Usporedba i primjena arhitekturalnih uzoraka dizajna kod razvoja Xamarin mobilnih aplikacija

Smoljan, Antonio

Master's thesis / Diplomski rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:324752

Rights / Prava: Attribution 3.0 Unported/Imenovanje 3.0

Download date / Datum preuzimanja: **2024-04-17**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Antonio Smoljan

USPOREDBA I PRIMJENA ARHITEKTURALNIH UZORAKA DIZAJNA KOD RAZVOJA XAMARIN APLIKACIJA

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Antonio Smoljan

Matični broj: 0016123808

Studij: Informacijsko i programsko inženjerstvo

USPOREDBA I PRIMJENA ARHITEKTURALNIH UZORAKA DIZAJNA KOD RAZVOJA XAMARIN APLIKACIJA

DIPLOMSKI RAD

Mentor :

Izv. prof. dr. sc. Zlatko Stapić

Varaždin, srpanj 2022.

Antonio Smoljan

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Posljednjih godina, na tržištu mobilnih uređaja su se ustalila dva operacijska sustava, Android i iOS. Ta podjela, razvojnim programerima, predstavlja problem jer je potrebno razvijati istu aplikaciju dva puta uz korištenje različitih tehnologija. Taj problem pokušava riješiti tehnologija Xamarin u kojoj je moguće razvijati aplikacije za oba operacijska sustava. U ovome radu se predstavlja Xamarin tehnologija s naglaskom na korištenje različitih arhitekturalnih uzoraka dizajna te isticanje prednosti i nedostataka određenog uzorka. Iako je MVVM arhitektura standardizirana za razvoj Xamarin aplikacija, u teorijskom dijelu rada, kroz jednostavne primjere, su obrađena četiri arhitekturalna uzorka, to su MVVM, MVC, VIPER i Flux. Izrađen je i praktični primjer aplikacije u kojoj se koriste pojedini principi iz arhitektura MVVM, MVC i Flux s ciljem da se spomenute arhitekture usporede u kontekstu razvoja aplikacije većeg obujma.

Ključne riječi: Mobilne aplikacije; Android; iOS; Xamarin; .NET; Arhitekturalni uzorci dizajna; MVVM; MVC; VIPER; Flux

Sadržaj

1. Uvod 1

2. Metode i tehnike rada 2

3. Xamarin 3
3.1. Način rada Xamarina 3
3.2. Programski jezik C# i razvojni okvir .NET 4
3.3. Xamarin.Forms 4
3.4. Kreiranje novog rješenja u Xamarinu 5

4. Arhitekturalni uzorci dizajna 7
4.1. MVVM 9
4.1.1. Implementacija 10
4.1.2. Zaključak 12
4.2. MVC 14
4.2.1. Implementacija 15
4.2.2. Xamarin i ASP.NET 22
4.2.3. Zaključak 24
4.3. VIPER 26
4.3.1. Implementacija 27
4.3.2. Zaključak 34
4.4. Flux 35
4.4.1. Implementacija 36
4.4.2. Zaključak 43
4.5. Usporedba 44
4.5.1. Vrijeme učitavanja pogleda sa zapisima 44
4.5.2. Broj klasa 45
4.5.3. Analiza koda 46
4.6. Zaključak 49

5. Praktičan primjer 51
5.1. Pregled aplikacije 51
5.2. Alati 53
5.3. Implementacija 53
5.3.1. Korišteni arhitekturalni uzorci dizajna 54
5.3.1.1. MVVM 55
5.3.1.2. MVC 60
5.3.1.3. Flux 62

6. Zaključak 66

Popis literature 69

Popis slika 70

Popis popis tablica 71

1. Uvod

Neprijatelj svakog razvojnog programera je fragmentacija platformi za određenu industriju. Jedna od takvih industrija je mobilna u kojoj su korisnici podjednako podijeljeni na Android i iOS tako da se prilikom razvoja ne smije zanemariti niti jedna niti druga. Za razvoj nativnih Android aplikacija koriste se jezici Java, Kotlin ili C++ unutar Android Studio razvojnog okruženja [1] dok za razvoj iOS aplikacija koriste se jezici Objective-C ili Swift unutar Xcode okruženja [2] tako da se ne poklapa niti jedan programski jezik niti jedno razvojno okruženje. Za navedeni problem postoji nekoliko dobrih rješenja za razvojne programere koji iz nekog razloga ne mogu ili ne žele koristiti nativne tehnologije.

Metino rješenje je React Native čija prva stabilna verzija je izašla 2015. godine. Velika prednost ovoga razvojnog okvira je ta da se koriste jezici HTML5, CSS te JavaScript s kojima je veliki dio razvojnih programera upoznat. Obzirom da je razvijen na temelju React okvira (koristi se za Web aplikacije), ovaj okvir je posebno dobrodošao programerima koji koriste React. Navedeni jezici se kompiliraju u nativne tehnologije za Android i iOS [3].

Pored Mete, drugi veliki igrač u industriji je Google koji je 2017. godine predstavio Flutter. Primarna uloga Fluttera jest ta što nudi alate za brzi UI razvoj gdje se programera oslobađa velikog dijela pisanja koda. Kao programski jezik unutar Fluttera, koristi se Dart te se aplikacija kompilira u nativne tehnologije za Android, iOS, Desktop ili Web aplikacije [4].

Rješenje za ovaj problem ima i Microsoft sa Xamarin tehnologijom koja je tema ovoga rada. Xamarin je platforma otvorenog koda (engl. open-source platform) koja služi za razvoj aplikacija za Windows, Android i iOS pomoću .NET programskog okvira [5]. Prva verzija je stigla već 2011. godine.

Godine 2021., Microsoft je predstavio razvojni okvir .NET MAUI kojeg opisuje kao evolucijsku nadogradnju na Xamarin. Pomoću .NET MAUI-a je moguće, koristeći programski okvir .NET, razvijati aplikacije za Android, iOS, Windows i macOS.

U ovome radu je najprije teorijski prikazana tehnologija Xamarin, njezine prednosti i nedostaci, nakon čega je pojašnjen pojam arhitekturalnih uzoraka dizajna.

Nakon što su pojmovi Xamarin i arhitekturalni uzorak dizajna objašnjeni, opisano je nekoliko uzoraka te je za svaki izrađen jednostavan primjer aplikacije. Arhitekturalni uzorci koji su opisani su MVVM, MVC, VIPER i Flux.

U ovome radu će se prikazati razvoj aplikacija koristeći Xamarin tehnologiju s naglaskom na primjenu različitih arhitekturalnih uzoraka dizajna. Arhitekturalni uzorci koji su opisani su MVVM, MVC, VIPER i Flux te je za svaki izrađen primjer aplikacije.

Na kraju, u praktičnom dijelu rada je napravljena i opisana aplikacija "Faksistent" za kontrolu ostvarenih bodova studenta u toku jednog semestra. Aplikacija je realizirana koristeći principe iz MVVM, MVC i Flux arhitekture.

2. Metode i tehnike rada

Ovaj rad se može podijeliti na dva dijela, teorijski i praktični dio. U teorijskom dijelu se definiraju pojmovi potrebni za ovaj rad te se pregledavaju četiri arhitekturalna uzorka dizajna. U ovome dijelu se koristi razna literatura koja uključuje knjige, članke i dlužbenu dokumentaciju.

Za svaki uzorak dizajna je napravljen primjer aplikacije malog obujma pri čemu se naglasak stavlja na težinu implementiranja uzorka te na buduće nadogradnje aplikacije. Svi primjeri su, jasno, napravljeni u tehnologiji Xamarin što za sobom povlači korištenje programskog jezika C# unutar .NET okruženja. Primjeri su izrađeni koristeći Visual Studio 2019. U poglavlju za usporedbu obrađenih primjera se analizira brzina izvođenja aplikacije, broj klasa te analiza koda tih aplikacija. Bitno je napomenuti da su se aplikacije testirale na virtualnom mobilnom uređaju Google Pixel 2 (Android 9.0) i fizičkom mobilnom uređaju Google Pixel 6.

Za praktični dio je se razvijala samo jedna aplikacija koja je, u usporedbi s primjerima pojedinog uzorka, dosta većeg obujma. Za razvoj aplikacije je korištena službena dokumentacija tehnologija koje se koriste u aplikaciji.

Aplikacija je izrađena u Xamarin tehnologiji koristeći programski jezik C# unutar .NET okruženja. Razlika u odnosu na primjere je korištenje alata Visual Studio 2022. Razvoj praktičnog dijela je potpomognut s nekoliko paketa. To su Newtonsoft.Json koji služi za serijalizaciju i deserijalizaciju objekata, SQLitePCLRaw.core koja omogućava spremanje podataka pomoću SQLite baze podataka. Pored navedena dva paketa, tu je nekoliko DevExpress paketa koji pružaju razne elemente sučelja o čemu će više biti riječi u praktičnom dijelu rada.

3. Xamarin

Kao što je navedeno u uvodu, Xamarin je platforma otvorenog koda za višeplatformski razvoj modernih aplikacija za Windows, Android i iOS pomoću .NET programskog okvira [5]. Za aplikacijsku logiku se koristi C# programski jezik, dok za korisničko sučelje se koristi Microsoftov Xaml jezik koji se prevodi u nativne jezike [5]. Xamarin se trenutno nalazi u verziji 5.0 [6].

U ovome kontekstu, pojam višeplatformskog razvoja aplikacija označava razvoj jednog rješenja koji se može izvoditi na više platformi [7]. Kao što je već spomenuto, Xamarin dopušta višeplatforski razvoj aplikacija za mobilne uređaje što znači da se s jednim rješenjem mogu kreirati aplikacije za Android, iOS i Windows Phone.

Dva su velika razloga zašto koristiti Xamarin ispred nativnih tehnologija:

- Kod, testovi i aplikacijska logika se dijele između platformi [5] što skraćuje vrijeme potrebno za razvoj aplikacije.
- Koristi se C# kao programski jezik te tehnologije kao što su .NET i Visual Studio što olakšava prilagodbu programerima koji već znaju koristiti te tehnologije [5].

Kao što je navedeno u uvodu, cilj rada je napraviti osvrt razvoj više-platformskih aplikacija pomoću Xamarin tehnologije koristeći različite arhitekturalne uzorke dizajna.

U narednim potpoglavljima se može detaljnije pročitati o načinu rada Xamarina, o programskom jeziku C# i razvojnom okruženju .NET koje Xamarin koristi te na kraju je prikazana aplikacija koju Visual Studio automatski kreira.

3.1. Način rada Xamarina

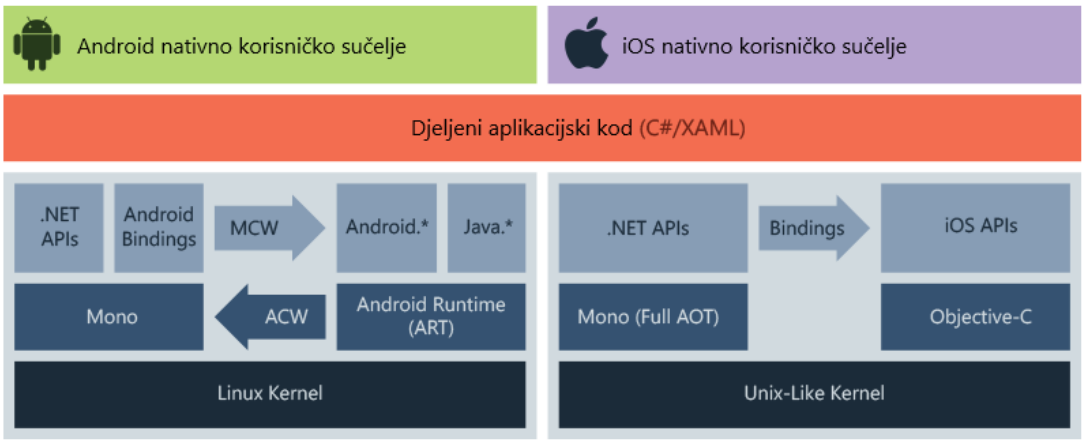
S tehničke strane, Xamarin predstavlja sloj apstrakcije između koda aplikacije i platforme na kojoj se izvodi aplikacija [5].

Prevođenje Xamarin aplikacije u aplikaciju koja se može koristiti na različitim mobilnim operacijskim sustavima se izvršava pomoću okruženja Mono unutar kojega je Xamarin razvijen [5]. Mono je cross-platform implementacija C# kompilira čiji je CLR binarno kompatibilan sa .NET-om. Mono CLR je s vremenom prenesen na mnoge platforme od kojih su, za potrebe ovog rada, najzanimljivije Android i iOS [8].

Na slici 1 se može vidjeti dijagram o načinu rada Xamarina.

Gledajući gornji dio dijagrama, možemo zaključiti to da se aplikacijski kod pisan u C#-u te korisničko sučelje u Xaml-u prevode u nativni kod te u nativne elemente korisničkog sučelja [5]. Pored toga, možemo koristiti i nativne elemente pojedine platforme koje nisu zajedničke za sve platforme.

Na lijevoj strani dijagrama se može vidjeti izvođenje Xamarin aplikacije na Android platformi. Aplikacija se izvodi unutar Mono okruženja s Android Runtime virtualnim strojem. Glavni



Slika 1: Dijagram o načinu rada Xamarina [5]

dio Android aplikacije su veze između .NET API-ja te Android API-ja [8] gdje se C# datoteke, prema Točno-na-vrijeme (engl. Just-In-Time) principu, prevode u datoteke koje Android platforma zna čitati [5].

Za razliku od Androida, iOS aplikacije su u potpunosti ispred-vremena (engl. Ahead-Of-Time) gdje se C# kompilira u ARM asemblerski kod [5].

3.2. Programski jezik C# i razvojni okvir .NET

Programski jezik C# je moderni, objektno orijentirani programski jezik [9]. Svoje korijene pronalazi u C obitelji programskih jezika što znači da se mogu pronaći sličnosti s C, C++, Java i JavaScript jezicima [9]. Prva verzija jezika C# je izašla 2002. godine, a trenutno se nalazi u verziji 10.0 [10].

Programski okvir .NET je besplatna i višeplatformska razvojna platforma za kreiranje mnogih različitih tipova aplikacija [11]. Pojmovi C# i .NET su postali praktično sinonimi obzirom na to da se C# izvodi samo u .NET okruženju, međutim, .NET dopušta korištenje nekoliko jezika koji uključuju c#, F# i Visual Basic.

Kroz godine su razvijene različite tehnologije unutar .NET okruženja, tako da, koristeći .NET, mogu se razvijati desktop aplikacije, Web aplikacije i servisi (ASP.NET) te, s Xamarinom, mogu se razvijati i mobilne aplikacije. Zato je bitno da .NET ima jedan konzistentni API koji sadrži standardizirane biblioteke koji se mogu koristiti u svim .NET aplikacijama [11] tako da se razvojni programeri mogu lako prebaciti na Xamarin ako već imaju iskustva s razvojem u .NET-u.

3.3. Xamarin.Forms

Spomenuto je da za korisničko sučelje, Xamarin koristi jezik Xaml što može predstavljati izazov razvojnim programerima obzirom da ima usku primjenu (osim Xamarina, koristi se samo za razvijanje desktop aplikacija unutar .NET-a). Pomoć u tome pruža okvir za korisničko sučelje

zvani Xamarin.Forms koji pruža niz elemenata korisničkog sučelja koji se prevode u nativno korisničko sučelje [12].

Xamarin.Forms je okvir za korisničko sučelje otvorenog koda koji omogućava kreiranje Android, iOS i Windows aplikacija koristeći samo jednu kodnu bazu [12]. Xamarin.Forms pruža konzinstentni API za kreiranje elemenata korisničkog sučelja koji se koriste na svim platformama te taj API može biti implementiran u Xaml ili C# jeziku.

Na dijelu koda 3.1 se može vidjeti kako izgleda jedan pogled realiziran pomoću Xamarin.Forms-a.

```
1   <StackLayout Spacing="3" Padding="15">
2       <Label Text="Text" FontSize="Medium" />
3       <Entry Text="{Binding Text, Mode=TwoWay}" FontSize="Medium" />
4       <Label Text="Description" FontSize="Medium" />
5       <Editor Text="{Binding Description, Mode=TwoWay}"
            AutoSize="TextChanges" FontSize="Medium" Margin="0" />
6       <StackLayout Orientation="Horizontal">
7           <Button Text="Cancel" Command="{Binding CancelCommand}"
                HorizontalOptions="FillAndExpand"></Button>
8           <Button Text="Save" Command="{Binding SaveCommand}"
                HorizontalOptions="FillAndExpand"></Button>
9       </StackLayout>
10  </StackLayout>
```

Kod 3.1: Primjer Xamarin.Forms

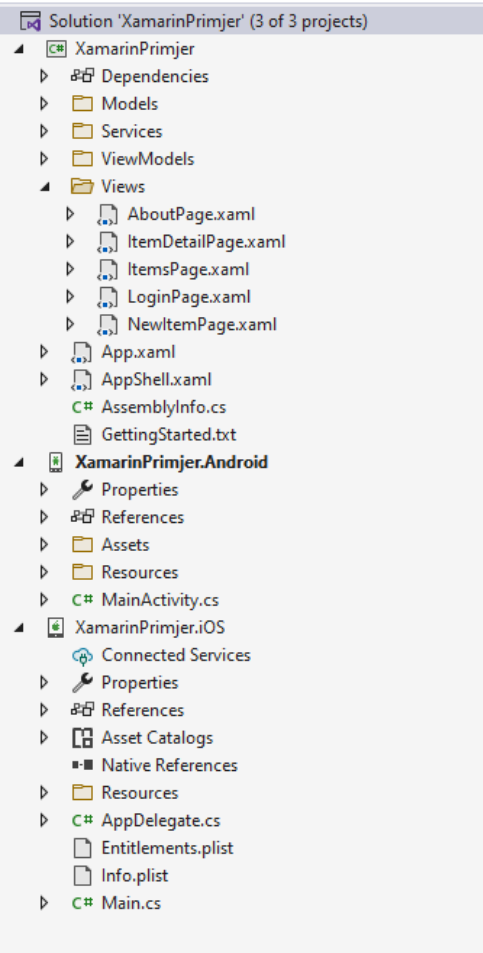
Svaki element unutar dijela koda se prevodi u element korisničkog sučelja. Tako, naprimjer, element Label na drugoj liniji koda se prevodi u tekst. Svaki element ima svoje atribute da bi se željeni element dodatno prilagodio. Važna stavka Xamarin.Forms okvira je vezanje za kontekstualnu klasu (ViewModel u pravilu). Primjer vezanja se može vidjeti na trećoj liniji koda gdje je definirano "Binding Text, Mode=TwoWay" što znači da se taj atribut povlači iz kontekstualne klase te prikazuje korisniku.

Svi elementi korisničkog sučelja koji će se spominjati u ovom radu su dio Xamarin.Forms paketa. Izuzetak je nekoliko elemenata koji su realizirani pomoću DevExpress-a.

3.4. Kreiranje novog rješenja u Xamarinu

Cilj ovog potpoglavlja je pokazati koliko je zapravo jednostavno razviti višeplatformsku aplikaciju pomoću Xamarina. Računalo na kojem se razvija aplikacija treba imati nekoliko instaliranih programa, to su Visual Studio (s kojim dolazi .NET okruženje) te Android Studio za testiranje na Android uređajima i/ili Xcode za testiranje na iOS uređajima. Važno je napomenuti da je za testiranje i objavljivanje na iOS uređajima potrebno imati macOS uređaj.

Nakon što su instalirani svi potrebni programi, preko programa Visual Studio se može kreirati novo rješenje. Na slici 2 se može vidjeti struktura datoteka i direktorija u novokreiranom rješenju.



Slika 2: Zadana struktura datoteka i direktorija

U rješenju pod nazivom "XamarinPrimjer" se nalaze tri projekta, to su "XamarinPrimjer", "XamarinPrimjer.Android" i "XamarinPrimjer.iOS". Razvojnog programera najviše zanima projekt "XamarinPrimjer" jer se u njemu, u pravilu, nalazi naviše koda, dok druga dva projekta sadrže samo neke dijelove koda specifične za određenu platformu. Tako će se u projektu "XamarinPrimjer" nalaziti sve potrebne klase, poslovna logika te pogledi.

Ovu novokreiranu aplikaciju je već moguće pokrenuti i testirati na željenoj platformi. U slučaju da se razvija aplikacija malog obujma, razvojni programer već može krenuti u razvoj pojedinačnih modula, a u slučaju da je aplikacije većeg obujma, preporučuje se da razvojni programer odvoji neko vrijeme za postavljanje arhitekturalnog uzorka dizajna koji će se koristiti kroz cijelu aplikaciju.

4. Arhitekturalni uzorci dizajna

Da bi se razvoj unutar Xamarin tehnologije olakšao, nužno je primjenjivati principe nekog arhitekturalnog uzorka dizajna. U ovom poglavlju će se definirati pojam arhitekturalnog uzorka dizajna, no prvo je potrebno objasniti pojam uzorka dizajna.

Svaki razvojni programer se može "pohvaliti" svojim prvim većim projektom u kojem je tek nakon pola odrađenog posla shvatio da je neke stvari trebao drugačije posložiti da se ne dovede u poziciju gdje projekt postaje teško održiv i još teži za daljnji razvoj. Međutim, dobar razvojni programer će samo jednom napraviti tu grešku te će se, svaki idući put prilikom postavljanja projekta, konzultirati s obaveznom literaturom vezanu uz uzorke dizajna. Uzorak dizajna je opis komunicirajućih objekata i klasa koji su podešeni da riješe opći problem dizajna u pojedinačnom kontekstu [13].

Sličan pojam uzorku dizajna je arhitekturalni uzorak dizajna koji izražava temeljnu organizacijsku shemu softverskog sustava te pruža skup predefiniranih podsustava, specificira njihove odgovornosti i uključuje pravila i smjernice za organiziranje odnosa između njih [13].

Nije točno jasno koliko "velik" uzorak treba biti da bi bio arhitekturalni tako da je često teško pronaći liniju između arhitekturalnih uzoraka i uzoraka dizajna. Idući problem je taj što ne postoji nijedan katalog arhitekturalnih uzoraka, umjesto toga razvojni programeri moraju koristiti opsežne i heterogene literature uzoraka [14].

