju koda tj. lokalizaciju (18 je broj slova u engleskoj reči internationalization između prvog „i“ i poslednjeg „n“)

Zatim se ekstenzija može koristiti bilo gde u kodu korišćenjem i18n:Translate direktive:

```
<StackLayout Orientation="Horizontal">
  <StackLayout.GestureRecognizers>
    <TapGestureRecognizer Command="{Binding WelcomeCommand}"/>
  </StackLayout.GestureRecognizers>
  <StackLayout.Margin>
    <OnPlatform x:TypeArguments="Thickness"  WinPhone="5,20,0,0" Android=
  </StackLayout.Margin>
  <Image Source="home.png" WidthRequest="36" HeightRequest="36" />
  <Label FontSize="24" Text="{i18n:Translate Title}" Margin="5,0,0,0" />
</StackLayout>
<Label Margin="5,0,0,0" FontSize="14" Text="{i18n:Translate PMFName}"/>
<Label Margin="5,0,0,0" FontSize="14" Text="{i18n:Translate UNS}"/>
<Label Margin="5,0,0,0" FontSize="12" Text="{i18n:Translate Version}"/>
```

6.8. Ekstenzije XAML komponenti

Pored ove ekstenzije za lokalizaciju, u razvoju aplikacije bilo je potrebno implementirati dodatne XAML komponente koje se oslanjaju na Xamarin-ove implementacije.

Razlog za ovo je što je Xamarin.Forms biblioteka još uvek u razvoju i neke funkcionalnosti nisu moguće ukoliko se koristi XAML kod za definiciju interfejsa. U ovakvim slučajevima postoje samo dva rešenja: implementacija celog korisničkog interfejsa u C# kodu, ili definicija dodatnih komponenti.

Ovakvi problemi se uglavnom odnose na nemogućnost korišćenja Data Binding-a (pa ni MVVM paterna) na neke promenljive već postojećih komponenti, što veoma narušava strukturu i dizajn koda aplikacije.

6.8.1. Implementacija PinMap komponente

U Xamarin biblioteci postoji Map komponenta koja predstavlja geografsku mapu. Velika prednost ove komponente je što radi na svim mobilnim uređajima nezavisno od platforme, tj. povezuje se na odgovrajućeg provider-a navigacionih usluga: Google Maps za Android, Apple Maps za iOS, Bing Maps za Windows.

Ova komponenta je jedna od najrazvijanijih komponenti Xamarin.Forms biblioteke, ali i pored toga ne postoji mogućnost postavljanja takozvanog Pin-a (grafičkog obeležja lokacije na mapi) kroz XAML kod. Zbog toga, ovde je implementirana komponenta PinMap koja se oslanja na komponentu Map iz Xamarin.Forms biblioteke ali ima mogućnost definisanja geografske širine i dužine na kojoj će biti postavljen Pin odmah pri konstrukciji objekta. Takođe, po postavljanju Pin-a PinMap-a će automatski prikazati lokaciju tog Pin-a tačno na sredini mape uz odgovarajuću animaciju – što takođe nije moguće iz XAML koda sa običnom Map komponentom.

```
public class PinMap : Map
{
    public double Latitude { get; set; }
    public double Longitude { get; set; }

    public double PinDistance { get; set; }
    public string PinTitle { get; set; }

    public bool NavigateImmediately
    {…
    }

    private void NavigateNow()
    {
        var location = new Position(Latitude, Longitude);

        Pins.Add(new Pin() {
            Position = location,
            Label = PinTitle
        });

        MoveToRegion(MapSpan.FromCenterAndRadius(location, Distance.FromMeters(PinDistance)));
    }
}
```

Takođe, moguće je definisati i šta će pisati na samom Pin-u i sa koje udaljenosti (u metrima) će biti prikazan. U setter-u polja NavigateImmediately se nalazi poziv funkciji NavigateNow koja će postaviti Pin na mapu i prikazati ga sa određene udaljenosti. Ovo je veoma jednostavan način za pozivanje metoda direktno iz XAML koda – svaka promena vrednosti će okinuti setter prekidač koji će tada pozvati navedeni metod.