Kroz povijest programskog inženjerstva su se razvili mnogi arhitekturalni uzorci za različite svrhe, različite programske jezike i tehnologije te za različite subjektivne preferencije razvojnog programera. Tehnologija Xamarin služi za razvoj mobilnih aplikacija ili, drugačije rečeno, služi za razvoj aplikacije za interakciju računala (konkretno mobilnog uređaja) i čovjeka što znači da u ovom radu su zanimljivi samo arhitekturalni uzorci koji služe za tu svrhu, odnosno uzorci interaktivnih sustava. Uzorci interaktivnih sustava služe za sustave koji predstavljaju interakciju između računala i čovjeka. Ti sustavi sadrže dio koji predstavlja aplikacijsku logiku, dio koji predstavlja sučelje za korisnike te, jasno, sustav na neki način mora ostvariti komunikaciju između ta dva dijela [14].

Svaki uzorak koji će se obraditi ima istu funkciju, a to je da pruži sučelje za interakciju čovjeka i računala. Da bi u nastavku lakše razumjeli koje sve uloge obuhvaća neki dio uzorka, navest će se nekoliko funkcija (ili koraka od podatka do prikaza i, obrnuto, od korisničkog unosa do zapisivanja ili obrade podatka) koje treba pružiti sustav koji služi za interakciju čovjeka i računala. Te funkcije su:

- Sadrži klase kao modele za podatke
- Dohvaćanje podataka (poziv API-a ili dohvaćanje iz baze podataka)
- Slanje podataka (na API ili zapisivanje u bazu)
- Pretvorba dobivenih podataka
- Obrada podataka

- Obrada korisnikovog unosa
- Prikaz podataka korisniku
- Rukovanje korisnikovog unosa

Bitna napomena je ta da navedene točke predstavljaju samo putanju podatka iz baze do korisnika i obrnuto, osim navedenih funkcija, svaki uzorak mora riješiti pitanje navigacije kroz aplikaciju, ažuriranja pogleda, testova i sličnih funkcija.

Za pružanja primjera na koji način se implementiraju određeni arhitekturalni uzorci, u ovom radu će se prikazati izrada jednostavne aplikacije koja pruža osnovne funkcije za kreiranje, čitanje, ažuriranje i brisanje (CRUD). Svaki primjer će imati tri pogleda, jedan za prikaz svih stavki, jedan za detalje stavke te jedan za kreiranje i uređivanje stavke. Poslovna logika će se svoditi na komuniciranje s bazom koja će biti, zbog jednostavnosti, datotečna. Pored spomenutog CRUD-a za stavke, dodan je još jedan pogled u kojemu se nalaze zapisi u kojima je zapisano vrijeme učitavanje navedenog pogleda zapisa. Ti zapisi služe za analiziranje performansi prilikom učitavanja neke stranice.

S tehničke strane, svi primjeri su izrađeni koristeći Visual Studio 2019 te su testovi odrađeni na virtualnom mobilnom uređaju Google Pixel 2 (Android 9.0) i na fizičkom mobilnom uređaju Google Pixel 6 (Android 12.0).

Kao što je spomenuto, Xamarin aplikacije se razvijaju za mobilne platforme pomoću .NET razvojnog okruženja te na temelju tih parametara su odabrani uzorci koji se obrađuju u ovome radu. Dakle, zbog vezanosti Xamarina i MVVM uzorka, jasno je da je MVVM uključen. Isto tako je uključen i uzorak MVC zbog svoje značajnosti, kako općenito u industriji, tako i u .NET razvojnom okruženju (konkretnije ASP.NET-u).

Ako se u obzir uzme samo mobilna platforma, zadnjih godina veliku ulogu igra uzorak VIPER kod razvoja iOS aplikacija tako da je i taj uzorak opisan u ovome radu.

Zadnji od obrađenih uzoraka je uzorak Flux i on ima jako malo poveznica sa Xamarinom obzirom da se Flux koristi za razvoj Web aplikacija u JavaScript programskom jeziku, međutim, Flux je uključen u ovaj rad da bi se pokazalo kako se jedan takav arhitekturalni uzorak može iskoristiti u Xamarinu.

Osim povezanosti spomenutih parametara, uzeta je u obzir i raznolikost uzoraka te tako je, naprimjer, iz rada isključen uzorak MVP zbog svoje sličnosti s uzorcima MVVM, MVC i VIPER (VIPER i MVP dijele komponentu Presenter).

U nastavku se nalaze četiri poglavlja za četiri obrađena arhitekturalna uzorka, MVVM, MVC, VIPER i Flux sa primjerom implementacije.

4.1. MVVM

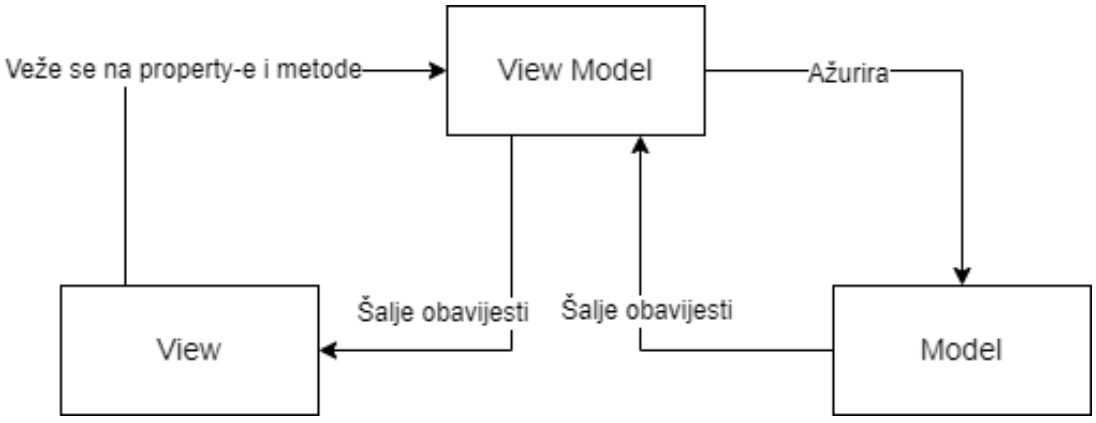
MVVM je jedan u nizu uzoraka koji su nastali po uzoru na MVC. Pojam MVVM je prvi predstavio John Gossman 2005. godine. John Gossman je značajan za .NET platformu jer je jedan od arhitekata WPF-a [15]. WPF je okvir za korisničko sučelje desktop aplikacija [16] te bitno je napomenuti da, osim što sa Xamarinom dijeli .NET platformu (to znači da Model može biti identičan u Xamarinu i WPF-u), dijele korištenje XAML jezika što znači da će i View biti dosta sličan.

Zbog svega toga MVVM je zapravo arhitektura koja se preporučuje za korištenje unutar Xamarina.

Kontrastno broju znakova u kratici, MVVM se sastoji od tri dijela:

- Model – Sadrži poslovnu logiku aplikacije. (Sadrži klase kao modele podataka, dohvaća i šalje podatke, obrađuje dobivene podatke)
- View – Rješava sve vezano za grafički dio aplikacije. Dobivaju podatke od Modela te prikazuju te podatke (prikaz podataka korisniku, rukovođenje korisnikovog unosa)
- ViewModel – Implementira property-je i metode na koje se View može vezati te obavještava View o promjenama stanja. (prosljeđuje podatke Viewu, rukovodi korisnikovim unosom)

Na slici 3 se može vidjeti dijagram koji pokazuje odnos dijelova MVVM uzorka.



Slika 3: MVVM [17]

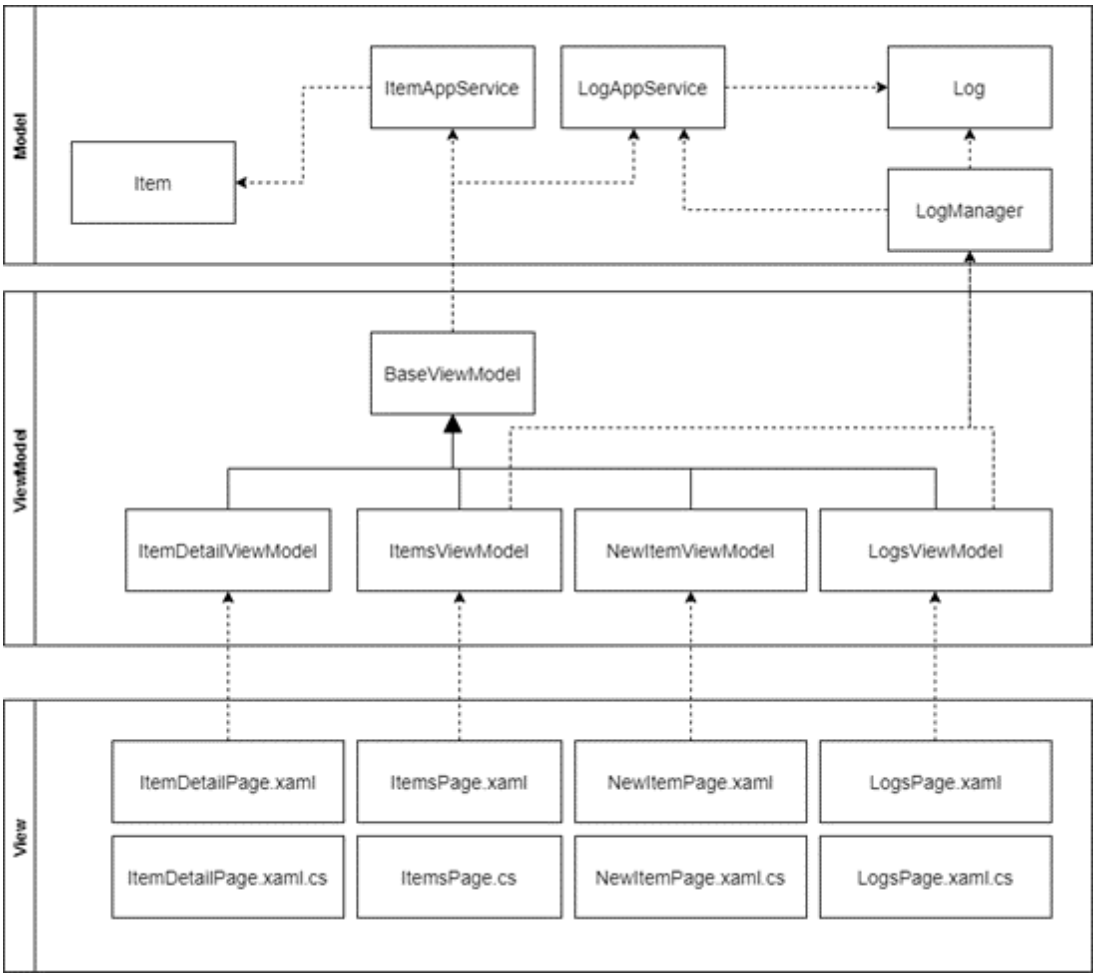
Model je komponenta koja predstavlja poslovnu logiku u aplikaciji, također, može sadržavati klase kao modele podataka. Model može obavještavati i ažurirati samo ViewModel komponentu. Komponenta View prikazuje korisniku podatke te mu pruža mogućnost unosa podataka.

Ono po čemu je uzorak MVVM poseban jest komponenta ViewModel. Ta komponenta pruža View-u podatke, zadužena je za slanje obavijesti i ažuriranje View-a. ViewModel pruža metode i property-e na koje se View komponente mogu vezati. Također, kao i Controller za

MVC, ViewModel povezuje preostala dva dijela uzorka tako što preuzima podatke od Modela te može ažurirati podatke na Model-u.

4.1.1. Implementacija

Kao što je već spomenuto, MVVM arhitektura je arhitektura koja se preporučuje za korištenje u Xamarinu i prilikom kreiranja novog projekta, MVVM bude već postavljen. U primjeru je korištena već postavljena arhitektura bez ikakvih promjena, iako uzorak pruža slobodu da se neke stvari promijene poput načina navigacije. Na slici 4 može se vidjeti dijagram klasa za ovaj primjer.



Slika 4: MVVM - dijagram klasa

Na dijagramu klasa se mogu vidjeti tri odvojene komponente: View, ViewModel i Model. Komponenta View sadrži četiri pogleda sa četiri pripadne klase koje služe samo za vezanje na ViewModel klase.

Komponenta ViewModel sadrži četiri ViewModel klase, svaka je vezana uz jedan od pogleda te klasu BaseViewModel.cs koja je natklasa konkretnim ViewModel klasama. Obzirom da je ovo primjer malog obujma, klasa BaseViewModel.cs sadrži referencu na dvije AppService klase (koje služe za dohvaćanje i spremanje podataka). U praktičnom primjeru ovog rada, koji

sadržava veći broj AppService klasa, te klase se referenciraju u svakoj konkretnoj ViewModel klasi.

U komponenti Model se nalaze dvije spomenute AppService klase (ItemAppService.cs i LogAppService.cs), klase Item.cs i Log.cs koje služe za modele podataka te klasa LogManager.cs.

Na dijagramu klasa su izostavljena dva pogleda, App i AppShell. Bitno je napomenuti da klasa MainActivity u Android projektu, odnosno Main u iOS projektu prvo inicijaliziraju App koji u sebi sadrži referencu na AppShell. Zadana implementacija MVVM-a u Xamarinu unutar AppShell-a stavlja navigaciju koja u ovom primjeru nije potrebna obzirom da se svaki pogled dobiva iz pogleda sa svim stavkama. Tako da ovaj pogled u sebi sadrži samo referencu za pogled sa svim stavkama (ItemsPage). U pripadnoj klasi (AppShell.xaml.cs) se nalazi kod za registriranje svih ruta koji se može vidjeti na primjeru koda 4.1. Spomenuta dva pogleda i njihove pripadne klase se nalaze u svakom Xamarin projektu i mogu se smatrati kontejnerima aplikacije.

```
1         public AppShell()
2         {
3             InitializeComponent();
4
5             Routing.RegisterRoute(nameof(ItemDetailPage),
                  typeof(ItemDetailPage));
6             Routing.RegisterRoute(nameof(NewItemPage),
                  typeof(NewItemPage));
7             Routing.RegisterRoute(nameof(LogsPage), typeof(LogsPage));
8         }
```

Kod 4.1: MVVM - registriranje ruta

Za registriranje rute su potrebna dva argumenta, prvi je tipa string koji definira naziv rute koja će se kasnije dohvaćati a drugi je klasa koja je vezana uz određeni View.

Kako je ovaj primjer vrlo jednostavan, poslovna logika se svodi samo na komuniciranje s datotekom koja služi kao spremište podataka. Tako da se cijeli Model sastoji od svega pet klasa koje se mogu podijeliti na klase koje služe za zapise i klase koje služe za stavke. Klase Item.cs i Log.cs služe kao modeli podataka. S druge strane klase ItemAppService.cs i LogAppService.cs služe za komuniciranje s datotekama te nude metode za zapisivanje i dohvaćanje podataka. Klasa koja nije spomenuta, LogManager.cs se ponaša kao Singleton te pojednostavljuje spremanje zapisa.

Već je spomenuto da se prilikom kreiranja projekta dobije Boilerplate kod napisan u MVVM arhitekturi. U tom kodu, između ostalog se nalazi klasa BaseViewModel.cs koja služi kao natklasa ostalim ViewModel-ima. Ona sadrži referencu na ItemAppService i LogAppService (to ne bi imalo smisla u većim aplikacijama s više servisa osim ako bi se koristio Unit of Work koncept ili neki sličan) da se te klase ne moraju inicijalizirati na svakome ViewModel-u zasebno.

Najveću razliku u odnosu na druge uzorke, jasno, predstavlja ViewModel. On u sebi

sadrži pozive Model-a te definira metode koje se vežu na određene elemente u View-u. Tako da će, naprimjer, ItemsViewModel implementirati metodu za učitavanje svih stavki i komande za prijelaz na ostale tri stranice.

Primjer prelaženja na drugu stranicu možemo vidjeti na dijelu koda 4.2.

```
async void OnItemSelected(Item item)
{
    if (item == null)
        return;

    await Shell.Current.GoToAsync($"{nameof(ItemDetailPage)}" +
    "?{nameof(ItemDetailViewModel.ItemId)}={item.Id}");
}
```

Kod 4.2: MVVM - registriranje ruta

Na dijelu koda 4.1 se možemo prisjetiti kako smo, prilikom registriranja rute, definirali naziv rute koji u ovom koraku trebamo pozvati. Obzirom da u ovom primjeru želimo se premjestiti s pogleda svih stavki na pogled detalja jedne stavke, metodi za navigaciju kao argument prosljeđujemo naziv kojeg smo pridružili tom pogledu a to je u ovom slučaju nameof(ItemDetailPage) odnosno string "ItemDetailPage". To je sasvim dovoljno da se prebacimo na taj pogled, međutim, taj pogled ne zna koju stavku treba prikazati pa na naziv rute dodajemo znak "?" te nakon njega definiramo ključ i vrijednost argumenta. Ovakav način prosljeđivanja argumenata podsjeća na prosljeđivanje URL parametara na Webu iako postoje neke razlike kad želimo poslati listu objekata, međutim to se ne preporučuje obzirom da brže i bolje tu listu učitati u novom pogledu.

Sveukupna navigacija u ovom primjeru je jednostavna te ju je moguće shvatiti bez dodatnih dijagrama, ali u svrhu stvaranje referentne točke za ostale uzorke gdje je to malo kompliciranije, napravljen je dijagram aktivnosti koji prikazuje cjelokupnu navigaciju od početne metode koja zahtjeva navigaciju do pozivanja Shell.Cureen.GoToAsync() metode. Taj dijagram možemo vidjeti na slici 5.

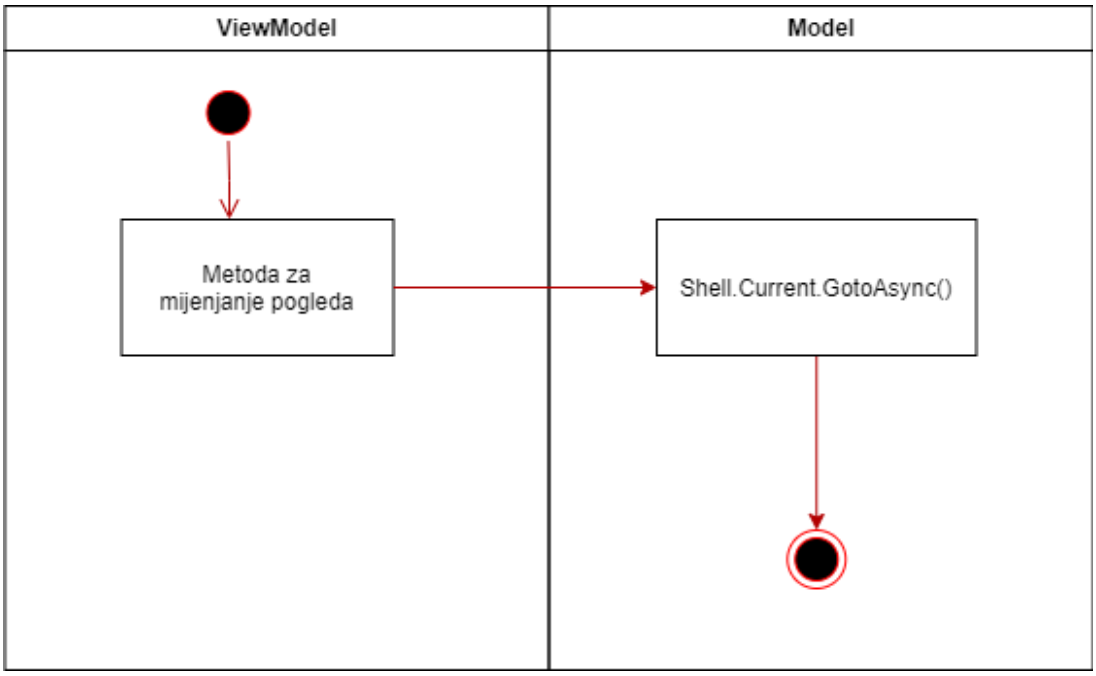
Navigacija ovog primjera je svedena na samo dvije već spomenute metode, metoda koja zahtjeva mijenjanje pogleda te metoda Shell.Cureen.GoToAsync().

4.1.2. Zaključak

Kao što je spomenuto, MVVM je uzorak koji Microsoft preporučuje [17] za korištenje u Xamarin okruženju. Obzirom na to, u pravilu, svaki Xamarin pogled se veže uz jednu klasu ViewModel-a što olakšava protok podataka i okidača između View-a i ViewModel-a.

Prilikom kreacije novog Xamarin projekta, Visual Studio generira boilerplate kod koji je u MVVM arhitekturi, tako da za MVVM možemo reći da nudi najbrži početak razvoja u Xamarinu jer je razvojnom programeru vrlo lako pratiti primjer CRUD-a koji je dobio u boilerplate dijelu koda. Međutim, to ne znači da je MVVM prilagođen samo za aplikacije manjeg obujma.

Rekli smo da ova implementacija prati boilerplate kod te smo spomenuli neke mane



Slika 5: MVVM - dijagram aktivnosti za navigaciju

poput zasebnog registriranja svake rute i nezgrapnog načina za promjenu pogleda gdje je lako pogriješiti, ali kad bi išli u razvoj aplikacije većeg obujma, MVVM nudi slobodu da se ti nedostatci riješe na drugi način (neki od mogućih načina su prikazani u narednim poglavljima kod drugih arhitektura).

Unatoč svim prednostima i nedostacima svih arhitekturalnih uzoraka, MVVM ostaje standard za Xamarin ponajviše zbog Xamarinove podrške toj arhitekturi.

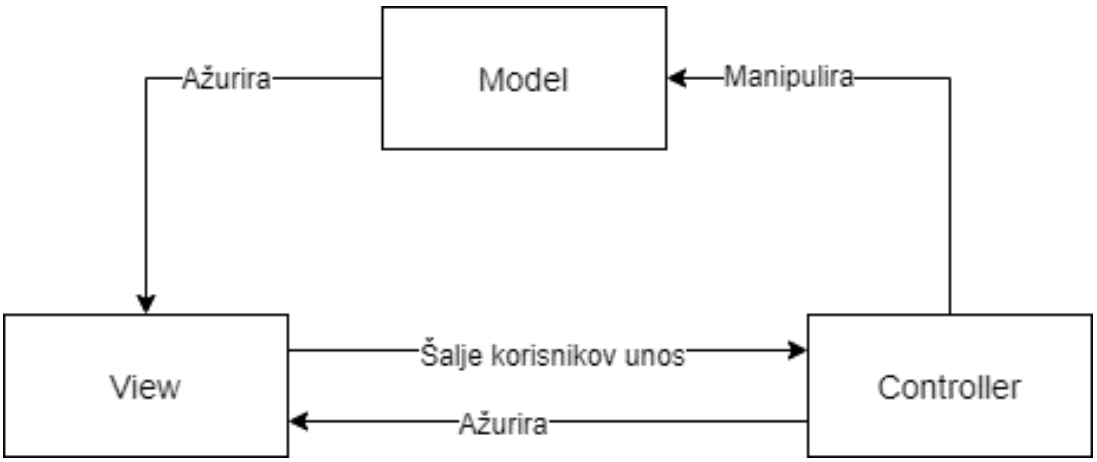
4.2. MVC

MVC je praktički sinonim za uzorke interaktivnih sustava zahvaljujući svojoj dugoj povijesti te širokoj primjeni u različitim programskim jezicima. MVC svoje početke ima u jeziku Smalltalk 70-ih godina 20. stoljeća [18]. Obzirom na svoju dugu povijest, mnoge promjene su bile neizbježne za MVC kako bi i dalje s vremenom bio relevantan. Kao posljedica toga, iz MVC-a su se razvili i neki drugi slični uzorci kao što su MVVM, MVP i MVA.

Uzorak MVC se sastoji od:

- Model – poslovna logika aplikacije (Sadrži klase kao modele podataka, dohvaća i šalje podatke, obrađuje dobivene podatke)
- View – rješava sve vezano za grafički dio aplikacije. Dobiva podatke od Modela i Controllera te prikazuju te podatke (prikaz podataka korisniku, rukovođenje korisnikovog unosa)
- Controller – služi kao sučelje između Viewa i Modela (prosljeđivanje podataka Viewu, rukovođenje korisnikovim unosom, služi i kao navigacija)

Način na koji su dijelovi MVC arhitekture povezani se mogu vidjeti na slici 6.



Slika 6: MVC [19]

Kao i kod većine ostalih uzoraka interaktivnih sustava, odmah se mogu primijetiti Model i View te iz toga se da zaključiti čemu otprilike služe te dvije komponente. Model, osim što pruža poslovnu logiku aplikacije, registrira View-ove i Controllere te ih obavještava o promjenama podataka [13]. View je zadužen za prikazivanje podataka korisniku te pruža korisniku mogućnost unosa. I View i Model su veoma slični istoimenim komponentama u MVVM uzorku uz razliku da Model u MVC može ažurirati View i Controller komponentu, dok u MVVM-u može samo ViewModel.

U MVC arhitekturi, kao vezivo između Modela i Viewa služi Controller. Obzirom na to da je „samo“ vezivo te da najveći dijelovi aplikacije kao što su poslovna logika i prikaz podataka sadrže Model i View, Controller, u pravilu, ne bi trebao sadržavati velik dio koda. Međutim, ono što Controller sadrži je dio koda koji dobiva podatke od Modela te te iste podatke prosljeđuje

Viewu čime ga ažurira, također, od Viewa dobiva korisnikov unos pa, ovisno o unosu, može manipulirati zahtjevom na Modelu ili može ažurirati pogled. [20]

Kao što je navedeno, arhitekturalni uzorak MVC je predstavljen 70-ih godina prošlog stoljeća kada je programsko inženjerstvo bilo tek u ranom razvitku. Kroz tih skoro 50 godina su se razvili mnogi jezici, mnogi okviri, te možda najbitnije, pojavile su se mnoge platforme, no unatoč tome, MVC je i dalje standard u velikom dijelu programskih jezika i okvira.

Jasno, MVC je kroz to vrijeme doživio neke promjene te uloge komponenata se razlikuju na različitim tehnologijama.

4.2.1. Implementacija

Obzirom na sličnost komponenata u MVC i MVVM uzorku, za implementaciju MVC uzorka možemo koristiti isti Model kao i u MVVM uzorku.

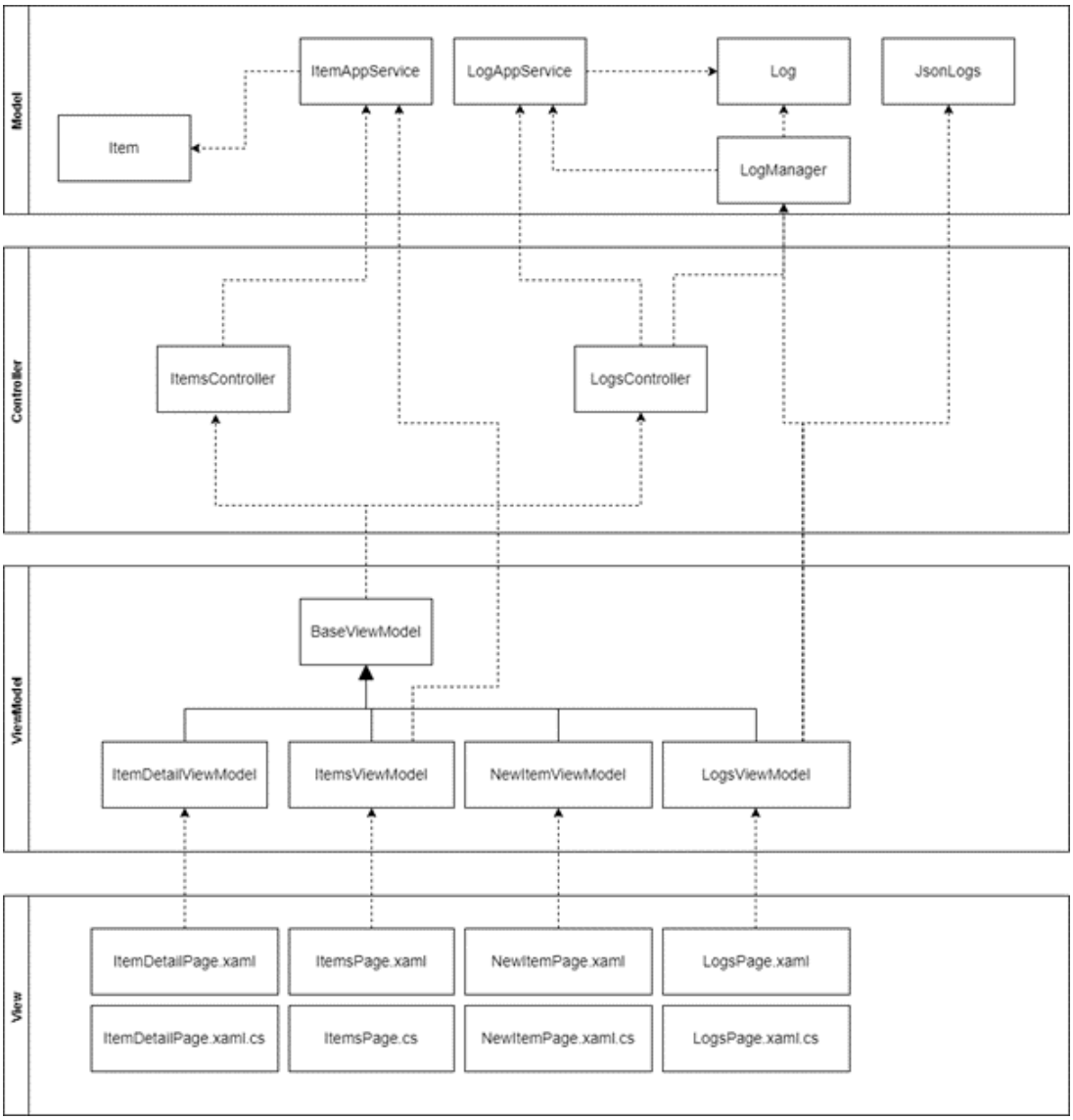
Kao što je ranije spomenuto, da bi MVC ostao relevantan kroz godine, morao je doživjeti mnoge promjene tako da sad postoje različite implementacije. Za ovu konkretnu implementaciju, uzor je pronađen u ASP.NET MVC arhitekturi zbog sličnosti tehnologija. Iako ovo rješenje možda ne nudi najbolju moguću implementaciju, može biti korisna, zbog sličnosti implementacije, velikom broju razvojnih programera koji su prethodno razvijali web aplikacije u .NET okruženju.

Iako se koristi isti razvojni okvir kao ASP.NET, implementacija uzorka ne može biti identična. Prisjetimo se načina na koji su vezani View i ViewModel u MVVM uzorku. ViewModel sadrži varijable i metode na koje se View može vezati. Klase koje pripadaju ViewModel-u su specifične u Xamarinu tako što nude neke osnovne funkcije poput instanciranja klase Command koja okida metodu na View-ov zahtjev, nadalje, u ViewModel-u se nalaze sve vrijednosti koje View dinamički treba učitati, bio to statični tekst koji se nalazi u ViewModel-u radi dobre prakse ili bila to lista nekog objekta koju želimo prikazati u tablici. Također, zbog vezanosti View-a na ViewModel, unutar ViewModel-a se nalaze varijable koje se dobiju iz parametara upita prilikom prelaska na taj pogled.

Zbog toga svega navedenoga, iako bi bilo moguće napraviti arhitekturu bez ViewModel-a, to bi previše opterećivalo klase *.xaml.cs koje su vezane uz pogled pa će se svaki View, i u MVC i u svakom idućem uzorku, i dalje vezati na ViewModel, međutim uloga ViewModel-a će se smanjiti samo na spomenuto specifičnosti te, ukoliko je potrebno, na asinkrono učitavanje podataka. ViewModel možemo smatrati dijelom View-a u MVC arhitekturi.

Dijagram klasa ovog primjera se može vidjeti na slici 7.

Na temelju dijagrama klasa, razlika između MVVM-a i MVC-a je ta što MVC sadrži komponentu Controller koja, jasno, sadrži sve Controller klase. Ta komponenta sadrži tri klase od kojih je jedna nazvana Controller.cs koja je natklasa svakom konkretnom Controlleru čime se olakšava raspoznavanje koja klasa je Controller te može sadržavati neke generičke metode koje su potrebne (u ovom primjeru se nalazi samo jedna metoda za navigaciju na drugu stranicu što će se navesti kasnije).



Slika 7: MVC - dijagram klasa

Ostale komponente sadrže iste klase kao i MVVM primjer s tim da ViewModel klase imaju manju odgovornost nego u MVVM primjeru obzirom da je komponenta Controller preuzela dio odgovornosti.

Prisjetimo se, u MVVM primjeru rute su se registrirale tako da svaku zasebnu stranicu je potrebno dodati u AppShell, no obzirom da u MVC arhitekturi svaku stranicu možemo definirati kao metodu unutar nekog od Controller-a, te stranice možemo registrirati pomoću tih metoda. Registriranje ruta se može vidjeti na dijelu koda 4.3.

```
1  public async Task RegisterRoutes()
2          {
3              var controllers = Assembly
4                  .GetAssembly(typeof(Controller))
5                  .GetTypes()
6                  .Where(t => t.IsSubclassOf(typeof(Controller)));
7
```

```
8               foreach (var controller in controllers)
9               {
10                  var controllerName = controller.Name.Replace("Controller",
                        "");
11
12                  var methods = controller.GetMethods();
13
14                  foreach (var method in methods)
15                  {
16                      var type = Type.GetType("MVC.Views." + controllerName
                            + "." + method.Name);
17                      Routing.RegisterRoute(controllerName + "/" +
                            method.Name, type);
18                  }
19              }
20          }
```

Kod 4.3: MVC - registriranje ruta

Na dijelu koda se vidi metoda RegisterRoutes() koja se poziva iz konstruktora AppShell.xaml.cs klase i ta metoda, pomoću refleksije, dohvaća sve klase koje su potklase klasi Controller.cs, nakon toga se za svaku klasu dohvaćaju sve metode te se registrira ruta pod nazivom "naziv Controller-a/naziv metode". Pored naziva, potrebno je definirati klasu pripadnog View-a koji se refleksijom dohvaća pomoću naziva controllera i metode. Možemo primijetiti da ova implementacija forsira strogo definiranje naziva Controller-a, metode te View-a koji se mora nalaziti u za to predviđenoj mapi. Ovo ograničenje definitivno predstavlja problem, međutim može se riješiti pomoću anotacija što nije implementirano u ovom primjeru je, obzirom na obujam, nije bilo potrebe.

Ovakav način registriranja ruta, također, ima nedostatak jer koristi refleksiju koja je relativno spora, međutim, metoda se okida samo jednom prilikom pokretanja aplikacije te je asinkrona što znači da neće usporiti pokretanje aplikacije i dovoljno je brza da korisnik ne može zahtijevati navigaciju prije nego što se izvrši. Dakako, prednost je u tome što se ne treba svaki put iznova definirati ruta prilikom dodavanja nove stranice, potrebno je samo dodati novu metodu u jedan od Controller-a.

Kao što je već spomenuto, MVVM i MVC dijele Model komponentu te ta komponenta je kao u primjeru za MVVM.

Klasa Controller.cs služi, osim što služi za raspoznavanje klasa, sadrži metodu za navigaciju koju možemo vidjeti na dijelu koda 4.4.

```
1  protected async Task ChangeView(object model = null, bool isJson = false,
       string methodName = "", string controllerName = "",
2              [CallerMemberName] string callerName = "", [CallerFilePath]
                   string callerClass = "")
3          {
4              if (methodName == "")
5              {
```

```
6                   methodName = callerName;
7               }
8               if (string.IsNullOrEmpty(controllerName))
9               {
10                  var split = callerClass.Split('\\');
11                  controllerName = split[split.Length - 1];
12                  controllerName = controllerName.Replace("Controller.cs",
                        "");
13              }
14              string query = "";
15              if(model != null)
16              {
17                  if (isJson)
18                  {
19                      query = "?JsonValue=" +
                            JsonConvert.SerializeObject(model);
20                  }
21                  else
22                  {
23                      query = "?" + ToQueryString(model);
24                  }
25              }
26              await
                    Shell.Current.GoToAsync($"{controllerName}/{methodName}{query}");
27          }
```

Kod 4.4: MVC - navigacija

Zadani način navigacije među stranicama, u Xamarinu, je pomoću statične metode Shell.Current.GoToAsync() koja se koristi i u ovom primjeru kao što se može vidjeti na zadnjoj liniji metode ChangeView.

Navedeno je kako su metode unutar Controllera vezane jedan na jedan s pogledima tako da se kod zadanog načina u MVVM arhitekturi (što je nepotrebno u ovoj arhitekturi) mora definirati naziv rute. Ova metoda rješava taj problem tako što se naziv rute refleksijom dohvaća iz naziva metode koja poziva ChangeView metodu. Pored toga, dodana je mogućnost da se u metodi koja poziva mogu proslijediti parametri methodName i controllerName tako da se ostavlja mogućnost prelaska na neku drugu stranicu ako za to ima potrebe. Ova opcija se ne koristi u ovom primjeru. Nedostatak ovog načina navigacije je ta što refleksija uzima više vremena nego što bi to bio slučaj sa zadanim načinom što će se pokazati u poglavlju za usporedbu uzoraka.

Bitna stavka prilikom navigacije je prosljeđivanje argumenata u novi pogled. Metoda ChangeView pojednostavljuje taj proces tako što se toj metodi proslijedi objekt koji se serijalizira u upit. Ponuđena su dva načina serijalizacije objekta, jedan je gdje se objekt serijalizira u parametre tako da upit izgleda kao onaj u MVVM primjeru, dok je drugi način taj da se taj objekt može serijalizirati u JSON objekt što je jednostavnije ako je potrebno poslati kompliciraniji objekt ili listu objekata. Nadalje, dobiveni JSON se deserijalizira u ViewModel-u natrag u objekt koji

je potreban. Oba načina su sporija od zadanog načina zbog potrebe serijaliziranja objekata, međutim, olakšavaju posao razvojnom programeru jer je lakše definirati objekt nego ručno serijalizirati sve parametre koje želi poslati. Dodatna prednost za razvojnog programera je ta što se izbjegavaju mogući tipfeleri.

Primjer konkretnog Controller-a možemo vidjeti na dijelu koda 4.5. (ItemsController.cs)

```
1  public class ItemsController : Controller
2      {
3          private readonly ItemAppService _itemAppService;
4          public ItemsController()
5          {
6              _itemAppService = new ItemAppService();
7          }
8
9          public async Task ItemDetailPage(string itemId)
10         {
11             await ChangeView(new { ItemId = itemId });
12         }
13
14         public async Task NewItemPage()
15         {
16             await ChangeView();
17         }
18
19         public async Task Items2Page()
20         {
21             var items = await _itemAppService.GetItemsAsync();
22             await ChangeView(new
23             {
24                 Items = items
25             }, true);
26         }
27     }
```

Kod 4.5: MVC - Metoda za promjenu pogleda

U aplikaciji, postoje tri pogleda vezana za stavke, to su ItemsPage, ItemDetailPage i NewItemPage. U ovoj aplikaciji nije bilo potrebno u Controller-u definirati metodu za ItemsPage jer je ta stranica definirana u navigaciji, međutim u svrhu pokazivanja primjera za JSON dodana je i metoda Items2Page. Ovaj Controller je idealan za primjer jer se u njemu nalaze tri različita načina pozivanja metode ChangeView koja je prikazana ranije.

Prvi, najjednostavniji, način je mijenjanje pogleda bez parametara (NewItemPage) te vidimo da je potrebno samo pozvati metodu za promjenu pogleda bez ikakvih parametara. Drugi način je za ItemDetailPage kojemu je potrebno proslijediti jedan parametar (nije označeno da će se objekt serijalizirati u JSON tako da se parametar šalje kao standardni upit). Treći način je prosljeđivanje parametara pomoću JSON-a jer se želi proslijediti lista stavki za što je, pored objekta, potrebno poslati vrijednost „true“ za parametar isJson.

Osim drugačijeg raspoređivanja po mapama te, posljedično, drugačijeg namespace-a, View komponenta je ista kao u MVVM primjeru.

Kao što je već spomenuto, ovaj primjer sadrži i ViewModel komponentu koja je slična onoj u MVVM primjeru, ali je olakšana za neke stvari koje su prebačene u Controller. Tako da ovaj ViewModel ne dohvaća podatke iz Model-a (iako se može koristiti u tu svrhu ako je potrebno asinkrono osvježavanje podataka). Što se tiče direktnog dohvaćanja podataka iz Model-a, to također može ići preko metode u kontroleru, međutim, taj korak je često redundantan pa se u ovom primjeru ne koristi to. Ako bi dobivene podatke trebalo na neki način obraditi ili mapirati u objekt druge klase, preporučuje se da dohvaćanje podataka ipak ide preko kontrolera jer je cilj maksimalno "olakšati" ViewModel.

Na dijelu koda 4.6 možemo vidjeti kako funkcionira mijenjanje pogleda iz perspektive ViewModel-a.

```
1         private async void OnAddItem(object obj)
2         {
3             InvokeControllerMethod("ItemsController", "NewItemPage");
4         }
5
6         private async void OnLogs(object obj)
7         {
8             LogManager.GetInstance().StartTime = DateTime.Now.Ticks;
9             InvokeControllerMethod("LogsController", "LogsPage");
10        }
11
12        async void OnItemSelected(Item item)
13        {
14            if (item == null)
15                return;
16
17            InvokeControllerMethod("ItemsController", "ItemDetailPage", new
18            {
19                itemId = item.Id
20            });
21        }
```

Kod 4.6: MVC - Promjena pogleda iz ViewModel-a

Prikazane tri metode se nalaze u ItemsViewModel-u te vidimo da svaka poziva metodu InvokeControllerMethod() s dva ili tri parametra. Prvi je naziv kontrolera, drugi je naziv metode u kontroleru dok treći koji je opcionalan je objekt koji želimo proslijediti metodi koja je navedena.

Metoda InvokeControllerMethod() se nalazi u klasi BaseViewModel.cs koje se sjećamo iz MVVM dijela. Ta metoda pomoću refleksije instancira potrebni kontroler te poziva njegovu metodu. Ovaj način instanciranja i pozivanja pomoću refleksije nije optimalan obzirom da je refleksija sporija od običnog instanciranja te je problem u tome što IntelliSense (pojam za inteligentno dovršavanje koda u Visual Studiu) neće uočiti grešku ako se proslijedi pogrešan string za kontroler ili metodu i što neće uočiti grešku ako proslijedimo krive parametre ili ih ne prosli-

jedimo a metoda ih očekuje. Prednost ovakvog načina je generaliziranje načina pozivanja tako da se ne mora instancirati svaki kontroler zasebno. Ove probleme je, u nekoj razini, moguće riješiti s Unit of Work uzorkom.

Ako se prisjetimo načina na koji se parametri mogu proslijediti pogledu, znamo da se ti parametri dohvaćaju u ViewModel-u. Spomenuto je da je za ovaj uzorak specifično da se parametri mogu proslijediti u JSON obliku. Na dijelu koda 4.7 možemo vidjeti kako izgleda deserijalizacija JSON-a.