Ovako napravljene XAML komponente koriste se slično kao i ekstenzije koje su bile prikazane u prošlom poglavlju. Prvo se definiše u kom su paketu (maps):

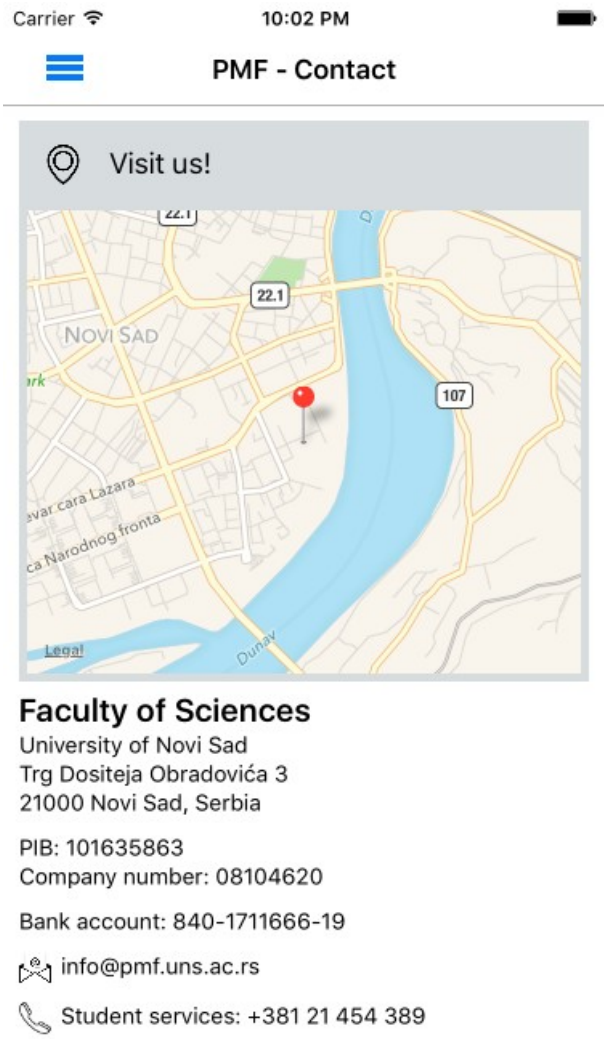
```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
xmlns:maps="clr-namespace:PMF.CustomControls"
BindingContext="{Binding Source={StaticResource ViewModelLocator}, Path=Contact}"
Title="{i18n:Translate ContactTitle}">
```

A zatim se normalno koriste sa prefiksom:

```
    <StackLayout Orientation="Horizontal" Margin="5,5,0,0">
      <Image Source="placeholder.png" HeightRequest="25" Margin="10"/>
      <Label Text="{i18n:Translate ContactVisitUs}" VerticalOptions="Center" FontSize="18"/>
    </StackLayout>
    <maps:PinMap MapType="Street" Latitude="45.245411" Longitude="19.852777"
         PinTitle="{i18n:Translate PMFName}" PinDistance="1500"
         NavigateImmediately="True" Margin="5,0,5,5"/>
  </StackLayout>
  <Label Text="{i18n:Translate PMF}" FontSize="20" FontAttributes="Bold" Margin="10,0,0,0"/>
  <Label Text="{i18n:Translate UNS}" FontSize="14" Margin="10,-5,0,0"/>
```

U ovakvoj definiciji, polja Latitude, Longitude, PinTitle, PinDistance, NavigateImmediately su sva iz PinMap klase koja je naknadno definisana. Na ovaj način ne gubimo na urednosti koda, a ni na kompatibilnosti sa različitim platformama, pošto je ovo i dalje u osnovi Xamarin.Forms komponenta koja može da se koristi na svim platformama.

Ovde je prikazana komponenta na iOS platformi:



6.8.2. Implementacija ExtendedPicker komponente

Još jedna komponenta koja je nedostajala je Picker (slično ComboBox komponenti u drugim bibliotekama) koja ima mogućnost Data Binding-a.