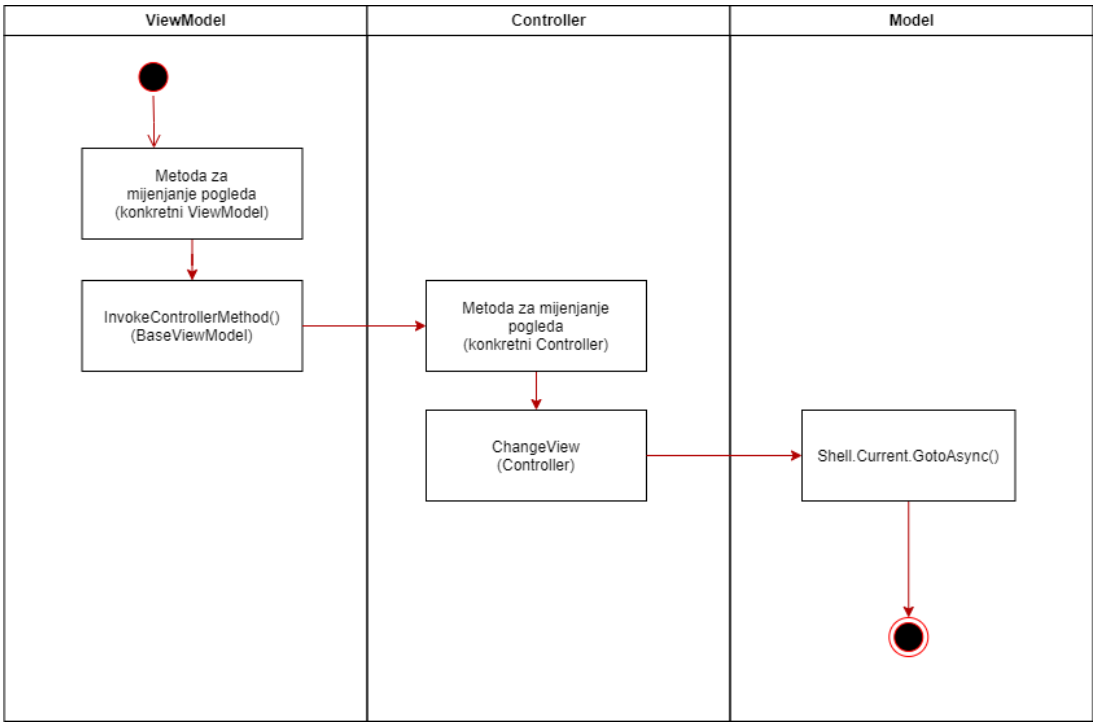
```
1         async Task ExecuteLoadLogsCommand()
2         {
3             IsBusy = true;
4
5             try
6             {
7                 Logs.Clear();
8                 var logs = await _logAppService.GetLogsAsync();
9                 foreach (var log in logs)
10                {
11                    Logs.Add(log);
12                }
13                Total = Logs.Count;
14                Average = Logs.Average(x => x.Duration);
15            }
16            catch (Exception ex)
17            {
18                Debug.WriteLine(ex);
19            }
20            finally
21            {
22                LogManager.GetInstance().EndTime = DateTime.Now.Ticks;
23                await LogManager.GetInstance().GenerateLog();
24                IsBusy = false;
25            }
26        }
```

Kod 4.7: MVC - Deserijalizacija JSON-a u ViewModel-u

Metoda koju vidimo se nalazi u klasi LogsViewModel.cs te se okida prilikom učitavanja pogleda. Kao što vidimo, JSON vrijednost se deserijalizira u varijablu naziva "logs" koja je lista zapisa.

Za razliku od uzorka MVVM, ovaj uzorak ima više koraka u mijenjanju pogleda te sve te korake možemo vidjeti na dijagramu aktivnosti koji se nalazi na slici 8.

Kao što vidimo, ovaj način navigacije ima nekoliko koraka više od MVVM primjera. Metoda unutar ViewModel-a preko InvokeControllerMethod() poziva metodu u kontroleru koja preko ChangeView() i Shell.Current.GoToAsync() mijenja pogled. Iako ovaj dijagram sadrži 3 koraka više nego MVVM, metode InvokeControllerMethod() i ChangeView() se već nalaze u Boilerplate dijelu koda te se ponovno koriste za svaku navigaciju tako da ovaj način ima samo



Slika 8: MVC - dijagram aktivnosti za navigaciju

jedan korak više od MVVM primjera što se nije velika razlika pogotovo kad se u obzir uzme to da metoda unutar konkretnog kontrolera može služiti i za druge svrhe kao naprimjer dohvaćanje podataka.

4.2.2. Xamarin i ASP.NET

U više navrata je spomenuto kako ASP.NET koristi MVC kao zadanu arhitekturu te da je u toj implementaciji MVC arhitekture pronađen uzor za implementaciju u Xamarinu. Odmah se nameće pitanje mogu li se te dvije tehnologije spojiti u jedan projekt. Odgovor je, naravno, da mogu, samo je pitanje u kojoj mjeri bi se isti kod mogao koristiti za oba rješenja.

Model u Xamarinu i Model u ASP.NET-u bi trebao biti isti, oba okvira koriste C# jezik i .NET im dopušta da koriste iste biblioteke. Implementacija View-a u ovom radu obuhvaća ViewModel i poglede koji su pisani u XAML jeziku, dok ASP.NET View implementira s HTML i JavaScript jezikom (moguće je koristiti i Razor ili Blazor stranice). Dakle, Model je vrlo lako realizirati tako da oba projekta koriste isti kod, što nije moguće za komponentu View.

Kod komponente Controller nastaje siva zona, kod oba okvira se koristi C# kod te su sve klase slične, međutim, postoje neke razlike kao što je npr. to da svaka metoda u Controlleru unutar ASP.NET rješenja mora vraćati implementaciju interfacea IActionResult. Nadalje, uloge Controllera u jednom i drugom projektu se ne poklapaju u potpunosti, jer po prirodi, ASP.NET Controller će imati više metoda koje će rukovoditi korisničkim unosom.

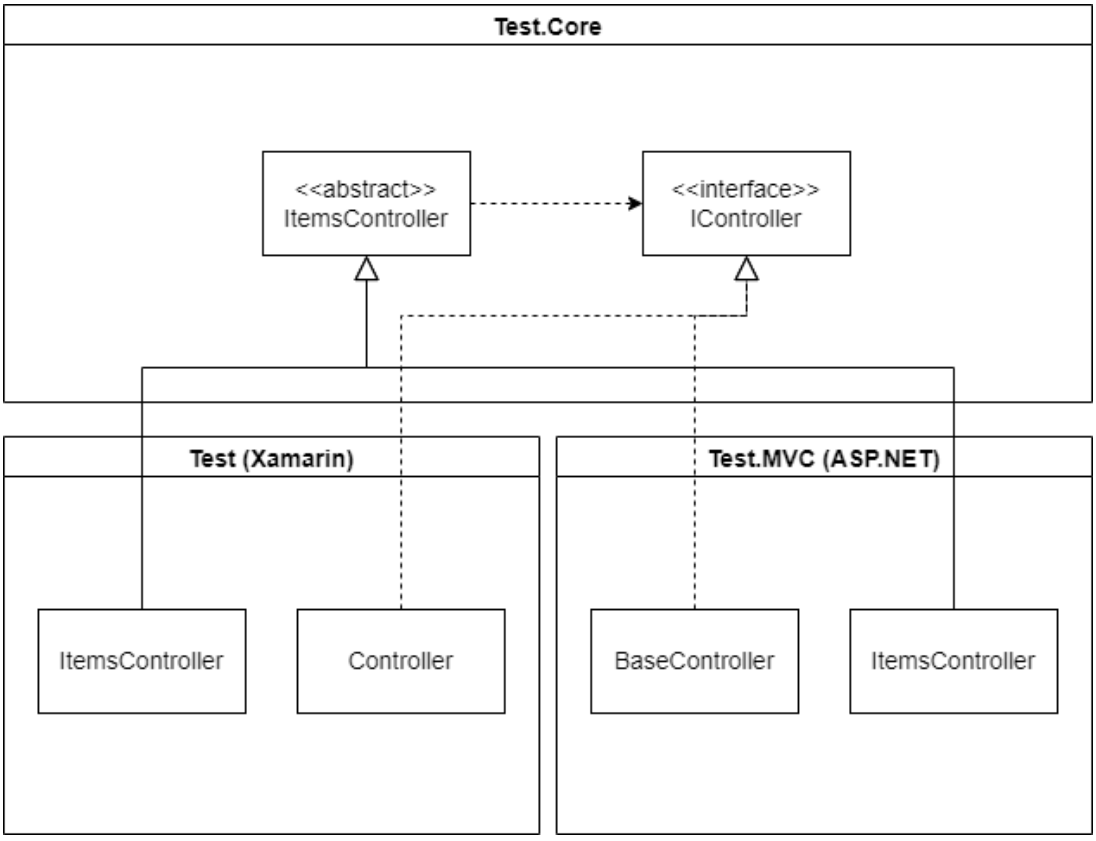
U svrhu rada je kreiran primjer koji prikazuje realizaciju ideje da se u isto rješenje stave i Xamarin i ASP.NET aplikacija. Izrađena je slična, ali malo jednostavnija aplikacija kao aplikacije koje služe za pružanje primjera implementacije pojedine arhitekture. Aplikacija ima tri stranice,

jednu za pregled svih stavki, drugu za dodavanje stavke i treću za pregled detalja stavke.

Prilikom kreacije, Visual Studio, za Xamarin aplikaciju, kreira tri projekta unutar rješenja. Jedan je dijeljeni projekt (u ovom primjeru se taj projekt zove Test) kojeg referenciraju druga dva projekta, a to su Android (Test.Android) i iOS (Test.iOS). Većina koda se nalazi unutar tog dijeljenog projekta te kako svi primjeri u ovom radu nisu dosegli dovoljnu razinu kompleksnosti, sve MVC komponente se nalaze unutar tog projekta. Međutim, za ovu implementaciju je potrebno dodati još jedan ASP.NET projekt (Test.MVC) te jedan zajedinčki projekt (Test.Core) kojeg će koristiti i Xamarin i ASP.NET.

Sav kod za komponentu Model je preseljen u Test.Core projekt, dok sav kod za komponentu View je zasebno napravljen i za Xamarin i za ASP.NET. Glavni predmet razmišljanja će biti Controller.

Da bi bolje predočili način na koji radi komponenta Controller, poslužit ćemo se na dijagramu klasa na slici 9.



Slika 9: Xamarin i ASP.NET - dijagram klasa

Dijagram pokazuje sve klase i sučelja vezana za komponentu Controller u cijelom rješenju. Pet klasa i jedno sučelje možemo podijeliti na dvije skupine, odnosno, na generičke klase koje se koriste u svakoj konkretnoj klasi i na konkretne klase. Prvoj skupini pripadaju sučelje IController.cs, Controller.cs (unutar Test projekta) i BaseController.cs (unutar Test.MVC projekta). Drugoj skupini pripadaju tri klase istog naziva, ItemsController.cs, po jedan u svakom od spomenuta tri projekta s tim da onaj u Test.Core projektu je apstraktna klasa da se izbjegnu kolizije s istoimenom klasom unutar Test.MVC.

Sučelje IController.cs definira samo jednu metodu, ChangeView(), koja služi za mijenjanje pogleda. Klase Controller.cs (Test) i BaseController.cs (Test.MVC) implementiraju tu metodu, implementacija u Xamarin dijelu je slična kao metoda na dijelu koda 4.4. Najveći problem ove implementacije je drugačija priroda navigacije unutar web aplikacije i unutar Xamarin aplikacije. U ASP.NET-u, način definiranja rute je takav da se doda nova metoda u potrebni Controller te ta metoda mora vraćati instancu neke klase koja implementira sučelje IActionResult.cs. Problem koji nastaje je taj što metoda ChangeView(), za ASP.NET treba imati IActionResult.cs kao povratni tip, a za Xamarin ta metoda ne treba nikakav povratni tip. Ovaj problem je riješen tako što implementacija te metode u Xamarinu ima taj povratni tip, ali ga samo ignorira i vraća null. Važno je još napomenuti da su ove tri klase realizirane koristeći Dependency Injection. Za sučelje IController, u Test projektu se injektira Xamarinov Controller.cs, dok u Test.MVC ASP.NET-ov BaseController.

Primjer konkretnog Controller-a je Controller za stavke. Spomenuto je već da su dodane tri klase ItemsController.cs u tri projekta. Klasa u Test.Core u projektu definira tri metode za tri pogleda, slično kao na dijelu koda 4.5. Ta klasa sadrži objekt od sučelja IController.cs koji je dobiven preko konstruktora te, u pravilu, na kraju svake metode poziva ChangeView() metodu. Što se tiče druge dvije klase ItemsController.cs, one nasljeđuju ItemsController.cs iz Test.Core projekta te prosljeđuju objekt sučelja IController.cs, koji su dobili injektiranjem ovisnosti, u konstruktor natklase.

Pored toga, klase ItemsController.cs u projektima Test i Test.MVC ne moraju imati nikakve metode, međutim, razvojni programer ima slobodu dodavati nove ili nadjačavati metode iz natklase po potrebi. ItemsController.cs iz projekta Test ne definira nijednu novu metodu niti nadjačava ijednu iz Test.Core, međutim ItemsController.cs iz Test.MVC-a je definirao jednu novu metodu i nadjačao jednu iz Test.Core projekta. Nova metoda je POST metoda koja služi za dodavanje nove stavke, a nadjačana je metoda ItemDetailPage() zbog različitosti modela pogleda. Pogled za detalje u Xamarinu prima za argument samo identifikator stavke i ViewModel je zadužen za dohvaćanje cijelog objekta, dok pogled za detalje u ASP.NET dijelu prima cijeli objekt stavke za argument.

Ovaj primjer je napravljen isključivo u svrhu prikazivanja mogućnosti povezivanja Xamarina i ASP.NET-a. ASP.NET je Open Souce s preko tisuću suradnika [21] te je zahtjevno prilagoditi Xamarin da odgovara svim mogućnostima koje se mogu iskoristiti u ASP.NET-u.

4.2.3. Zaključak

Iako je MVVM zadani uzorak za Xamarin, MVC je toliko raširen u različitim tehnologijama i programskim jezicima da je velik dio razvojnih programera, koji tek dolaze na Xamarin tehnologiju, upoznat s MVC, ali nije s MVVM uzorkom. Sa svoje 3 komponente, MVC se pokazao kao dovoljno jednostavan za projekte manjeg obujma, ali i dalje ostavlja slobodu za fragmentiranjem postojećih komponenti (konkretno Model-a) da se prilagodi i za kompliciranije projekte.

Uzor u implementaciji u radu je MVC uzorak u ASP.NET razvojnom okviru. Razlog zašto

je taj primjer uzor leži u tome da će se Xamarinom baviti najčešće razvojni programeri koji su već razvijali u .NET-u, ali samo za različitu platformu tako da se razvojni programer, koji je prethodno radio na ASP.NET-u, se ne mora prilagođavati na MVVM.

Implementacija je donijela nekoliko poboljšanja u odnosu na MVVM, to je pojednostavljen način registriranja ruta kao i promjena pogleda. Ta poboljšanja su donijela sa sobom negativnu stranu u vidu performansi što će se prikazati u poglavlju usporedbi uzoraka. Zbog korištenja refleksije, navigacija je postala nešto sporija u odnosu na MVVM primjer.

Na kraju je implementirana ideja o povezivanju Xamarina sa ASP.NET-om u MVC uzorku. Pokazano je da je u jednom rješenju moguće imati i web i Xamarin projekt što može uvelike skratiti utrošeno vrijeme razvojnim programerima za projekte koji zahtijevaju web i mobilnu verziju.

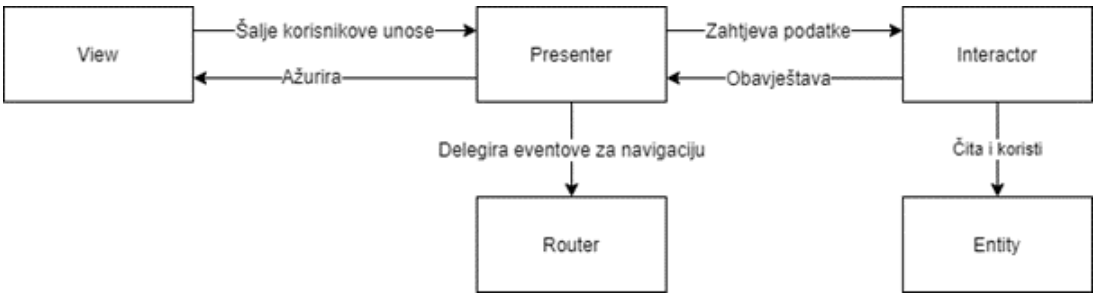
4.3. VIPER

Unutar domene uzoraka interaktivnih sustava najčešće se susreću kratice kao što su MVC, MVP i MVVM, međutim, u konkretnom pogledu za mobilne aplikacije, često svoje mjesto pronađe i kratica VIPER. VIPER je arhitekturalni uzorak dizajna kojeg susrećemo isključivo kad je u pitanju razvoj iOS aplikacija unutar programskog jezika Swift. Zahvaljujući tako uskoj domeni, ovaj uzorak je i dalje ostao relativno nepoznat.

VIPER se sastoji od čak pet dijelova:

- View – Prikazuje podatke na zaslonu koje je dobio od Presentera. (prikaz podataka korisniku i rukovanje korisnikovog unosa)
- Interactor – Sadrži poslovnu logiku aplikacije, koristi Entity klase. (dohvaćanje podataka, slanje podataka, pretvorba dobivenih podataka, obrada podataka, obrada korisnikovog unosa)
- Presenter – veže View, Router i Interactor, sadrži logiku za manipuliranje pogledima
- Entity – Sadrži klase u koje se zapisuju podaci, u većini ostalih uzoraka ovaj dio se naziva Model, ali ovdje je taj naziv otpao zbog zvučnosti kratice. (sadrži klase kao modele podataka)
- Router – Navigira između zaslona unutar aplikacije. [22]

Navedene dijelove i veze između dijelova se mogu vidjeti na slici 10.



Slika 10: VIPER [23]

Iz dijagrama vidimo da najveću funkciju ima Presenter koji služi kao ljepilo između Interactora, Viewa i Routera. Presenter od Interactora zahtjeva podatke pri čemu Interactor rješava dobavljanje podatka te poslovnu logiku (koristeći modele iz Entity-a) te te podatke prosljeđuje Viewu koji je zadužen isključivo za prikaz podataka te prihvaćanje korisničkih unosa. Korisnički unosi se šalju iz Viewa u Presenter koji obrađuje podatke te ih prosljeđuje Interactoru ili u slučaju da je korisnik zahtijevao promjenu pogleda, Presenter delegira navigaciju komponenti Router [22].

Iako VIPER i C# se jako rijetko susreću, ovaj uzorak je uključen u ovaj rad zbog svoje vezanosti s razvojem mobilnih aplikacija.

4.3.1. Implementacija

Već na prvu, zbog nedostatka početnih slova "MV" u akronimu, možemo razlikovati VIPER od ostalih "ustaljenih" uzoraka kao što su MVVM i MVC. Međutim, prilikom implementiranja ovog uzorka se mogu vidjeti sličnosti, posebno s MVC uzorkom. Čisto radi usporedbe, može se reći da se komponenta Model iz MVVM uzorka (samim time i iz MVC uzorka jer su iste) dijeli na komponente Interpreter (sadrži poslovnu logiku aplikacije) i Entity (sadrži klase kao modele podataka) dok komponenta ViewModel se dijeli na komponente Router (sadrži metode za navigaciju) i Presenter (sadrži sve isto što i ViewModel osim dijela za navigaciju). Jasno, ova usporedba nije u potpunosti točna, ali služi za dobivanje dovoljno dobrog početnog dojma kako će ovaj uzorak izgledati prilikom implementacije.

Kao što ovaj uzorak karakterizira veći broj komponenti od ranije obrađenih arhitektura, karakterizira ga i veći broj klasa što se najbolje može vidjeti na dijagramu klasa (slika 11).

Kao što je to bila praksa u ranijim primjerima, projekt je podijeljen na komponente uzorka uz dodatak ViewModel-a.

Komponente View i ViewModel su, po klasama, jednake kao u prethodnim primjerima. Komponenta Presenter sadrži jednu natklasu (BasePresenter.cs) i četiri potklase od kojih svaka je zadužena za jedan pogled te se na pogled veže pomoću ViewModel klasa.

Navedeno je da je komponenta Model iz MVC i MVVM primjera podijeljena na komponente Interactor i Entity što se jasno vidi na dijagramu klasa tako što, te dvije komponente sadrže iste klase kao Model u prethodnim primjerima. Komponenta entity sadrži klase Item.cs i Log.cs koje predstavljaju model podataka, a Interactor klase LogManager.cs, ItemInteractor.cs i LogInteractor.cs (zadnje dvije su, u usporedbi s MVVM i MVC uzorkom preimenovane sa *AppService.cs na *Interactor.cs).

Zadnja komponenta je komponenta Router koja sadrži sav kod vezan za navigiranje aplikacijom. Ovu komponentu sačinjavaju tri klase, RouteArgs.cs, Routes.cs i Router.cs i, bez obzira na veličinu aplikacije, ovoj komponenti nije potrebno dodavati nove klase. S komponentom Router komunicira samo klasa BasePresenter.cs.

U odnosu na ostale uzorke, VIPER ima veći broj komponenti i veći broj klasa što može za sobom povući jednu negativnu i jednu pozitivnu stvar. Navigacija projektnim rješenjem je otežana što, međutim, u projektima većeg obujma može dobro doći to što su klase entiteti fizički odvojeni od klasa koje sadrže poslovnu logiku te što se cijela logika oko navigacije nalazi u jednoj komponenti koja uvijek sadrži samo tri klase (Router.cs, Routes.cs i RouteArgs.cs).

Za registriranje ruta većinu posla obavlja klasa Routes.cs koja je implementirana kao Singleton koja u sebi sadrži listu ruta koje su nam potrebne u aplikaciji. Na dijelu koda 4.8 se može vidjeti konstruktor navedene klase.

```
1         private Routes()
2         {
3             RoutesList = new List<RouteArgs>();
4             RoutesList.AddRange(new List<RouteArgs>
5             {
```

```
6                   new RouteArgs
7                   {
8                       Name = nameof(ItemDetailPage),
9                       Type = typeof(ItemDetailPage),
10                      ModelType = typeof(ItemDetailModel)
11                  },
12                  new RouteArgs
13                  {
14                      Name = nameof(NewItemPage),
15                      Type = typeof(NewItemPage),
16                      ModelType = null
17                  },
18                  new RouteArgs
19                  {
20                      Name = nameof(LogsPage),
21                      Type = typeof(LogsPage),
22                      ModelType = null
23                  },
24              });
25          }
```

Kod 4.8: VIPER - konstruktor klase Routes.cs

Lista ruta je zapravo lista objekata klase RouteArgs.cs koja sadrži tri varijable, to su naziv (Name – naziv pogleda), tip (Type – klasa pogleda) koje, kao što je ranije spomenuto, su potrebne za registriranje pojedine rute. Pored toga, sadrži i varijablu ModelType pomoću koje možemo definirati koja je struktura modela koji želimo proslijediti pogledu.