Većina kompozitnih komponenti iz Xamarin.Forms biblioteke kao što su ListView ili TableView definišu polje ItemsSource koje se može manipulisati uz pomoć DataBinding-a. Međutim, iz nekog razloga ovo nije slučaj sa Picker komponentom, te je bila potrebna dodatna implementacija.

```
namespace PMF.CustomControls
{
    public class ExtendedPicker : Picker
    {
        public static readonly BindableProperty ItemsSourceProperty =
                BindableProperty.Create(propertyName: "ItemsSource", declaringType: typeof(List<string>),
                returnType: typeof(List<string>), propertyChanged: (bindable, oldValue, newValue) =>
                {
                    var picker = (ExtendedPicker) bindable;
                    picker.Items.Clear();
                    foreach (var i in (List<string>) newValue)
                        picker.Items.Add(i);
                });

        public List<string> ItemsSource
        {
            get { return (List<string>)GetValue(ItemsSourceProperty); }
            set { SetValue(ItemsSourceProperty, value); }
        }

    }
}
```

Implementacija je malo komplikovanija od implementacije PinMap komponente zbog neophodnosti da se koristi koncept BindableProperty. Kao što mu ime kaže BindableProperty je polje klase koje podržava Data Binding. Ovakav kod uglavnom ne piše programer već se on sam generiše, međutim u ovom slučaju ovo je bilo jedino rešenje. Dakle, postoji obična lista string objekata koje želimo da povežemo sa Picker objektom. Definičemo BindableProperty uz pomoć .NET metoda BindableProperty.Create i dajemo mu ime, tip, i u specijalnom lambda izrazu definišemo šta se dešava kada se vrednost ove promeljive promeni – u našem slučaju jednostavno u instanci same Picker komponente očistimo postojeću listu i dodamo sve elemente nove liste.

Ovakav kod je generički, i nije podležan promenama a u mnogome olakšava dalji rad sa Picker komponentom kada nam je potrebna mogućnost Data Binding-a na kolekciju objekata koju prikazuje:

```
<Label Grid.Row="2" Text="{i18n:Translate Module}" VerticalOptions="Center" FontSize="14"/>
<custom:ExtendedPicker HorizontalOptions="FillAndExpand" Grid.Row="3"
ItemsSource="{Binding Source={StaticResource ViewModelLocator}, Path=Programs.CurrentModuleNames}"
SelectedIndex="{Binding Source={StaticResource ViewModelLocator}, Path=Programs.CurrentModuleId, Mode=TwoWay}"/>

<Label Grid.Row="4" Text="{i18n:Translate SemesterCap}" VerticalOptions="Center" FontSize="14"/>
<custom:ExtendedPicker HorizontalOptions="FillAndExpand" Grid.Row="5"
ItemsSource="{Binding Source={StaticResource ViewModelLocator}, Path=Programs.CurrentSemesters}"
SelectedIndex="{Binding Source={StaticResource ViewModelLocator}, Path=Programs.CurrentSemesterId, Mode=TwoWay}"/>
```

Ovakve mogućnosti „dodavanja“ komponenti na samu biblioteku uz pomoć naprednih koncepata koje pruža programski jezik C# umnogome olakšavaju razvoj ovakvog tipa aplikacija.

6.8.3. XAML konverteri

Prilikom lokalizacije aplikacije, došlo je do problema lokalizacije vrednosti tipa enum. Pošto se ovakvi tipovi podataka u C# programskom jeziku mapiraju na vrednosti tipa int, nije postojao jednostavan način za lokalizaciju samih imena enum vrednosti. Zbog ovoga, korišćeni su XAML konverteri.

XAML konverter je jednostavna klasa koja implementira interfejs IValueConverter i implementira dva njegova metoda: Convert i ConvertBack. Na primer, XAML konverter koji lokalizuje tipove predavanja (predavanje, vežbe ili vežbe na računaru):

```
class ScheduleItemTypeToTypeNameConverter : IValueConverter
{
    Dictionary<int, string> Dictionary = new Dictionary<int, string>()
    {
        { (int)ScheduleItemType.Lecture, "Lecture".Localize() },
        { (int)ScheduleItemType.Lab, "Lab".Localize() },
        { (int)ScheduleItemType.Pratice, "Practice".Localize() }
    };


    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return Dictionary[(int)value];
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

U ovom slučaju nam nije potrebna implementacija prevođenja unazad, te taj metod nije ni implementiran. Lokalizovane vrednosti se čuvaju u rečniku (ključ-vrednost parovi) a sam XAML kod poziva Convert metod za nas:

```
<ContentView Grid.Row="0" Grid.Column="2" Padding="0,15,0,0" Margin="-2,0,0,0">
  <ContentView.BackgroundColor>
    <OnPlatform x:TypeArguments="Color" Android="{x:Static i18n:Colors.AndroidLightGray}" iOS="{x
  </ContentView.BackgroundColor>
  <Label Text="{Binding Type, Converter={StaticResource scheduleItemTypeConverter}"
  HorizontalOptions="CenterAndExpand" VerticalOptions="Start"/>
</ContentView>
```

XAML konverteri su definisani i instancirani u App.xaml klasi. Još jedan interesantan konverter je za dane u nedelji i njihova lokalizovana imena:

```
class IntToDayNameConverter : IValueConverter
{
    Dictionary<int, string> Dictionary = new Dictionary<int, string>()
    {
        { (int)DayOfWeek.Monday, "Monday".Localize() },
        { (int)DayOfWeek.Tuesday, "Tuesday".Localize() },
        { (int)DayOfWeek.Wednesday, "Wednesday".Localize() },
        { (int)DayOfWeek.Thursday, "Thursday".Localize() },
        { (int)DayOfWeek.Friday, "Friday".Localize() },
        { (int)DayOfWeek.Saturday, "Saturday".Localize() },
        { (int)DayOfWeek.Sunday, "Sunday".Localize() },
    };


    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return Dictionary[(int) value];
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

XAML konverteri su moćno oruđe za konverziju podataka koji nam ponovo pomažu da što više koda generalizujemo i izmestimo van korisničkog interfejsa.

6.9. Kod za specifične platfome

Iako je većina koda u PCL projektu nezavisna od platforme na kojoj se izvršava, Xamarin biblioteka nam pruža neke mogućnosti za provere na kojoj se platformi aplikacija trenutno izvršava, ukoliko nam je takva informacija potrebna. Ovo je veoma korisno u dizajno korisničkog interfejsa, ukoliko želimo da izgleda nešto drugačije na različitim platformama kako bi se uklopio u ostatak izgleda operativnog sistema. Na primer u XAML kodu možemo da definišemo različite boje komponenti na različitim platformama ili čak različite numeričke vrednosti za razne osobine – u ovom primeru visinu komponente:

```
<StackLayout Orientation="Vertical">
  <StackLayout Margin="10,10,10,0" Orientation="Vertical" VerticalOptions="Start">
    <StackLayout.BackgroundColor>
      <OnPlatform x:TypeArguments="Color" Android="{x:Static i18n:Colors.AndroidMediumGray}"
      iOS="{x:Static i18n:Colors.iPhoneMediumGray}"/>
    </StackLayout.BackgroundColor>
    <StackLayout.HeightRequest>
      <OnPlatform x:TypeArguments="sys:Double" Android="250" iOS="350"/>
    </StackLayout.HeightRequest>
```

Kod koji će se izvršavati samo na određenim platformama moguće je pisati i u samim C# klasama na sledeći način:

```
private void BuildMessage(ActionSheetConfig cfg, ScheduleItem item)
{
    var converter = Application.Current.Resources["scheduleItemTypeConverter"] as Convert

    if (Device.OS == TargetPlatform.iOS)
    {
        //iOS does not support multi line message
        cfg.Add(item.SubjectTitle)
            .Add(converter.Convert(item.Type, typeof(string), null, CultureInfo.CurrentCu
            .Add(item.TeacherNamesFormatted)
            .Add(item.TimeFormatted)
            .Add(item.Location);
    }
    else
    {
        var message = item.SubjectTitle + Environment.NewLine +
                  converter.Convert(item.Type, typeof(string), null, CultureInfo.CurrentC
                  item.TeacherNamesFormatted + Environment.NewLine +
                  item.TimeFormatted + Environment.NewLine +
                  item.Location;
        cfg.Add(message);
    }

}
```

U ovom primeru za iOS platformu je potrebna drugačija implementacija prikaza poruke, jer ta platforma ne podržava dijaloge koji imaju više od jednog reda.

Iako se ovakve konstrukcije retko koriste, u većini slučajeva kada su potrebne, zapravo su nezamenljive i neki delovi koda se moraju implementirati na ovaj način.

6.10. Pristup naprednijim mogućnostima mobilnih uređaja na jedinstven način

Iako je Xamarin biblioteka u potpunosti cross-platform, u nekim slučajevima mora se pisati kod specifičan za platformu na kojoj se aplikacija izvršava, kao što smo videli u prethodnom poglavlju.

Napredne mogućnosti mobilnih uređaja kao što su kamera, notifikacije, pozivi i poruke i slično se veoma razlikuju od platforme do platforme i još ne postoji jedinstveni način za pristup istima.