Definirane su tri rute, za detalje stavke, za dodavanje nove stavke te za pogled s logovima. Pogled sa svim stavkama nije potrebno registrirati jer se nalazi u korijenskoj navigaciji aplikacije odnosno definiran je već u AppShell-u.

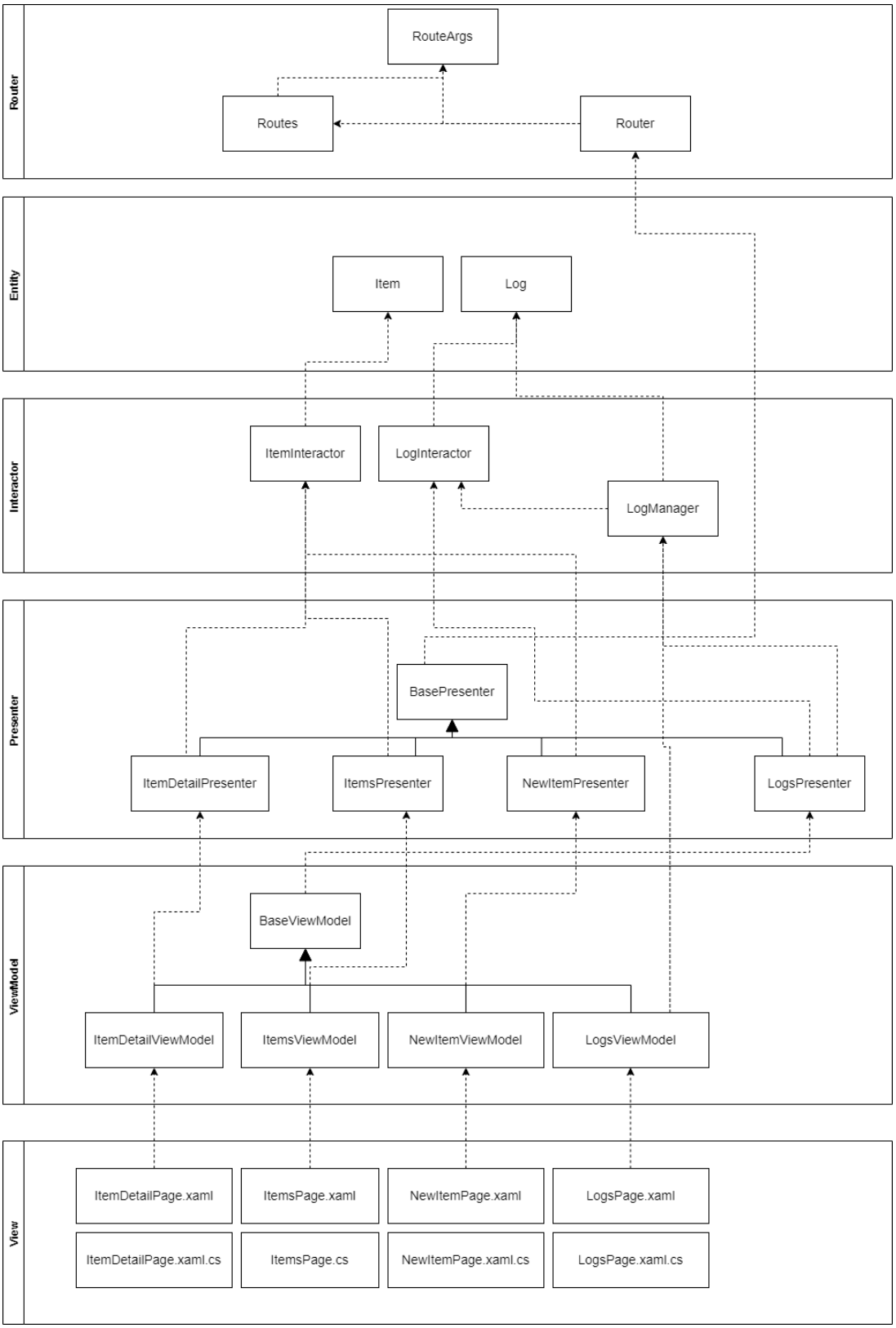
Što se tiče ModelType-a, definiran je samo za pogled s detaljima te klasa koja služi za to (ItemDetailModel.cs) se nalazi unutar mape Models sadrži samo jednu varijablu i to je ItemId. Pored toga, ta klasa nadjačava metodu ToString() tako što definira string upita.

Klasa Routes.cs sadrži još nekoliko metoda, jedna je za registriranje ruta koja iterira po listi svih ruta te registrira svaku. Također, sadrži dvije metode za dobivanje rute (točnije, dobivanje objekta klase RouteArgs.cs) po nazivu ili po tipu.

Već poznata poslovna logika za ove primjere je dobila izmjene u skladu s VIPER arhitekturom. Prva je, kao što je spomenuto u nekoliko navrata, odvajanje klasa koje sadrže poslovnu logiku (Interactor) od klasa koje služe kao modeli podataka (Entity).

Osim toga, klase ItemAppService.cs i LogAppService.cs. su preimenovane u ItemInteractor.cs, odnosno LogInteractor.cs kako bi bolje odgovarale opisu VIPER arhitekture, te navedene klase, po potrebi, sadržavaju referencu na BasePresenter.cs što omogućuje da se iz Interactor dijela obavještava Presenter.

Vezivno tkivo između cijele aplikacije predstavlja Presenter koji sadrži referencu na In-



Slika 11: VIPER - dijagram klasa

teractor, na Router te na ViewModel. Važno je napomenuti da postoji po jedna klasa Presenter za svaki pogled.

U ovom primjeru, Presenter je implementiran tako što je definirana klasa BasePresen-

ter.cs koja je natklasa svim Presenter-ima. Navedena klasa se može vidjeti na dijelu koda 4.9.

```
1         public class BasePresenter
2     {
3         public Router.Router Router { get; set; }
4
5         public BasePresenter()
6         {
7             Router = new Router.Router();
8         }
9
10        public async Task ChangeView<Tpage>(object model = null)
11        {
12            await Router.ChangeView(typeof(Tpage), model);
13        }
14
15        public async Task GoBack()
16        {
17            await Router.GoBack();
18        }
19    }
```

Kod 4.9: VIPER - klasa BasePresenter.cs

Kao što se može vidjeti, ova klasa je jednostavna u ovom primjeru, definira samo metode za navigaciju. Dvije metode koje implementira su za promjenu pogleda i za vraćanje na prethodni pogled. Konkretna Presenter klasa se može vidjeti na dijelu koda 4.10.

```
1         public class ItemsPresenter : BasePresenter
2     {
3         private ItemsViewModel _viewModel;
4         private ItemInteractor _itemInteractor;
5
6         public ItemsPresenter(ItemsViewModel viewModel)
7         {
8             _viewModel = viewModel;
9             _itemInteractor = new ItemInteractor();
10        }
11
12        public async Task<IEnumerable<Item>> GetItemsAsync()
13        {
14            var items = await _itemInteractor.GetItemsAsync();
15            return items;
16        }
17
18        public async void OnItemSelected(Item item)
19        {
20            if (item == null)
21                return;
```

```
22
23              await ChangeView<ItemDetailPage>(new ItemDetailModel { ItemId
                    = item.Id });
24          }
25
26          public async void OnAddItem(object obj)
27          {
28              await ChangeView<NewItemPage>();
29          }
30
31          public async void OnLogs(object obj)
32          {
33              LogManager.GetInstance().StartTime = DateTime.Now.Ticks;
34              await ChangeView<LogsPage>();
35          }
36      }
```

Kod 4.10: VIPER - klasa ItemsPresenter.cs

Za primjer konkretne klase Presenter odabrana je klasa ItemsPresenter.cs. U njoj se može pronaći referenca na ItemsInteractor.cs, na ItemsViewModel.cs (Presenter se instancira u ViewModel klasi pa se u konstruktoru nalazi argument preko kojeg ViewModel klasa prosljeđuje svoju instancu) te nekoliko metoda koje se mogu pronaći u komponentama ViewModel u uzorcima MVVM i MVC uz, jasno, nekoliko promjena koje su vezane uz navigaciju. Komponenta Presenter olakšava komponentu ViewModel.

Na dijelovima koda 4.9 i 4.10 se može vidjeti kako izgleda mijenjanje pogleda iz perspektive Presenter-a, potrebno je samo pozvati generičku metodu ChangeView() (nalazi se u BasePresenter.cs te poziva metodu istog naziva u Router-u) što znači da je obvezno definirati na koji pogled se želimo prebaciti te je opcionalan argument za model koji prosljeđujemo pogledu. Model je opcionalan u toj metodi, ali to ne znači da se neće pojaviti greška za vrijeme izvođenja ako smo pozvali pogled koji zahtjeva određeni model koji nismo proslijedili.

Što se tiče komponente Router, već smo spomenuli klase koje služe za registraciju ruta, međutim, pored toga, postoji i klasa Router.cs koja je zadužena za samo mijenjanje pogleda. Ta klasa se može vidjeti na dijelu koda 4.11.

```
1           public class Router
2       {
3           public async Task ChangeView(string name, object model)
4           {
5               var arg = Routes.GetInstance().GetRouteArgByName(name);
6               await CombinedChangeView(arg, model);
7           }
8           public async Task ChangeView(Type type, object model)
9           {
10              var arg = Routes.GetInstance().GetRouteArgByPage(type);
11              await CombinedChangeView(arg, model);
12          }
```

```
13
14          private async Task CombinedChangeView(RouteArgs arg, object model)
15          {
16              if (arg.ModelType == null)
17              {
18                  await Shell.Current.GoToAsync($"{arg.Name}");
19              }
20              else
21              {
22                  var type = Convert.ChangeType(model, arg.ModelType);
23                  await
                        Shell.Current.GoToAsync($"{arg.Name}?{type.ToString()}");
24              }
25          }
26          public async Task GoBack()
27          {
28              await Shell.Current.GoToAsync($"..");
29          }
30      }
```

Kod 4.11: VIPER - klasa Router.cs

Navedena klasa, praktično, nudi dvije opcije Presenter-u, to su navigacija na određeni pogled te vraćanje na prethodni pogled. Da bi se što više generaliziralo mijenjanje pogleda, napravljene su dvije ChangeView() metode koje primaju različite argumente ovisno o tome želimo li proslijediti klasu pogleda ili naziv pogleda. Pored toga postoji još jedna privatna metoda CombinedChangeView() koja sadrži dio koda koji je zajednički objema ChangeView() metodama.

Dio koji nije zajednički je dohvaćanje željenog objekta klase RouteArgs.cs iz singletona Routes.cs, dok, zajednički dio je mijenjanje pogleda preko statične metode Shell.current.GoToAsync() te, po potrebi prosljeđivanja modela pogledu (ako se isti zahtjeva prema RouteArgs.cs objektu koji smo definirali u Routes.cs klasi).

Za primjer ViewModel klase može se uzeti pripadna klasa za Items komponentu, odnosno ItemsViewModel.cs. Uloga ove klase je veza između pogleda i Presenter-a tako da sadrži varijable koje se koriste u pogledu te definira potrebne komande. Najviše o toj klasi će nam reći varijable koje sadrži i konstruktor koje možemo vidjeti na dijelu koda 4.12.

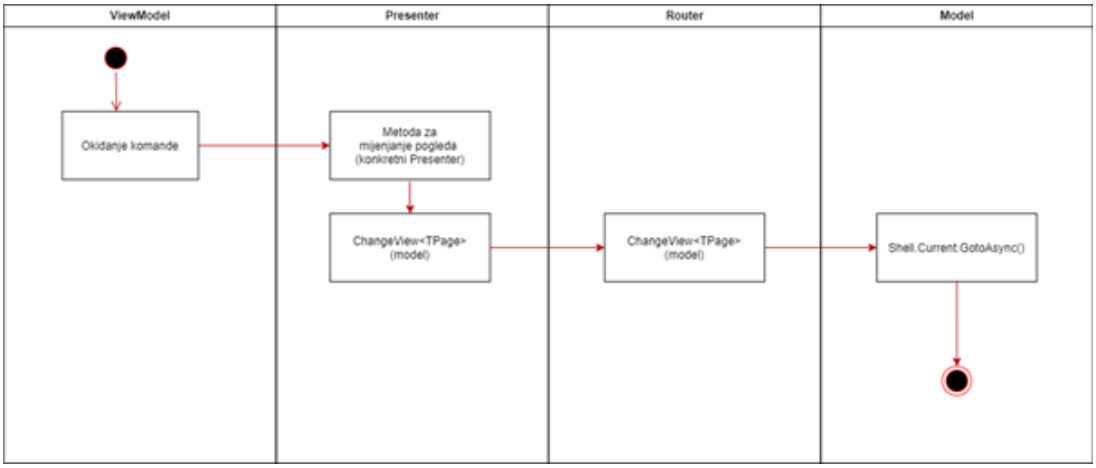
```
1           private Item _selectedItem;
2
3           public ObservableCollection<Item> Items { get; }
4           public Command LoadItemsCommand { get; }
5           public Command AddItemCommand { get; }
6           public Command LogsCommand { get; }
7           public Command<Item> ItemTapped { get; }
8           private ItemsPresenter _itemsPresenter;
9
10          public ItemsViewModel()
11          {
12              _itemsPresenter = new ItemsPresenter(this);
```

```
13
14
15              Title = "Browse";
16              Items = new ObservableCollection<Item>();
17              LoadItemsCommand = new Command(async () => await
                    ExecuteLoadItemsCommand());
18
19              ItemTapped = new Command<Item>(_itemsPresenter.OnItemSelected);
20
21              AddItemCommand = new Command(_itemsPresenter.OnAddItem);
22              LogsCommand = new Command(_itemsPresenter.OnLogs);
23          }
```

Kod 4.12: VIPER - klasa ItemsViewModel.cs

Na navedenoj slici možemo vidjeti sve varijable i komande koje sadrži navedena klasa. To su četiri komande koje se okidaju prilikom korisničkog unosa, pored toga tu su varijable, lista svih stavki koje se učitavaju u pogled, objekt jedne stavke koji predstavlja stavku koju je korisnik kliknuo te referenca na pripadni Presenter. U konstruktoru vidimo inicijalizaciju Presenter-a (prosljeđuje se vlastita instanca klase) te, za razliku od primjera u MVVM i MVC arhitekturi, komande se inicijaliziraju tako da se proslijedi metoda iz Presenter-a, a ne neka vlastita metoda. Ovaj ViewModel sadrži još metodu za dobivanje svih stavki te metodu OnAppearing() koja se okida prilikom učitavanja pogleda.

Na slici 12 se može vidjeti dijagram aktivnosti za navigaciju u primjeru.



Slika 12: VIPER - dijagram aktivnosti za navigaciju

Kao uvijek, mijenjanje pogleda počinje korisničkim unosom koji okida komandu u ViewModel klasi koja daljnju odgovornost prebacuje na Presenter-a čija uloga je i dalje jednostavna. Svaka metoda koja se nalazi u konkretnom Presenter-u označava na koji pogled se želi prebaciti te prosljeđuje taj pogled generičkoj metodi u BasePresenter.cs klasu koja poziva sličnu klasu u Router.cs. Konkretna metoda, po potrebi, prosljeđuje još model.

Router dohvaća željeni pogled te njegov model koji smo definirali u Routes.cs klasi, vrši pretvaranje objekta koji predstavlja model u objekt klase koji taj pogled zahtjeva (postoji

mogućnost javljanja greške prilikom izvođenja aplikacije ako se objekti ne podudaraju). Zatim Router poziva statičnu klasu Shell.current.GoToAsync() koja mijenja pogled.

Prilikom dodavanja nove navigacije razvojni programer mora kreirati komandu kao u svakom uzorku do sad, metodu u konkretnom Presenter-u, dok su ostale metode dio boiler-plate dijela koda. Jasno, potrebno je definirati i novu instancu klase RouteArgs.cs da bi Router mogao uspješno dobiti argumente za taj pogled. Možemo reći da mijenjanje pogleda u VIPER arhitekturi, po kompleksnosti, je ravan mijenjanju pogleda u MVVM arhitekturi, odnosno jednostavan koliko može biti.

4.3.2. Zaključak

Dodirna točka VIPER-a i razvojnog okruženja Xamarin, samim time i C#-a i .Net-a je samo razvoj mobilnih aplikacija. VIPER nudi veću razinu razdvojenosti komponenti od uzoraka MVVM i MVC, što znači da je prilagođen aplikacijama većeg obujma. Kad je u pitanju implementacija VIPER-a u Xamarinu, potrebno je dodati još jednu komponentu ViewModel čime se dolazi do broja od 6 komponenti u ovom arhitekturalnom uzorku.

Ovaj način implementacije, u odnosu na MVVM, je poboljšao probleme s navigacijom, ali je i dalje ostao problem sa zasebnim definiranje ruta, međutim, registriranju ruta je dodana i opcija definiranja modela koji se šalje pojedinom pogledu tako da je zasebno definiranje ruta dobilo veći smisao tako što otklanja potencijalne greške razvojnog programera da krivo definira model koji se šalje pogledu.

Komponenta koja najviše razlikuje ovaj uzorak od ostalih je svakako komponenta Presenter koja, u ovoj implementaciji, olakšava posao komponenti ViewModel. Međutim, zbog slične prirode Presenter-a i ViewModel-a, kod se samo podijeli između te dvije komponente tako da ViewModel-u ostane samo definiranje komandi koje slušaju unos korisnika, a sve ostalo se prebacuje na Presenter. Obzirom na sve to, VIPER nudi visoku razinu podjele poslova koja je nepotrebna u aplikacijama manjeg obujma, ali za one veće, nudi dobru alternativu MVVM-u i MVC-u.

4.4. Flux

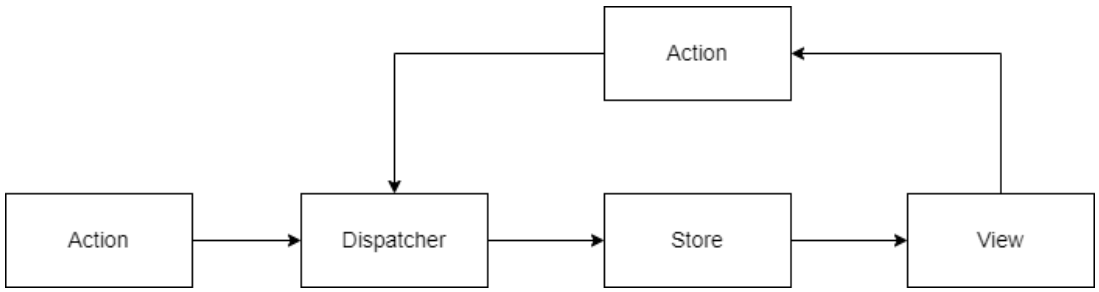
Flux je uzorak interaktivnih sustava koji je razvijen i dizajniran od strane Facebooka [24]. Razvijen je za potrebe React programskog okvira da bi riješio problem vezanja React komponenti (komponenta u ovom kontekstu se odnosi na klasu ili funkciju koja vraća dio korisničkog sučelja [25]).

React komponente se postavljaju hijerarhijski te problem koji je React imao jest jednosmjerni protok podataka između komponenti, odnosno moguće je poslati podatke samo s roditelja na dijete. Također, čak i ako razvojnom programeru nije potreban obrnuti protok, kod se može zakomplicirati ako podaci prolaze kroz previše roditelja.

Flux se sastoji od četiri komponente:

- Action - Pruža pomoćne metode koje kreiraju akciju iz argumenata metode i prosljeđuje je Dispatcher-u [26].
- Dispatcher - Središte aplikacije, upravlja svim protokom podatka u aplikaciji [26]. Prima akciju od komponente Action i šalje je svim pretplaćenim Store-ovima [24].
- Store - Prima akciju od Dispatcher-a, obrađuje poslovnu logiku i obavještava View o promjeni [24]. (dohvaćanje podataka, slanje podataka, pretvorba dobivenih podataka, obrada podataka)
- View - Prikazuje korisničko sučelje s podacima dobivenim od Store-a te može okidati akcije [24]. (prikaz podataka korisniku, rukovanje i obrada korisničkog unosa)

Na slici 13 možemo vidjeti dijagram koji pokazuje komponente i veze komponenti u uzorku Flux.



Slika 13: Flux [26]

Spomenuli smo maloprije kako je komponenta Dispatcher središte ovog uzorka što ne bismo rekli na prvi pogled na dijagram. Razlog tome je što je tok podataka u Flux uzorku jednosmjeran [26] i vidimo da Dispatcher se poziva isključivo iz komponente Action. Dispatcher akciju prosljeđuje na Store koji obrađuje podatke, tj. obavlja poslovnu logiku te, po potrebi obavještava View o promjeni podataka.

View reagira na informacije iz Store-a tako što mijenja pogled. Obzirom da View obrađuje korisnički unos, na temelju toga može okidati akcije čime se pokreće još jedan krug toka

informacija. Osim View-a, Action se može okinuti iz drugih izvora te je to razlog zašto je, na dijagramu, Action naveden dva puta.

Flux je jedan od prvih uzoraka dizajna koji su standardizirani na Reactu i s vremenom je izgubio na značaju. U prilog tome ide čak i to da Facebook u svojoj službenoj dokumentaciji za Flux preporučuje neke druge arhitekturalne uzorke kao što su Redux ili MobX [26].

Samim time što je dizajniran za programski okvir React, odnosno programski jezik JavaScript, ovaj uzorak ne definira neke dijelove koji su obvezni za programski jezik C# i platformi Xamarin. Tako da van spomenutih komponenti će biti potrebno definirati komponentu za navigaciju aplikacijom i komponentu koja će sadržavati klase kao modele podataka.

Flux je jedina arhitektura u ovom radu koja nije troslojna.

4.4.1. Implementacija

Zbog različite prirode Reacta i Xamarina, uzorak Flux se implementira na dosta različit način od svih dosadašnjih uzoraka, međutim, uz obilno korištenje injektiranja ovisnosti (engl. Dependency Injection), sve Flux komponente bilo je moguće implementirati i u Xamarinu bez previše razlika u odnosu na način na koji bi se to implementiralo u Reactu. Jasno, ova implementacija je zahtijevala da se dodaju i neke druge komponente tako da, kao i u dosadašnjim uzorcima, ubačene su i ViewModel klase. Pored toga nalaze se još komponente Model i Helpers.

Inspiracija za ovaj primjer je dobivena od GitHub korisnika pod korisničkim imenom SuavePirate koji je napravio predložak za Flux uzorak u Xamarinu [24]. Taj predložak koristi ima sličnu strukturu datoteka, ali se koristi drugi paket za Injektiranje ovisnosti tako da postoje mnoge razlike u implementaciji samih komponenti koje pripadaju Flux arhitekturi.

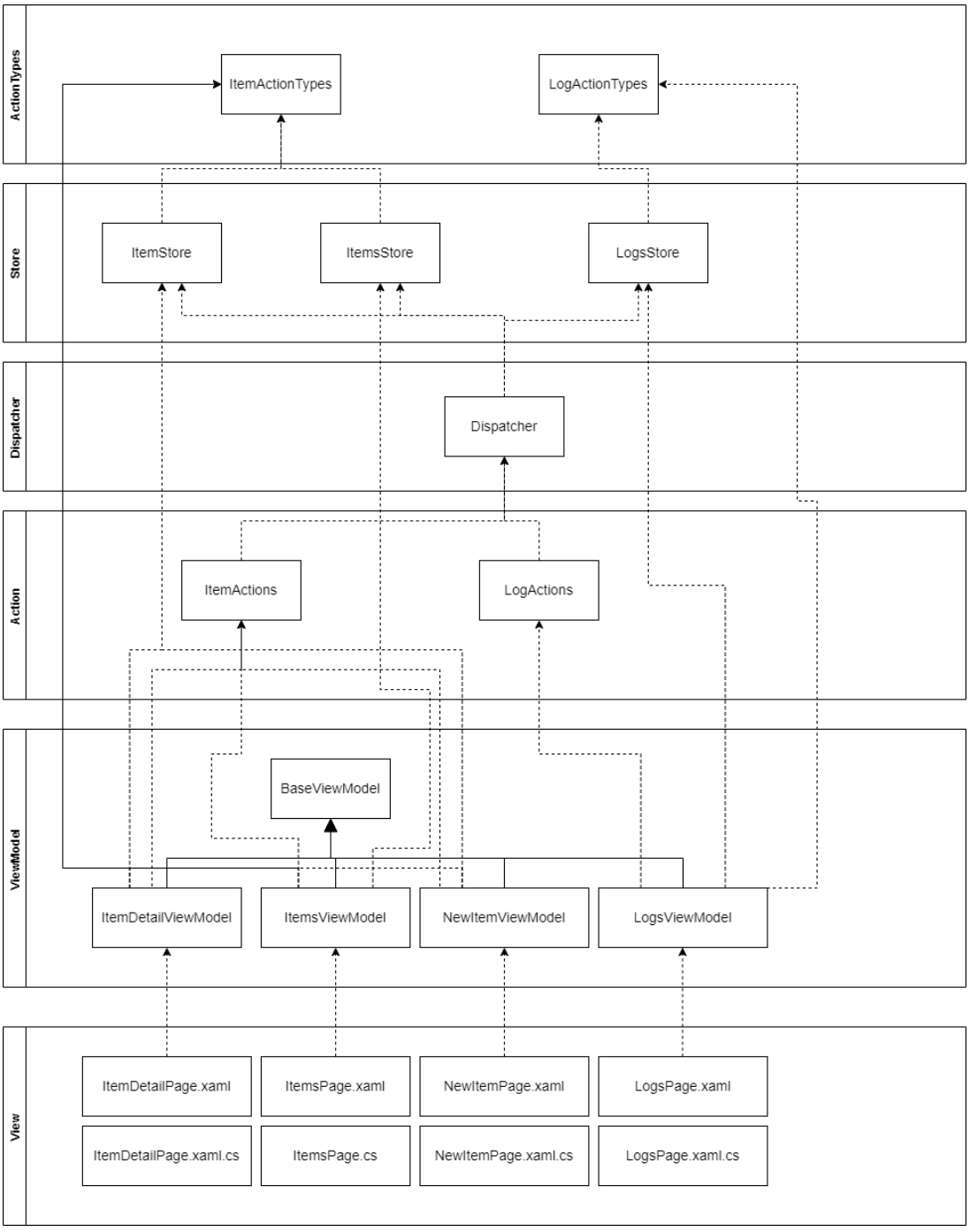
Kao i kod VIPER arhitekture, Implementacija Flux-a u Xamarinu se sastoji od velikog broja komponenti i klasa. Dijagram klasa se može vidjeti na slici 14.

Na dijagramu klasa odmah možemo prepoznati komponente specifične Flux-u, Action, ActionType, Store, Dispatcher i View. Osim navedenih komponenti, tu se nalazi još, već dobro poznata komponenta, ViewModel.

Iz dijagrama klasa su izostavljene dvije komponente, Model i Helper. Komponenta Model sadrži klase kao modele podataka (kao komponenta Entity u uzorku VIPER). Komponenta Helper je dodana da bi pružila "pomoć" ostalim klasama te sadrži klase LogManager.cs koja je ista kao u prethodnim primjerima te klase LogHelper.cs i ItemHelper.cs. Te dvije klase su statične i sadrže po dvije metode, jednu za dobivanje putanje datoteke na kojoj se spremaju stavke i zapisi, a druga je MapToItem(), odnosno MapToLog() koje služe za mapiranje linije teksta (koja se dobije iz datoteke) u objekt klase.

Komponente View i ViewModel su, po klasama, iste kao i u prethodnim primjerima s tim da su ViewModel klase prilagođene da dohvaćaju podatke iz Store-ova te, osim toga, sadrže referencu na komponente ActionType i Action pomoću kojih okidaju akcije.

Nakon što ViewModel okine akciju, komponenta Action prosljeđuje tu akciju Dispatcheru



Slika 14: Flux - dijagram klasa

koji, dalje, prosljeđuje tu akciju svim Store-ovima koji, po potrebi, reagiraju na tu akciju tako što izvrše dio koda.

Registriranje klasa za injektiranje ovisnosti se odvija u konstruktoru pripadne klase za pogled App. Taj konstruktor se može vidjeti na dijelu koda 4.13.

```
1         public App()
2         {
3             InitializeComponent();
4
```

```
5               DependencyService.Register<Dispatcher>();
6
7               DependencyService.Register<ItemStore>();
8               DependencyService.Register<ItemsStore>();
9               DependencyService.Register<LogsStore>();
10
11              DependencyService.Register<ItemActions>();
12              DependencyService.Register<LogActions>();
13
14              DependencyService.Get<LogActions>().DeleteLogs();
15
16              MainPage = new AppShell();
17          }
```

Kod 4.13: Flux - registriranje klasa za injektiranje ovisnosti

Za registraciju klasa koristi se klasa DependencyService koja je ugrađena u Xamarin.Forms. Ta ista klasa će se kasnije koristiti i za dobivanje klasa koje su registrirane. Potrebno je registrirati klasu Dispatcher.cs te sve klase u komponentama Actions i Stores. Ovakav način registriranja nije prilagođen za aplikacije velikog obujma, ali bilo bi jednostavno registrirati potrebne klase preko refleksije tako da razvojni programer ne mora se uvijek vraćati u klasu App.xaml.cs da bi registrirao neki Store ili Action koji je nanovo kreiran.

Flux arhitektura u svojoj definiciji ne definira ništa za navigaciju, tako da logika za registriranje ruta i promjenu pogleda je identična kao u MVVM uzorku, odnosno rute se registriraju u AppShell-u, a mijenjanje pogleda se pokreće iz ViewModel komponente koja poziva Shell.Current.GotoAsync() metodu. Ovaj način navigacije ostavlja slobodu da se, kod aplikacija većeg obujma, cijela logika za navigaciju odvoji u zasebnu komponentu.

Međutim, Flux je jasan kad je u pitanju poslovna logika. Svu poslovnu logiku sadrži komponenta Store čijim metodama View nikad ne pristupa izravno nego poziva potrebnu metodu u komponenti Action koja preko Dispatcher-a poziva Store.

View, Action i Store referenciraju klase koje se nalaze u mapi ActionTypes koje su vrlo jednostavne te ima najviše smisla da se one prve pokažu. Primjer ActionTypes-a se može vidjeti na dijelu koda 4.14 (ItemActionTypes.cs).

```
1       public class ItemActionTypes
2       {
3               public const string CREATE_ITEM = "create_item";
4
5               public const string GETALL_ITEMS = "getall_items";
6
7               public const string GET_ITEM = "get_item";
8       }
```

Kod 4.14: Flux - primjer ActionTypes-a (ItemActionTypes.cs)

Store prepoznaje o kojem se tipu akcije radi na temelju vrijednosti stringa koja je poslana iz ViewModel-a te, kao pomoć, komponenta ActionTypes služi samo tome da pruži kons-

tantnu vrijednost za akcije da bi se izbjegle greške u pisanju stringa da bi se označilo koja se akcija želi okinuti. Na slici vidimo da su definirane tri tipa akcije koja su vezana za stavke, to su CreateItem, GetAllItems i GetItem.

Svaka Store klasa mora nasljeđivati apstraktnu klasu BaseStore.cs koja se može vidjeti na dijelu koda 4.15.

```
1   public abstract class BaseStore<T>
2   {
3           public T Data { get; set; }
4
5
6           public abstract Task Invoke<TData>(string eventType, TData
                data);
7   }
```

Kod 4.15: Flux - BaseStore.cs

BaseStore.cs je implementirana kao generička klasa te sadrži varijablu Data koja je generičkog tipa koji je proslijeđen preko klase. Ovo nam daje mogućnost da svaki store sadrži podatke potrebne tom Store-u, tako da će ItemsStore.cs i LogsStore sadržavati listu stavki, odnosno zapisa dok će ItemStore.cs sadržavati jednu stavku. Također, BaseStore.cs definira i generičku metodu Invoke<>() koja se poziva iz Dispatcher-a te u konkretni Store preko nje vrši određene dijelove koda ako je pozvana akcija koja se tiče tog konkretnog Store-a.

Poslovna logika za stavke je u prijašnjim primjerima bila u cijelosti sadržana u jednoj klasi (ItemAppService.cs za MVVM i MVC i ItemInteractor.cs za VIPER), međutim, u ovom primjeru je razdvojena na ItemsStore.cs i ItemStore.cs. Store u svojoj prirodi sadrži objekt Data i razlog razdvajanja se može pronaći u tome što je podatak za ItemsPage lista stavki dok za ItemDetailPage jedna stavka što znači da ItemsStore.cs sadržava listu stavki te reagira na akciju "GETALL_ITEMS", a ItemStore.cs sadržava jednu stavku i reagira na akciju "GET_ITEM". Treća akcija vezanu za stavke, "CREATE_ITEM", je mogla biti dodana u obje klase (uzorak Flux dopušta da se čak, bez ikakvih problema, doda i u LogsStore.cs), ali je dodana u klasu ItemStore.cs zbog toga što ta klasa predstavlja samo jednu stavku.

Store-ove vezane za zapise nije bilo potrebno razdvajati pa će se klasa LogsStore.cs prikazati kao primjer konkretnog Store-a pošto reagira na najviše akcija. Ta klasa se može vidjeti na slici 4.16.

```
1   public class LogsStore : BaseStore<ObservableCollection<Log>>
2   {
3           public LogsStore()
4           {
5           Data = new ObservableCollection<Log>();
6           }
7
8           public override async Task Invoke<TData>(string eventType,
                TData data)
9           {
10                  switch (eventType)
```

```
11                          {
12                                  case LogActionTypes.GETALL_LOGS:
13                                  Data = new ObservableCollection<Log>();
14                                  GetLogsAsync();
15                                  break;
16                                  case LogActionTypes.DELETE_LOGS:
17                                  Data = new ObservableCollection<Log>();
18                                  DeleteLogs();
19                                  break;
20                                  case LogActionTypes.CREATE_LOG:
21                                  await AddLogAsync(data as Log);
22                                  Data = new ObservableCollection<Log>();
23                                  GetLogsAsync();
24                                  break;
25                          }
26                  }
```

Kod 4.16: Flux - LogsStore.cs

Možemo vidjeti da klasa LogsStore.cs nasljeđuje klasu BaseStore.cs kojoj kao generičku klasu prosljeđuje listu (ne običnu listu nego ObservableCollection) zapisa. Unutar klase vidimo nadjačavanje klase Invoke<>() koja za argumente prima varijablu eventType (to je vrijednost tipa akcije koji se može vidjeti na dijelu koda 4.14) i varijablu data koja je generičkog tipa. Unutar metode se nalazi programski konstrukt koji se ne viđa tako često, no za ovu potrebu je idealan. Taj programski konstrukt je switch. Unutar switch-a vidimo koje se linije koda okidaju za svaku akciju koja je pozvana. Tako da za akciju "GETALL_LOGS" se Data nanovo inicijalizira te se poziva metoda GetLogsAsync() koja popunjava varijablu Data. Za akciju "DELETE_LOGS" se Data nanovo inicijalizira te se poziva metoda DeleteLogs() dok za akciju "CREATE_LOG" se poziva metoda AddLogAsync() te se nakon toga ponovo učitavaju svi zapisi. Klasa LogsStore.cs sadrži još tri, već spomenute, metode, to su GetLogsAsync(), DeleteLogs() i AddLogAsync().

U ovom konkretnom pitanju se argument data se koristi samo za dodavanje novog zapisa. U tom slučaju ova varijabla je tipa Log.cs. Za dobivanje jasnije slike o ovom argumentu spomenut ćemo da se koristi i u ItemStore.cs te preko tog argumenta prosljeđujemo string koji predstavlja Id stavke koji želimo dohvatiti za akciju "GET_ITEM" ili pak objekt klase Item kada pozivamo akciju "CREATE_ITEM".

Metoda Invoke() je potpuno drugačije prirode od svih dosadašnjih metoda u svim primjerima gdje se izbjegava pozivanje jedne metode za različite akcije. Unatoč tom nedostatku, zbog karakterističnosti uzorka Flux, ova metoda nudi prednost da ViewModel može samo pozvati jednu akciju i ne brinuti se o nastavku rada aplikacije.

Komponenta Action u ovom Flux uzorku je jednostavna, njezina uloga je pružanje metoda koje će ViewModel pozvati. U ovom primjeru imamo dvije konkretne klase unutar ove komponente, ItemActions.cs i LogActions.cs. Konkretnu klasu ItemActions.cs možemo vidjeti na dijelu koda 4.17 (ItemActions.cs).

```
1         public class ItemActions
```

```
2       {
3               public void GetAllItems()
4               {
5               DependencyService.Get<Dispatcher>().Invoke<object>
6                       (ItemActionTypes.GETALL_ITEMS, null);
7               }
8
9               public void CreateItem(Item item)
10              {
11              DependencyService.Get<Dispatcher>().Invoke<Item>
12                      (ItemActionTypes.CREATE_ITEM, item);
13              }
14
15              public void GetItem(string id)
16              {
17              DependencyService.Get<Dispatcher>().Invoke<string>
18                      (ItemActionTypes.GET_ITEM, id);
19              }
20      }
```

Kod 4.17: Flux - konkretna klasa komponente Action (ItemActions.cs)

Ako se prisjetimo klase ItemActionTypes.cs (kod 4.14), znamo da smo definirali tri akcije koje su vezane uz stavke tako da klasa ItemActions.cs kreira samo metode za te tri akcije. Metoda GetAllItems() ne prima nikakav argument, dok metode CreateItem() i GetItem() primaju po jedan argument tipa Item.cs za CreateItem() i tipa string za GetItem(). Taj argument preko Dispatcher-a završi kao argument data u klasi Invoke() unutar nekog konkretnog Store-a.

Svaka metoda preko klase DependencyService.cs dohvaća Dispatcher.cs klasu te poziva njenu metodu Invoke<>(). Kao argumente prosljeđuje tip akcije koji je vezan uz tu metodu i, po potrebi, objekt koji je dobiven preko argumenta.

Klasa Dispatcher.cs sadrži samo jednu metodu, to je već spomenuta Invoke<>() koju možemo vidjeti na dijelu koda 4.18.

```
1       public void Invoke<TData>(string eventType, TData data)
2       {
3           _itemStore = DependencyService.Get<ItemStore>();
4           _itemsStore = DependencyService.Get<ItemsStore>();
5           _logsStore = DependencyService.Get<LogsStore>();
6
7           _itemStore.Invoke<TData>(eventType, data);
8           _itemsStore.Invoke<TData>(eventType, data);
9           _logsStore.Invoke<TData>(eventType, data);
10      }
```

Kod 4.18: Flux - metoda Invoke<>() u klasi Dispatcher.cs

Ova metoda ima jednu zadaću, a to je obavijestiti sve Store-ove o okidanju akcije. To radi na način da dohvaća sve storeove preko klase DependencyService.cs te u svakom Store-u poziva metodu Invoke<>() (ta metoda se nalazi na dijelu koda 4.16). Može se primijetiti da

je svaki konkretni Store potrebno zasebno dohvatiti i pozvati njegovu metodu što je svakako nedostatak ako bi u aplikaciji bilo previše Store-ova, ali taj problem bi se mogao jednostavno riješiti pomoću refleksije.

U teoretskom dijelu ovog poglavlja je spomenuto kako Dispatchera ne zanima koji Store reagira na koju akciju, nego šalje istu akciju svim Store-ovima, to možemo vidjeti i na ovom primjeru. Ako je pozvana, naprimjer, akcija za dodavanje nove stavke, dispatcher tu akciju prosljeđuje na sva tri Store-a, bez obzira što će na tu akciju reagirati samo ItemStore.cs. Ovakav način oduzima malo više resursa nego što bi inače bilo (vremenski ne bi trebalo oduzet nimalo vremena jer su metode asinkrone), ali daje mogućnost da se ista akcija dva puta.

Komponenta View je ista kao u prethodnim primjerima, dok je komponenta ViewModel doživjela sitne promjene u odnosu na MVVM. Promjene su vezane uz pozivanje akcija te uz dohvaćanje podataka iz Store-a. Svaka ViewModel klasa sadrži objekte klasa komponenti Action i Store. Referenca na Store služi da se može dohvatiti varijabla Data unutar Store-a te obvezna je ako se sadržaj na pripadnom pogledu dinamički učitava. Tako da ItemsViewModel.cs sadrži referencu na ItemsStore.cs, ItemDetailViewModel.cs na ItemStore.cs, LogsViewModel.cs na LogsStore.cs, a NewItemViewModel.cs ne sadrži nikakvu referencu na Store. Ono što svi ViewModel-i sadrži jeste referenca na neku klasu iz komponente Action. Na dijelu koda 4.19 možemo vidjeti kako NewItemViewModel.cs poziva akciju za kreiranje nove stavke.

```
1         private async void OnSave()
2         {
3             Item newItem = new Item()
4             {
5                 Id = Guid.NewGuid().ToString(),
6                 Text = Text,
7                 Description = Description,
8                 Price = Price
9             };
10
11            _itemActions.CreateItem(newItem);
12
13            await Shell.Current.GoToAsync("..");
14        }
```

Kod 4.19: Flux - pozivanje akcije iz ViewModel-a

Metoda na slici se poziva prilikom klika na Save gumb te zadaća joj je kreirati objekt klase Item.cs, taj objekt proslijediti kao argument metodi CreateItem() u klasi ItemActions.cs. Na kraju metoda, vraća pogled na onaj prethodni (u ovom slučaju to je ItemsPage).

Kao što je već spomenuto, navigacija je identična onoj u MVVM modelu tako da je dijagram aktivnosti isti kao na slici 5.

4.4.2. Zaključak

Flux je, s komponentom Store, donio potpuno drugačiji koncept na stol od prijašnjih uzoraka. Flux nije troslojna arhitektura kao MVVM, MVC i VIPER.

Način na koji je posložen protok podataka nakon što ViewModel pozove neku akciju zahtjeva dodavanje novog koda unutar nekoliko klasa. Prvo je potrebno definirati akciju u ActionTypes, zatim je definirati u Actions i na kraju u Invoke<>() metodi se tek može pozvati metoda koja zapravo obavlja poslovnu logiku. Unatoč dodavanju novog koda u više klasa, ovaj način nam donosi prednost da se ViewModel nikad ne treba brinuti o izvršavanju akcije, čak ne treba čekati povratnu vrijednost metoda koje poziva. Okidanje akcije će, uglavnom, promijeniti određeni podatak u određenom Store-u te, obzirom da referencira Store, ViewModel će moći ažurirati pogled po potrebi.

Na kraju vrijedi napomenuti da je inicijalna namjena Fluxa bila za različitu platformu, različit programski jezik i različit programski okvir od Xamarina. Neki koncepti koji su predstavljeni sa Fluxom se mogu iskoristiti, kao naprimjer, da komponenta Store sadržava podatke, a ne ViewModel, ali već u ovom jednostavnom primjeru se dolazi do pitanja kako će ti podaci biti strukturirani u jednom ili više različitih Store-ova.

4.5. Usporedba

Nakon što su predstavljena četiri arhitekturalna uzorka dizajna, u ovom poglavlju će se usporediti svi obrađeni uzorci.

Prilikom postavljanja aplikacije, razvojni programer ili arhitekt sustava uvijek mora donijeti odluku želi li brže izvođenje aplikacije ili jednostavniji (i brži) razvoj aplikacije. Da bi predstavili sve prednosti i nedostatke uzoraka, u narednim poglavljima se mogu pronaći rezultati mjerenja performansi pojedinog primjera i različite analize koda.

4.5.1. Vrijeme učitavanja pogleda sa zapisima

Prvo će se prikazati brzine izvršavanja pojedinih uzoraka. Mjerenje je išlo tako što se učita aplikacija, zatim deset puta bi se učitao pogled sa zapisima pa bi se uzeo prosjek vremena koji bi bio potreban za učitavanje tog pogleda. To sve se ponavlja pet puta (aplikacija se ugasi te ponovo upali da bi se zapisi izbrisali). Okidač za početak mjerenja je korisnički zahtjev za pogledom sa zapisima, a kraj je učitavanje svih zapisa u memoriju. Izmjereno vrijeme uključuje navigaciju i dohvaćanje podataka.

Sva vremena uzoraka se mogu vidjeti na tablici 1.