Trenutno postoji mnogo open-source biblioteka koje rade upravo ovo, i zbog njih ne moramo da pišemo kod za specifične platforme, jer se takav kod već nalazi u njima. U razvoju aplikacije za studentske servise korišćene su mnoge ovakve biblioteke.

Važno je napomenuti da je korišćenje ovakvih biblioteka moguće samo u projektima koji koriste PCL biblioteku kao osnovu ali ne i u takozvanim Shared projektima, te je ovo još jedan od razloga zbog kojih je bolje krenuti u razvoj uz pomoć PCL biblioteke.

6.10.1. Notifikacije

Na žalost nije moguće na jedinstven način kontrolisati notifikacije tj. Kratke poruke prikazane korisniku na svim različitim platformama.

Zbog ovoga je korišćena odlična open-source biblioteka Acr.UserDialogs. Ova biblioteka je veoma jednostavna za korišćenje, koristi se jedan singleton objekat:

```
UserDialogs.Instance.ErrorToast("Error".Localize(), "NewsError".Localize(), 1500);
```

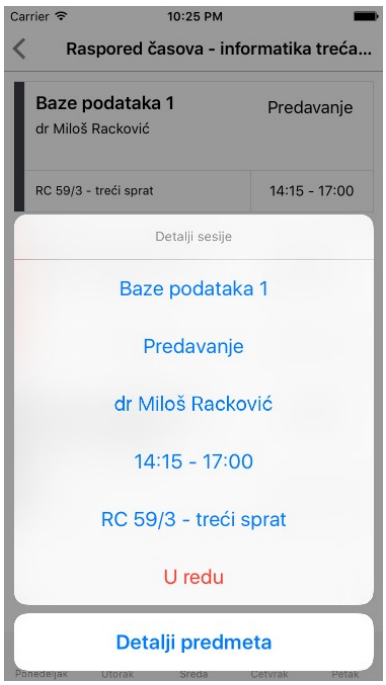
Poziv funkcije prima tri parametra: sadržaj, naslov i dužinu trajanja poruke u milisekundama. Takođe osim ovde prikazane Error poruke, postoje i druge razne kao što su Success ili Info poruke koje se razlikuju po izgledu.

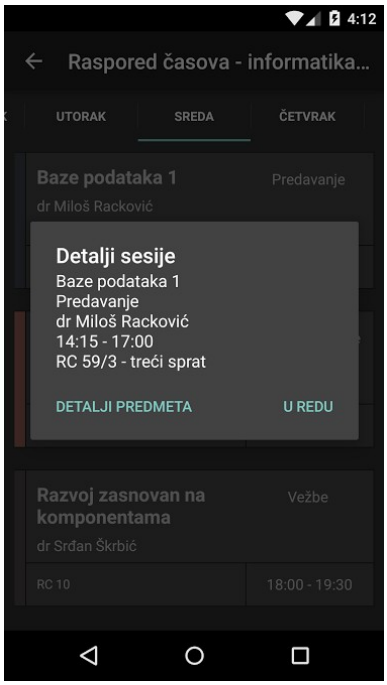
6.10.2. Poruke i dijalozi

Za prikaz drugih, komplikovanijih poruka i dijaloga može se koristiti ista biblioteka. Dijalozi se prave takođe na jednostavan način:

```
private void OpenScheduleItemDetails(ScheduleItem item)
{

    var cfg = new ActionSheetConfig()
            .SetTitle("SessionDetails".Localize())
            .SetDestructive("OK".Localize())
            .SetCancel("SubjectDetails".Localize(), () => ShowSubjectDetails(item.SubjectId));

    BuildMessage(cfg, item);

    UserDialogs.Instance.ActionSheet(cfg);

}
```

i izgledaju kao nativni dijalozi na svim platformama, na primer iOS i Android:



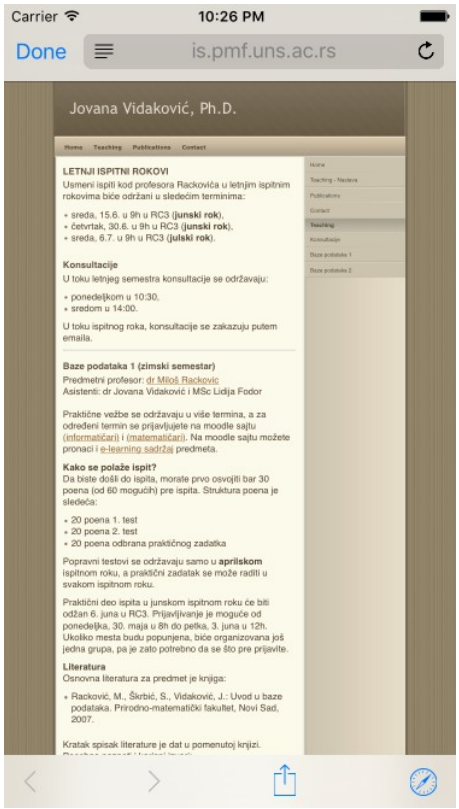


6.10.3. Otvaranje linkova u podrazumevanom internet pregledaču

Još jedna od mogućnosti koja nedostaje u Xamarin biblioteci je otvaranje jednostavnih linkova do web stranica u podrazumevanom web pregledaču koji korisnik podešava. Srećom, postoji open-source biblioteka koja služi samo za ovo (CrossShare):

```
private void Visit(string url)
{
    if (url.StartsWith("http://"))
        Plugin.Share.CrossShare.Current.OpenBrowser(url);
    else
        Plugin.Share.CrossShare.Current.OpenBrowser("http://" + url);
}
```

Jedini nedostatak biblioteke je što ukoliko link ne počinje sa standardnim http:// ili https:// prefiksom već direktno sa adresom (npr. www.google.com) stranica se neće otvoriti, te se taj slučaj mora ručno obraditi. Nakon pozivanja metoda OpenBrowser, korisniku će biti prikazan podrazumevani web pregledač i to u okviru naše aplikacije za studentske servise kao modalni dijalog:



Ovo je veoma važna opcija jer se korisnik lako može vratiti direktno u aplikaciju pritiskom na odgovarajuće dugme.

6.10.4. Deljenje linkova, slanje e-mail poruka

Slanje e-mail poruka preko predefinisane aplikacije za mejlove je takođe nemoguće bez dodatne biblioteke.

Za ovu mogućnost koristi se takođe CrossShare biblioteka:

```
private void SendEmail(string email)
{
    var emailMessenger = Plugin.Messaging.CrossMessaging.Current.EmailMessenger;
    if (emailMessenger.CanSendEmail)
    {
        emailMessenger.SendEmail(email, Current.Title, string.Empty);
    }
}
```

Po pozivu metoda SendEmail koji prima adresu, naslov, i sadržaj poruke kao parametre, korisniku će biti prikazan dijalog da odabere sa koje e-mail adrese želi da pošalje poruku (ukoliko korisnik ima više e-mail adresa podešenih) i zatim popunjena e-mail poruka, spremna za slanje.

Još jedna važna mogućnost modernih mobilnih aplikacija je mogućnost deljenja linkova i sadržaja na raznim društvenim mrežama. Srećom, svi operativni sistemi koje Xamarin biblioteka podržava definišu interfejse koji nam omogućavaju upravo ovo. Nažalost, svaki operativni sistem definiše različit interfejs, te nam je ponovo potrebna biblioteka. Ponovo koristimo CrossShare:

```
private void _saveList(bool toClipboard=false)
{
    var messenger = Plugin.Share.CrossShare.Current;
    var sb = new StringBuilder();
    sb.AppendLine(CurrentProgram.Name);
    sb.AppendLine(CurrentModuleName);
    sb.AppendLine($"{"SemesterCap".Localize()}: {Semester}");
    sb.AppendLine(string.Empty);
    foreach (var group in CurrentSubjects)
        foreach (var checkedSubject in group)
            if (checkedSubject.IsChecked)
            {
                var s = checkedSubject.Subject;
                sb.AppendLine($"{s.Id} - {s.Title} - {"ESPB".Localize()}{s.ESPB}");
            }
    sb.AppendLine(string.Empty);
    sb.AppendLine($"{"ESPB".Localize()}{CurrentESPB}");

    if (!toClipboard)
        messenger.Share(sb.ToString(), "AppName".Localize());
    else
    {
        if (messenger.SupportsClipboard)
        {
            messenger.SetClipboardText(sb.ToString());
            UserDialogs.Instance.ErrorToast("CopiedToClipboard".Localize());
        }
        else
        {
            UserDialogs.Instance.ErrorToast("Error".Localize(), "NotSupported".Localize());
        }
    }
```