Tablica 1: Prosječno vrijeme učitavanja pogleda sa zapisima

Uzorak	1.	2.	3.	4.	5.	Prosjek
MVVM	8,770ms	9,471ms	8,919ms	8,489ms	8,692ms	8,868ms
MVC 1	9,901ms	10,691ms	10,354ms	11,255ms	8,055ms	10,051ms
MVC 2	26,589ms	27,949ms	30,567ms	28,255ms	28,060ms	28,284ms
VIPER	10,712ms	10,768ms	9,000ms	9,084ms	9,738ms	9,860ms
Flux	9,410ms	9,935ms	8,684ms	8,133ms	9,965ms	9,225ms

Od obrađenih primjera, najbrži je uzorak MVVM što je, može se reći, očekivano zbog najjednostavnije navigacije te najmanje koraka dohvaćanja podataka.

Odmah iza MVVM-a se nalazi komponenta Flux za koju je rečeno da ima isti način navigacije kao i MVVM i zaključak je da dohvaćanje podataka iz Store-a je sporije od načina dohvaćanja podataka iz modela kod MVVM uzorka.

Na trećem mjestu se nalazi uzorak VIPER. Razlog zašto je VIPER sporiji od prethodna dva uzorka leži u tome što ima kompliciraniji način navigacije gdje se, umjesto pozivanja dvije metode u uzorcima MVVM i Flux, poziva pet metoda. Sporije vrijeme ima i svoju pozitivnu stranu, a to je ta da je lakše registrirati nove rute (potrebno je samo dodati novi objekt u klasu Routes.cs). Također, pozitivna je stvar što ova implementacija VIPER-a validira model proslijeđen pogledu čime se izbjegavaju moguće greške prilikom izvođenja.

Od svih obrađenih uzoraka, najsporiji je MVC. U tablici se mogu vidjeti dva retka za

MVC (MVC 1 i MVC 2) od kojih prvi prikazuje učitavanje stranice sa zapisima gdje se zapisi učitavaju u memoriju u ViewModel-u, dok u drugom se zapisi učitavaju u Controller-u te preko upita prosljeđuju ViewModel-u koji deserijalizira vrijednost upita. Ovaj drugi način je zapravo jedini način koji je značajno sporiji od ostalih uzoraka, međutim, ovaj primjer je nezahvalan jer se preko upita prosljeđuje cijela lista zapisa. Prosljeđivanje podataka preko upita ima najviše smisla kad je potrebno proslijediti neki objekt (npr. objekt stavke prilikom prelaska na pogled za ažuriranje stavke), a ne listu.

Dakle, za usporedbu ovog načina implementacije MVC-a s ostalim uzorcima, gledat ćemo redak MVC 1 koji je i dalje sporiji, ali ne mnogo. Razlog zašto je sporiji je u tome što MVC, prilikom navigacije koristi zahtjevnu refleksiju u dva navrata, prvo da se iz ViewModel-a dohvati metoda Controller-a na temelju naziva klase i metode te nakon toga Controller mijenja pogled, također, koristeći refleksiju. Duže vrijeme učitavanja je posljedica pojednostavljivanja navigacije za razvojnog programera te u ovom uzorku, za navigaciju je potrebno samo dodati novu Controller klasu s metodom koja predstavlja jedan pogled (ili dodati novu metodu u postojeći Controller).

Još jedna nezanemariva stvar za napomenut jest ta da u uzorcima VIPER i MVC, razvojni programer ne mora ručno definirati upit prilikom navigacije što mu olakšava posao te se time izbjegavaju moguće greške.

Na temelju ovih rezultata se može zaključiti da je brzina izvođenja proporcionalna težini definiranja navigacije te je na razvojnom programeru odluka želi li brže izvođenje ili jednostavniju navigaciju.

4.5.2. Broj klasa

U ovom poglavlju će se prikazati jednostavna metrika broja klasa u rješenju pojedinog uzorka. Da bi dobili punu sliku, izdvojit će se broj klasa koji su potrebne za postavljanje rješenja (boilerplate kod) i broj klasa potrebnih za definiranje dijela za zapise i dijela za stavke. Broj klasa u dijelu za stavke se može promatrati kao broj klasa potrebnih za dodavanje novog modula obzirom da dio za stavke definira sve potrebne klase i poglede za kreiranje, ažuriranje, čitanje i brisanje stavke. Bitno je napomenuti da se pogled se računa kao jedna klasa zbog svoje pripadne klase. Iz broja klasa su izostavljeni pogledi App i AppShell obzirom da dolaze odmah kao predložak prilikom kreiranja Xamarin projekta.

Broj klasa svakog uzorka se može vidjeti na tablici 2.

Kad uzmemo u obzir samo ukupan broj klasa u rješenju, MVVM je se pokazao kao uzorak s najmanje klasa, a VIPER i Flux kao uzorci s najviše klasa. Međutim ovaj podatak dosta sakriva obzirom da svaka klasa u MVVM-u se sastoji od velikog broja varijabli i metoda dok, naprimjer, uzorak Flux sadrži klase kao što su klase unutar komponenti ActionType i Action koje sadrže svega nekoliko linija koda. MVC sadrži tri klase više od MVVM-a i to su klase koje pripadaju komponenti Controller. Možemo reći da veći broj klasa se kompenzira jednostavnijim registriranjem ruta i jednostavnijom navigacijom.

Idući stupac u tablici je broj boilerplate klasa te bi ovaj podatak trebao igrati najmanju

Tablica 2: Broj klasa

Uzorak	Ukupan broj klasa u rješenju	Broj boilerplate klasa	Broj klasa za zapise	Broj klasa za stavke
MVVM	14	1	5	8
MVC	17	2	6	9
VIPER	23	5	6	12
Flux	23	3	8	12

ulogu u odabiru uzorka obzirom da se te klase definiraju samo jednom na početku razvoja. Većina boilerplate klasa pripada skupini natklasa u nekoj od komponenti tako da svaki uzorak ima klasu BaseViewModel.cs (to je jedina boilerplate klasa u MVVM uzorku), MVC ima još klasu Controller.cs, VIPER ima klasu BasePresenter.cs, dok Flux na sličan način ima klasu BaseStore.cs. Van navedenih klasa, Uzorak VIPER ima tri klase u komponenti Router te Flux ima klasu Dispatcher.cs.

Spomenuto je da podatak u broju boilerplate klasa treba igrati najmanju ulogu u odabiru uzorka, međutim, ako je potrebno napraviti jednostavnu aplikaciju bez mogućnosti širenja, kao najbolji je se pokazao MVVM zbog samo jedne boilerplate klase koja dolazi u predlošku prilikom kreiranja Xamarin projekta tako da razvojni programer se odmah može posvetiti razvoju modula potrebnih za aplikaciju.

Obzirom da se boilerplate klase pišu samo jednom, a klase za zapise su dosta specifične, može se reći da je broj klasa za stavke najvažniji podatak u ovoj tablici obzirom da predstavlja broj klasa potrebnih za definiranje cijelog modula. Tu je se MVVM pokazao kao najjednostavniji, MVC ima samo jednu klasu više, a to je konkretni Controller. Znatno veći broj klasa imaju uzorci VIPER i Flux zbog komponente Presenter u slučaju VIPER-a, odnosno zbog komponenti Action i ActionTypes u slučaju Flux-a.

Iako Flux, uz VIPER, ima najveći broj klasa potrebnih za novih modul, može se reći da se to kompenzira tako što su nove klase jednostavne, tj. nemaju velik broj linija koda.

Podaci o broju klasa otkrivaju da je MVVM uzorak s najmanjim brojem klasa u svakom slučaju, ali sakrivaju neke druge stvari kao što je opterećenost ViewModel klasa ili kompliciranost navigacije. U idućem poglavlju će se detaljnije analizirati kod da bi saznali koliko su koje klase opterećene u ovim uzorcima.

4.5.3. Analiza koda

Visual Studio nudi alat za analizu koda gdje je moguće spoznati razne podatke o rješenju kao i o pojedinoj komponenti koji mogu koristiti za detaljniju analizu koda.

Obzirom na različitost komponenti uzoraka, metrika je podijeljena na nekoliko dijelova te na tablici 3 se mogu vidjeti svi uzorci s ukupnim brojem izvršivih linija koda i referenci na

druge klase.

Tablica 3: Broj linija koda i referenci na druge klase - ukupno

Uzorak	Broj izvršivih linija koda	Broj referenci na druge klase
MVVM	199	59
MVC	271	72
VIPER	245	68
Flux	241	71

Na tablici se vidi da, i u ovoj metrici, MVVM se pokazuje kao najbolji uzorak s najmanjim brojem izvršivih linija koda i s najmanjim brojem referenci na druge klase. Međutim, potrebno je naglasiti da je ovo projekt malog obujma gdje će se, kao najbolji, pokazati onaj uzorak koji ima najmanje boilerplate koda. MVC koji ima najveći broj linija koda i referenci na druge klase, će se u većim projektima pokazati kao najbolji jer, trenutnu razliku od 72 linije u odnosu na MVVM čine linije koda koje su u boilerplate dijelu koda te se ta razlika neće povećavati dodavanjem novih modula.

Spomenuto je da zbog različitosti komponenti različitih uzoraka, teško je uspoređivati pojedinačne dijelove rješenja, ali je moguće nekoliko komponenti izdvojiti. Tako da u svakom uzorku se mogu grupirati komponente View, ViewModel, kao i komponente za poslovnu logiku. Na tablici 4 se može vidjeti broj linija izvršivog koda i referenci na druge klase za komponentu View.

Tablica 4: Broj linija koda i referenci na druge klase - View

Uzorak	Broj izvršivih linija koda	Broj referenci na druge klase
MVVM	33	15
MVC	40	16
VIPER	36	15
Flux	44	16

Na tablici se vidi da svi uzorci dijele sličan broj linija koda i referenci na druge klase. Za taj broj se može reći da je zanemariv jer komponenta ViewModel uzima svu odgovornost od klasa koje se nalaze u komponenti View.

Na tablici 5 se može vidjeti broj izvršivih linija koda i broj referenci na druge klase u komponenti ViewModel.

Komponenta ViewModel, također, u svim uzorcima prikazuje sličnu metriku, ali, opet može se reći da ta metrika skriva podatke poput toga da, MVC, VIPER i Flux uključuju veći broj linija koda za boilerplate klasu BaseViewModel.cs te se, širenjem projekta, ta tri uzorka manje

Tablica 5: Broj linija koda i referenci na druge klase - ViewModel

Uzorak	Broj izvršivih linija koda	Broj referenci na druge klase
MVVM	95	30
MVC	96	35
VIPER	94	31
Flux	108	34

opterećuju ovu komponentu od uzorka MVVM. Također, u MVVM poglavlju je spomenuto da klasa BaseViewModel.cs sadrži referencu na AppService klase, umjesto da svaka konkretna ViewModel klasa sadrži reference na klase koje su joj potrebne čime bi se značajno povećao broj referenci na druge klase, posebno kod projekata većeg obujma.

Iduća usporedba za broj linija koda i referenci na druge klase se odnosi na poslovnu logiku aplikacije se može vidjeti na tablici 6.

Tablica 6: Broj linija koda i referenci na druge klase - poslovna logika

Uzorak	Uključene komponente	Broj izvršivih linija koda	Broj referenci na druge klase
MVVM	Model	56	19
MVC	Model	56	19
VIPER	Interactor, Entity	56	19
Flux	Store, ActionType, Helper, Action	51	30

Na tablici se može vidjeti da su uzorci MVVM i MVC identični zbog toga što je komponenta Model identična u oba uzorka. Poslovna logika u uzorku VIPER je podijeljana na komponente Interactor i Entity pokazuje isti broj linija i referenci kao MVVM i MVC. Uzorak Flux sadrži je podijeljen na dosta više komponenti i to za posljedicu ima nešto manji broj linija, ali zato dosta veći broj referenci na druge klase.

U ove tri usporedbe nisu uključene neke specifične komponente koje imaju neki uzorci i na tablici 7 se mogu vidjeti te komponente.

Uzorci MVVM i Flux nemaju dodatni komponenti koje nisu uključene u prethodne usporedbe, dok MVC i VIPER imaju komponente Controller, odnosno Presenter. Komponenta Controller ima relativno visok broj izvršivih linija koda i broj referenci, ali oba broja najviše otpadaju na klasu Controller.cs za koju je spomenuto da je dio boilerplate koda te, širenjem aplikacije, ovi brojevi se neće previše povećavati. Komponenta Presenter ima dosta manji broj linija i referenci, međutim većina tih linija i referenci nisu dio boilerplate koda te se, kod većih projekata, VIPER pokazati kao uzorak s najvećim brojem linija i referenci u odnosu na ostale

Tablica 7: Broj linija koda i referenci na druge klase - ostale komponente

Uzorak	Uključene komponente	Broj izvršivih linija koda	Broj referenci na druge klase
MVVM	/	0	0
MVC	Controller	43	22
VIPER	Presenter	25	17
Flux	/	0	0

uzorke.

4.6. Zaključak

Nakon što su u prethodnim poglavljima predstavljena i uspoređena četiri uzorka, može se zaključiti da svaki od uzoraka, jasno, ima svoje prednosti i nedostatke te je na razvojnom programeru odluka koji uzorak najviše odgovara aplikaciju koju planira razvijati.

Primjeri su jednostavni bez puno funkcionalnosti i očekivano je se kao najbolji pokazao najjednostavniji uzorak MVVM ponajviše zbog toga što su i ostali uzorci morali imati komponentu ViewModel zbog vezanosti te komponente uz Xamarin mogućnosti.

Unatoč tome, MVVM je, u nekim stvarima, već kod ove jednostavne aplikacije pokazao svoje nedostatke u vidu registriranja ruta i navigacije te zbog velike opterećenosti klasa koje pripadaju ViewModel komponenti. Registriranje ruta i navigacija se mogu poboljšati na neki drugi način tako da bi ti nedostatci nestali na projektima većeg obujma, međutim opterećenost ViewModel klasa se ne može riješiti.

Ostali uzorci manje opterećuju komponentu ViewModel, ali to sa sobom snosi posljedicu dodavanja drugih komponenti pa se, u svakom slučaju, taj nedostatak izbalansira.

Uzorak MVC nudi veliku prednost sa svojom komponentom Controller koja značajno olakšava navigaciju te pruža mogućnost učitavanja i obrade podataka iz komponente Model čime se olakšava komponenta ViewModel.

Uzorak Flux je, također, pokazao neke prednosti zbog spremanja podataka u Store-u i zbog svoje prirode da više Store-ova može reagirati na jednu akciju. Iako Flux ne rješava problem registriranja ruta i navigacije, taj problem, kao i u slučaju MVVM-a, može riješiti na neki način.

Uzorak VIPER nudi neke svoje prednosti, ali i veliki nedostatak postojanja komponente Presenter koja se u mnogim svojim ulogama poklapa s komponentom ViewModel koja je nužna za Xamarin projekt. Tako da, iako način navigacije pomoću komponente Router i izdvojenost poslovne logike od klasa koje služe kao modeli podataka predstavljaju prednosti ovog uzorka, te prednosti nestaju u usporedbi s MVVM-om i posebno MVC-om obzirom da MVC, također, ima jednostavan način registriranja ruta i navigacije te komponenta Model nudi fleksibilnost da se

poslovna logika i klase koje služe kao modeli podataka fizički razdvoje ako razvojni programer odluči da mu je to potrebno.

Zbog svega navedenog, u praktičnom primjeru će se koristiti principi iz uzoraka MVVM, MVC i Flux. Praktično rješenje sadrži komponente Model, ViewModel i View koje odgovaraju MVVM uzorku te će te komponente sadržavati najveći dio koda. Osim tih komponenti, iskorištena je komponenta Controller iz MVC uzorka što pruža jednostavniju navigaciju i fleksibilnost učitavanja podataka u Controller te prosljeđivanja preko upita. Također tu su komponente iz uzorka Flux, ActionType, Action i Store koje su zbog svoje globalne prisutnosti iskorištene za prikaz pogleda za navigaciju što će se detaljnije objasniti u narednim poglavljima.

5. Praktičan primjer

Da bi bolje shvatili sličnosti i razlike, prednosti i nedostatke obrađenih arhitekturalnih uzoraka, izrađen je praktičan primjer koji će se prikazati u ovome poglavlju. Naziv aplikacije je "Faksistent" i služi za kontrolu ostvarenih bodova korisnika tijekom semestra.

Source kod praktičnog primjera se može pronaći na poveznicama:

- https://github.com/bayo04/FaksistentApi - Serverska aplikacija
- https://github.com/bayo04/FaksistentX - Xamarin aplikacija

U narednim poglavljima će se dati uvid u svrhu i mogućnosti aplikacije te način na koji je razvijena aplikacija.

5.1. Pregled aplikacije

Kao što je spomenuto, aplikacija "Faksistent" služi za kontrolu ostvarenih bodova korisnika tijekom semestra na pojedinim predmetima. Dakle, prijavljeni korisnik ima mogućnost za svaki test u nekom predmetu unijeti broj bodova te zadatak aplikacije je, na temelju tih bodova, korisniku dati uvid u uspješnost nekog predmeta.

Jasno, aplikacija ne može biti samo veza između korisnika i predmeta, nego je potrebno dodati još neke module. Tako da, korisnik ima mogućnost definiranja semestra te odabira predmeta za taj semestar, može kreirati predloške za predmet (gdje se definiraju svi testovi te ograničenja za polaganje predmeta).

Na koncu, dodan je modul za komentare koji pruža mogućnost korisnicima postavljati informacije, pitanja i odgovore koji su vezani uz neki predmet.

Aplikacija se sastoji od sljedećih modula:

- Prijava i registracija
- Predmeti
- Semestri
- Predlošci za predmete
- Korisnički predmeti
- Komentari

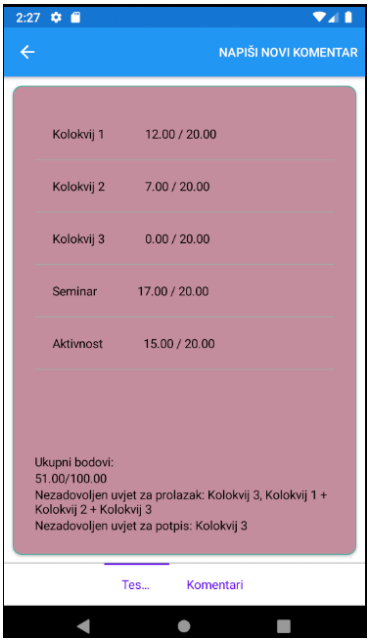
Prvi modul je, jasno, prijava i registracija. Modul predstavlja početak rada aplikacije i nije moguće pristupiti nijednom drugom modulu bez da je korisnik već prijavljen u aplikaciju. Pored standardne prijave i registracije, korisnik ima opciju birati studij na koji se želi registrirati ili prijaviti.

Modulu predmeti mogu pristupati samo administratori aplikacije te imaju pristup CRUD opcijama za predmete, odnosno moguće je pregledavati, kreirati, uređivati i brisati predmete. Ovaj modul je jedini modul kojemu obični korisnici nemaju pristup.

Semestrima mogu pristupati svi korisnici te također su dostupne CRUD s tim da korisnici mogu odabirati zadani semestar (semestar koji će biti prikazan prilikom rada aplikacije) te za svaki semestar se mogu odabrati predmeti koji se polažu u tom semestru.

Ideja modula predlošci za predmete je ta da korisnik definira testove, bodove na testovima i ograničenja za neki predmet ovisno o modelu praćenja tog predmeta. Korisnik ima opciju pregledavati javne i privatne predloške te kreirati nove. Kreirani predložak služi za prikazivanje uspješnosti polaganja nekog predmeta.

Korisnički predmeti predstavljaju centar aplikacije, nalaze se na početnoj stranici koja se otvara nakon što se korisnik uspješno prijavi. Korisniku se prikazuju svi predmeti koje je upisao u zadanom semestru te njihovi predlošci. Ulaskom na detalje jednog predmeta korisnik može uređivati broj bodova za neki test te mu aplikacija izračunava uspješnost polaganja tog predmeta. Izgled pogleda za detalje korisničkog predmeta se može vidjeti na slici 15.



Slika 15: Izgled pogleda za prikaz detalja korisničkog predmeta

Na tom pogledu se prvo može vidjeti lista testova ("Kolokvij 1", "Kolokvij 2"...) unutar koje se za svaki test se vidi broj bodova koje je korisnik ostvario i ukupan broj bodova za taj test (vrijednost je definirana u predlošcima za predmete). Na dnu pogleda se nalaze podaci o uspješnosti polaganja gdje je prikazan ukupan broj bodova nakon čega su ispisana sva ograničenja koje korisnik nije zadovoljio (ako nije zadovoljio).

Zadnji modul u ovoj aplikaciji je modul komentara. Komentari su vezani za predmete i korisnik ima mogućnost pregledavanja komentara, kreiranja novog komentara te odgovaranja na neki komentar.

U ovom poglavlju su kratko objašnjenje funkcionalnosti aplikacije, međutim, svaki od

ovih modula se detaljnije opisuje u narednim poglavljima.

5.2. Alati

Aplikacija se sastoji od dva rješenja, serverska i klijentska aplikacija. Serverska aplikacija služi za komunikaciju s bazom, dijelove poslovne logike te pruža API koji poziva klijentska aplikacija. Serverska aplikacija je izrađena u alatu Visual Studio 2022 koristeći ASP.NET razvojni okvir. Ova aplikacija je izrađena koristeći predložak ASP.NET Boilerplate [27].

Serverska aplikacija se neće detaljnije opisivati obzirom da nije predmet ovoga rada.

Klijentska aplikacija je, naravno, Xamarin projekt kojim ćemo se baviti u ovom dijelu. Kao i serverska, ova aplikacija je izrađena u alatu Visual Studio 2022 sa Xamarin.Forms verzijom 5.0.0.2196 [28].

Za olakšavanje razvoja aplikacije, u projekt je uključeno nekoliko paketa koji se mogu vidjeti na tablici 8.

Tablica 8: Korišteni paketi u praktičnom rješenju

Naziv	Verzija	Opis
Newtonsoft.Json	13.0.1	Serijalizacija i deserijalizacija podataka u Json formatu [29]
SQLitePCLRaw.core	2.0.4	Sprema podatke pomoću SQLite baze podataka [30]
DevExpress.XamarinForms.CollectionView	21.2.5	Komponente korisničkog sučelja za prikaz kolekcija [31]
DevExpress.XamarinForms.Editors	21.2.5	Komponente korisničkog sučelja za unos podataka [32]
DevExpress.XamarinForms.Navigation	21.2.5	Komponente korisničkog sučelja navigaciju aplikacijom [33]

Vrijedi posebno izdvojiti DevExpress.XamarinForms pakete čije su komponente intenzivno korištene u svakom od pogleda te je značajno olakšalo i ubrzalo razvoj aplikacije.