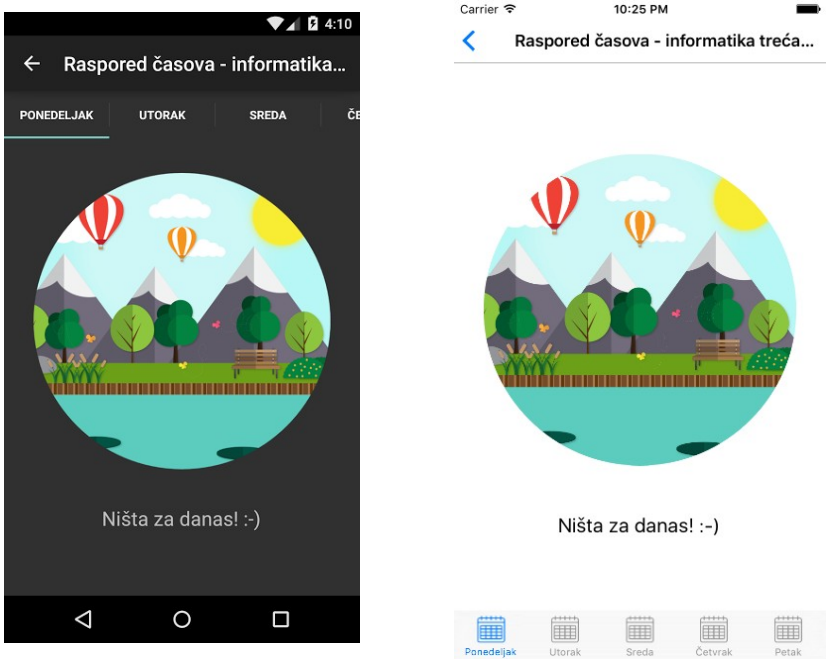
Ovaj deo koda omogućava korisniku da „podeli“ ili snimi lokalno (u aplikacijama koje to podržavaju ili u memoriju – clipboard) listu predmeta koje je odabrao za prijavu semestra. Samo Android platforma podržava snimanje teksta u clipboard, te je ta opcija jedino prikazana na toj platformi.

Kod je veoma jednostavan, koristi se StringBuilder za pravljenje poruke koja se zatim jednostavno prosleđuje metodu Share iz CrossShare biblioteke.

6.10.5. Keširanje slika

Pošto je cilj razvoja cross-platform mobilne aplikacije da što više korisnika ima mogućnost da je koristi, mora se voditi računa i o performansama.

Kada je reč o telefonima starijih generacija ili limitiranih performansi, veoma je korisno koristiti keširanje za velike slike koje se koriste na više mesta kroz aplikaciju. Na primer, u prikazu rasporeda časova koristi se velika slika ukoliko je raspored prazan:



koja se može ponavljati više dana u nedelji. Iako na prvi pogled u ovom slučaju keširanje izgleda nepotrebno, bilo kakvo štеđenje RAM memorije mobilnog uređaja je veoma korisno, i unapređuje performanse ostalih delova aplikacije.

Takođe, u aplikaciji za studentske servise keširaju se i slike koje se preuzimaju preko mreže, što takođe smanjuje prenešene podatke što je veoma važno na mobilnim uređajima.

Za keširanje slika ne postoji ugrađen mehanizam u samoj Xamarin biblioteci, već se koristi eksterna biblioteka FFImageLoading direktno iz XAML koda:

```
<ffimageloading:CachedImage HorizontalOptions="Center" VerticalOptions="Center"
    IsVisible="{Binding HasNoItems}"
    WidthRequest="400" HeightRequest="400"
    DownsampleToViewSize="True"
    Source = "freetime.png" FadeAnimationEnabled="True">
  <ffimageloading:CachedImage.Margin>
    <OnPlatform x:TypeArguments="Thickness" iOS="30,50,30,0" Android ="30,-15,30,-20"/>
  </ffimageloading:CachedImage.Margin>
</ffimageloading:CachedImage>
```

Ova bibloteka koristi ime fajla tj. URL do slike ukoliko se ona preuzima sa mreže kao ključ za keširanje. Takođe se može podesiti u dužina važenja slike u memoriji, animacije i slično.