5.3. Implementacija

U ovom poglavlju će se obraditi način implementacije ove aplikacije najprije po korištenim arhitekturalnim uzorcima, a zatim po modulima aplikacije.

Zadani predložak Xamarina stvara tri projekta u rješenju, jedan bez ekstenzije gdje se

nalazi većina koda (*) te druga dva koja služe za razvoj na zasebnim platformama, tj. jedan projekt za Android (*.Android) i jedan za iOS (*.iOS).

Ovo praktično rješenje ima malo drugačiju strukturu projekata u usporedbi sa zadanim Xamarin predloškom, tako da se ovo sastoji od četiri projekta, to su FaksistentX.Services (sadrži poslovnu logiku aplikacije), FaksistentX.Shared (projekt FaksistentX je preimenovan u FaksistentX.Shared), FaksistentX.Android i FaksistentX.iOS.

Projekt FaksistentX.Services se može promatrati kao komponenta Model, u njega je izdvojena poslovna logika i komunikacija sa serverskom aplikacijom preko API-a. Također, ovaj projekt sadrži klase kao modele podataka.

Za ovaj rad, najzanimljiviji je projekt FaksistentX.Shared koji se sastoji od pogleda, klasa pripadnih pogledima te klasama za navigaciju aplikacijom.

Projekti FaksistentX.Android i Faksistent.iOS služe za konkretno definiranje Android, odnosno, iOS aplikacije te, osim registriranja nekih DevExpress komponenti, nije ništa mijenjano u odnosu na početni predložak koji Visual studio izgenerira.

Dakle, četiri projekta od kojih je sačinjena aplikacija, se mogu, uglavnom, svesti na samo dva, FaksistentX.Services i FaksistentX.Shared u kojima se nalazi najveći dio koda.

U aplikaciji je realizirana arhitektura s više stanara (engl. Multi-Tenancy) najviše zbog toga što ASP.NET Boilerplate, koji je korišten u serverskoj aplikaciji, nudi tu arhitekturu. Ova arhitektura pruža mogućnost da različiti stanari koriste istu aplikaciju i istu bazu podataka s različitim pristupom podacima [34]. Ta arhitektura, za ovu aplikaciju, je realizirana tako što jedan studij predstavlja jednog stanara. Time je omogućeno da svaki studij (konkretno na Fakultetu Organizacije i Informatike: IPS, IPI, OPS...) ima svoje korisnike i svoje predmete. Također, ovo omogućava potencijalno širenje aplikacije na nove fakultete sa svojim studijima. Svaki studij ima dvije vrste korisnika, jedna je administrator, a druga obični korisnik.

Arhitekturalni uzorci koji su objašnjeni u prethodnim poglavljima se u pravilu ne bi smjeli miješati, ali, u svrhu proučavanja prednosti i nedostataka pojedinih arhitektura, preuzeti su neki koncepti iz MVVM-a, MVC-a i Flux-a.

Više o tome kako je koji uzorak korišten će pisati u narednom poglavlju.

5.3.1. Korišteni arhitekturalni uzorci dizajna

Kao što je spomenuto, u ovoj aplikaciji su korišteni principi iz tri arhitekturalna uzorka, MVVM, MVC i Flux. Uzorak VIPER je izostavljen iz razloga što ne omogućava ništa što se ne može obuhvatiti s navedena tri uzorka na bolji, ili barem, podjednako dobar način.

Uzorak MVVM je prožet kroz cijelo rješenje obzirom da svaki pogled ima svoju pripadnu ViewModel klasu kojoj su glavna zaduženja ažuriranje pogleda te dohvaćanje podataka iz Model-a.

Uzorak MVC je iskorišten ponajviše za navigaciju aplikacije uz, u nekim slučajevima, korištenje mogućnosti prosljeđivanja cijelog objekta kroz upit.

Uzorak Flux je koristan za podatke koji trebaju biti dostupni globalno zbog jednostavnosti dohvaćanja i mijenjanja tih podataka.

Više o načinu implementiranja i korištenja svakog uzorka se može vidjeti u narednim potpoglavljima.

5.3.1.1. MVVM

Uzorak MVVM je, od svih uzoraka, najviše zastupljen u ovoj aplikaciji. Može se reći da je aplikacija realizirana na sličan način kao i, ranije naveden, primjer implementacije MVVM-a uz iznimku korištenja Controller-a iz MVC uzorka koji oduzima odgovornost za navigaciju i registriranje ruta komponenti ViewModel. Također, postoje neki specifični slučajevi gdje se koristi uzorak Flux što će se objasniti u narednim poglavljima.

Način iskorištavanja ovog uzorka će se objasniti tako što će se prikazati modul predmeti. Bitno je napomenuti da svaki od modula ima sličan način implementiranja uz neke dodatke iz drugih uzoraka. Dijagram klasa za modul Predmeti se može vidjeti na slici 16.

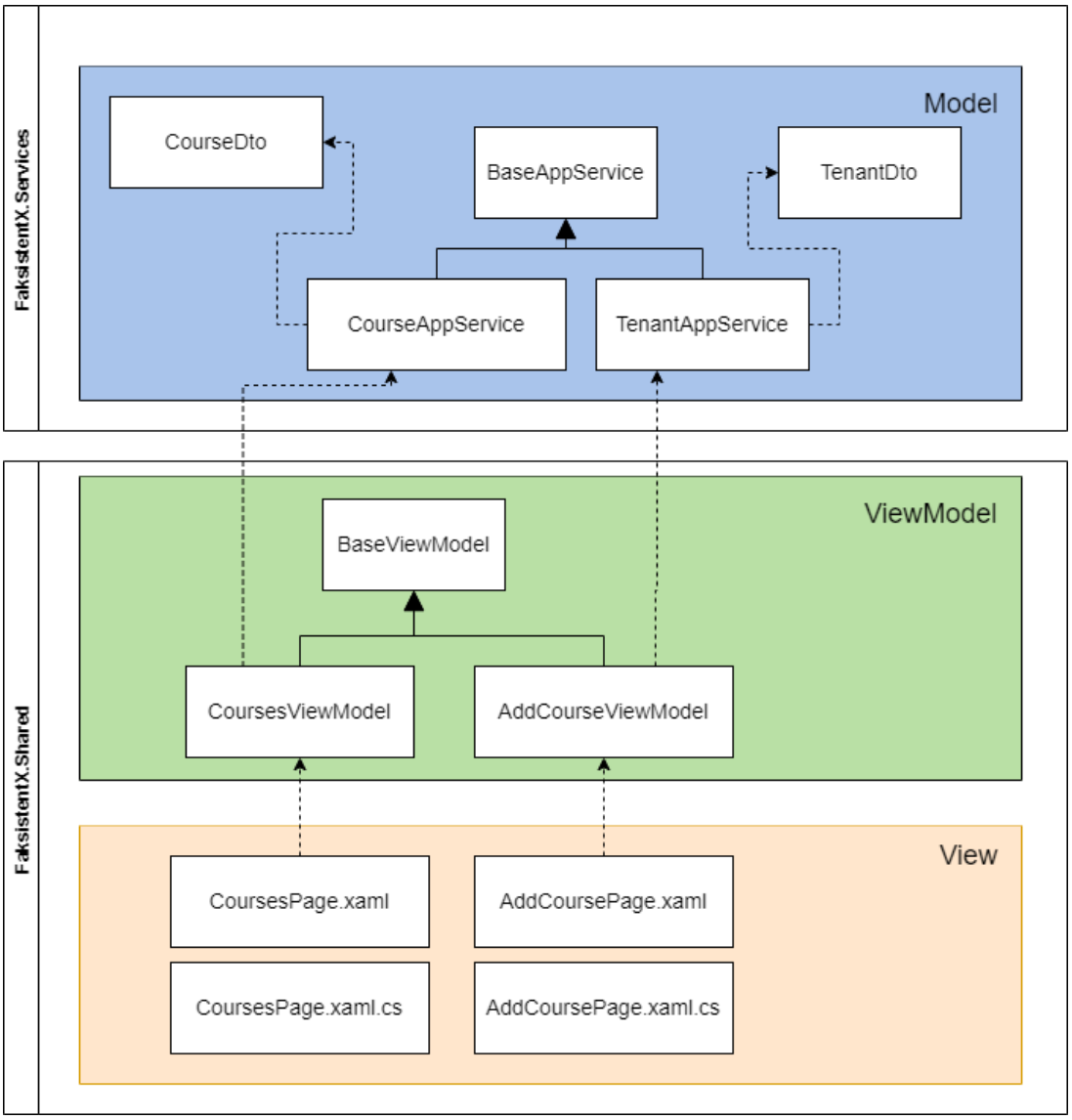
Na dijagramu klasa jasno se mogu izdvojiti tri komponente koje čine MVVM uzorak: Model, View i ViewModel. Prvo ćemo se fokusirati na komponentu View.

Iz prethodnih poglavlja znamo da View čine pogledi sa svojim pripadnim klasama. U ovom konkretnom primjeru *.xaml datoteke sadrže elemente korisničkog sučelja, a pripadne klase (*.xaml.cs) sadrže minimalnu količinu koda, a to je, najčešće, vezanje uz ViewModel klasu i eventualno gaženje metoda iz natklase Page. Dakle, u toj pripadnoj klasi se View veže za ViewModel komponentu. Primjer konkretne pripadne klase (CoursesPage.xaml.cs) se može vidjeti na dijelu koda 5.1.

```
1         public partial class CoursesPage : ContentPage
2     {
3         CoursesViewModel viewModel;
4         public CoursesPage()
5         {
6             InitializeComponent();
7 
8             BindingContext = viewModel = new CoursesViewModel();
9         }
10 
11        protected override void OnAppearing()
12        {
13            base.OnAppearing();
14 
15            viewModel.OnAppearing();
16        }
17    }
```

Kod 5.1: Klasa CoursesPage.xaml.cs

Na spomenutom dijelu koda se vidi konstruktor klase i metoda OnAppearing(). U kons-



Slika 16: MVVM - dijagram klasa za modul predmeta

truktoru se poziva metoda InitializeComponent() koja se automatski generira u parcijalnoj klasi koja je razvojnom programeru bitna samo radi toga što se mora pozvati u konstruktoru svake *.xaml.cs klase. Druga linija konstruktora je vezanje ViewModel klase za BindingContext tog pogleda. Metoda OnAppearing(), također, sadrži dvije linije od kojih je prva pozivanje OnAppearing() metode u natklasi, a druga je pozivanje OnAppearing() metode u ViewModel-u.

Tu zapravo staju sve zaduženja *.xaml.cs klase, učitavanje podataka za pogled je delegirano ViewModel klasi.

Metoda OnAppearing() unutar ViewModel-a je, u slučaju za predmete, zadužena za učitavanje svih predmeta te prikazivanje istih na pogledu.

Klasu CoursesViewModel.cs, osim spomenute metode OnAppearing() sadrži još jednu metodu, konstruktor i nekoliko varijabli. Konstruktor i varijable se mogu vidjeti na dijelu koda 5.2.

```
1         private readonly CourseAppService _courseAppService;
```

```
2         public ObservableCollection<CourseDto> Courses { get; set; }
3
4         public Command<CourseDto> AddCourseCommand { get; set; }
5
6         public CoursesViewModel()
7         {
8             _courseAppService = new CourseAppService();
9             AddCourseCommand = new Command<CourseDto>(OnAddCourseCommand);
10
11            Courses = new ObservableCollection<CourseDto>();
12        }
```

Kod 5.2: Klasa CoursesViewModel.cs - varijable i konstruktor

U klasi CoursesViewModel.cs postoje tri varijable, instanca klase CourseAppService.cs koja se instancira u konstruktoru klase da bi bila dostupna u cijeloj klasi, lista tipa ObservableCollection koja sadrži listu objekata klase CourseDto.cs koja se, također, instancira u konstruktoru, te komanda AddCourseCommand koja se u konstruktoru veže za metodu OnAddCourseCommand() koja se nalazi u ovoj klasi. Razlog zašto je komanda generičkog tipa (za klasu CourseDto.cs) je taj što se iz pogleda može proslijediti objekt u slučaju da je potrebno ažurirati neki predmet.

Spomenuta metoda OnAppearing() se može vidjeti na dijelu koda 5.3.

```
1         public async void OnAppearing()
2         {
3             IsBusy = true;
4             var courses = await _courseAppService.GetAllAsync(new
                  CourseRequestDto());
5
6             Courses.Clear();
7             foreach (var course in courses)
8             {
9                 Courses.Add(course);
10            }
11            IsBusy = false;
12        }
```

Kod 5.3: Klasa CoursesViewModel.cs - OnAppearing()

Kao što je spomenuto, metoda OnAppearing(), jednostavno rečeno, iz klase CourseAppService.cs dohvaća sve predmete (trenutno je bitno samo da metoda GetAllAsync() iz te klase vraća listu objekata tipa CourseDto.cs) kojima popunjava listu Courses unutar CoursesViewModel.cs klase. Nakon što je izvršena ova metoda, lista naziva Courses sadrži sve predmete koji se trebaju prikazati na pogledu i View ima ulogu da referencira tu listu način na koji je View povezan s tom listom se može vidjeti na dijelu koda 5.4.

```
1         <dxcv:DXCollectionView ItemsSource="{Binding Courses}"
              x:Name="CoursesCollectionView">
2             <dxcv:DXCollectionView.ItemTemplate>
```

```
3                       <DataTemplate>
4                           <dxcv:SwipeContainer>
5                               <dxcv:SwipeContainer.ItemView>
6                                   <StackLayout Margin="0" Spacing="0">
7                                       <Label Margin="20" Text="{Binding
                                            Name}"/>
8                                       <BoxView Style="{StaticResource
                                            SeparatorStyle}"/>
9                                   </StackLayout>
10                              </dxcv:SwipeContainer.ItemView>
11
12                              <dxcv:SwipeContainer.StartSwipeItems>
13                                  <dxcv:SwipeItem Caption="Uredi"
14                                          BackgroundColor="Blue"
15                                          Command="{Binding
                                                Source={RelativeSource
16                                          AncestorType={x:Type
                                                local:CoursesViewModel}},
17                                          Path=AddCourseCommand}"/>
18                              </dxcv:SwipeContainer.StartSwipeItems>
19                          </dxcv:SwipeContainer>
20                      </DataTemplate>
21                  </dxcv:DXCollectionView.ItemTemplate>
22              </dxcv:DXCollectionView>
```

Kod 5.4: Pogled CoursesPage - lista predmeta

U prvoj liniji koda se može vidjeti atribut ItemsSource čija je vrijednost "Binding Courses" što znači da se na elementu CollectionView iz te linije prikazuju predmeti koji se nalaze u varijabli Courses iz CoursesViewModel.cs klase.

Na dijelu koda 5.2 je prikazano inicijaliziranje komande AddCourseCommand i, slično kao i s listom Courses, komponenta View ima ulogu da referencira tu komandu. Na linijama 15-17 se može vidjeti vezanje za komandu AddCourseCommand za koju je već spomenuto da se okida prilikom uređivanja (ili kreiranja) predmeta. Ta komanda, dalje, okida metodu OnAddCourseCommand() koja se može vidjeti na dijelu koda 5.5.

```
1           public async void OnAddCourseCommand(CourseDto course = null)
2           {
3               if(course != null)
4               {
5                   InvokeControllerMethod("Courses", "AddCourse", new
                        EntityDto(course.Id));
6
7               }
8               else
9               {
10                  InvokeControllerMethod("Courses", "AddCourse");
11              }
```

```
12          }
```

Kod 5.5: Klasa CoursesViewModel.cs - OnAddCourseCommand()

Metoda OnAddCourseCommand() kao argument prima objekt klase CourseDto.cs (koji je NULL u slučaju da se kreira novi predmet) te, u ovisnosti od tog argumenta, navigira na pogled AddCoursePage. Navigiranje je riješeno koristeći principe iz MVC uzorka te će više o tome saznati u narednom poglavlju.

U ovom poglavlju se neće detaljno objašnjavati Model aplikacije obzirom da nije bitan za arhitekturu aplikacije. Važno je napomenuti da klasa CourseAppService.cs, pomoću svoje nat-klase BaseAppService.cs, šalje API zahtjeve na serversku aplikaciju nakon čega deserijalizira vrijednost dobivenu s API-a.

Pogled AddCoursePage se može svesti na sličan način kao i pogled CoursesPage. Iz klase AddCoursePage.xaml.cs se pogled veže na AddCourseViewModel.cs koji u svojoj OnAppearing() metodi učitava objekt tipa CourseDto.cs (u slučaju da je se predmet uređuje) te jedina razlika je u vezanju varijabli iz ViewModela s pogledom obzirom da veza ovaj put mora biti dvosmjerna iz razloga što korisnik kroz pogled može mijenjati podatke o nekom predmetu. Da bi se to prikazalo, promatrat će se podatak o nazivu predmeta. U klasu AddCourseViewModel.cs postoji varijabla Name koja se deklarira na način koji je prikazan u dijelu koda 5.6.

```
1           private string _name;
2
3           public string Name
4           {
5               get => _name;
6               set => SetProperty(ref _name, value);
7           }
```

Kod 5.6: Varijabla za naziv predmeta u klasi AddCourseViewModel.cs

Za varijablu Name se postavljaju get i set metode od kojih se, u set metodi, poziva metoda SetProperty() (ova metoda se nalazi u BaseViewModel.cs klasi te je došla uz Xamarinov predložak) čime se postiže ažuriranje pogleda prilikom unosa podataka. Na dijelu koda 5.7 se vidi način na koji se veže element u pogledu za tu varijablu.

```
1           <dxe:TextEdit LabelText="Naziv" Text="{Binding Name}"/>
```

Kod 5.7: Varijabla za naziv predmeta u pogledu AddCoursePage

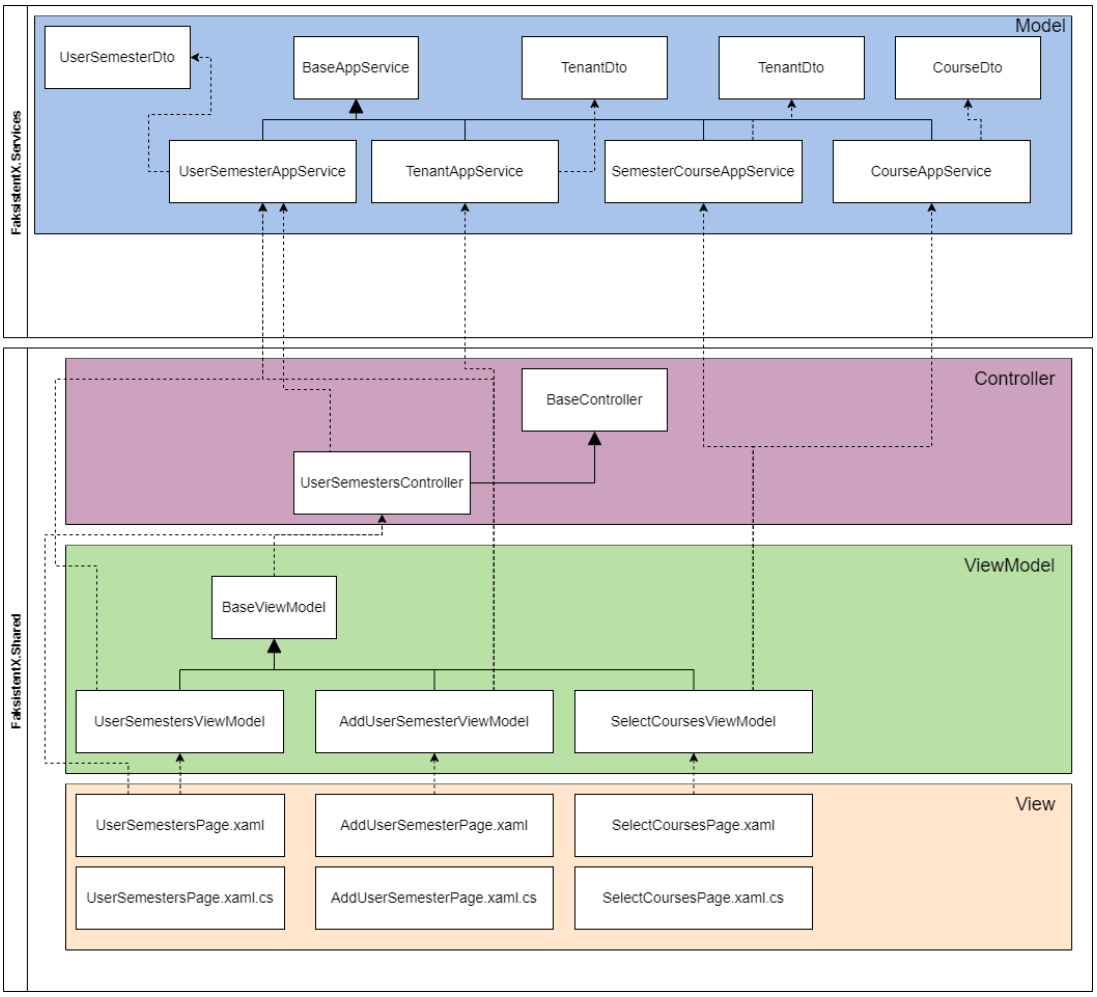
U kodu se može vidjeti da je atribut Text postavljen na "Binding Name" čime se označava to da vrijednost tog elementa je jednaka toj varijabli u ViewModel klasi.

Modul za predmete je samo jedan od modula koji je realiziran pomoću MVVM uzorka, ali nije potrebno objašnjavati druge module jer se sve posebnosti MVVM uzorka svode na već prikazane mogućnosti. To su učitavanje i prikazivanje podataka, pružanje varijabli i komanda na koje se pogled može vezati.

5.3.1.2. MVC

Uzorak MVC je iskorišten za ovu aplikaciju najviše iz razloga da olakša navigaciju aplikacijom. Shodno tome, rute su se registrirale na isti način kao u ranijem MVC poglavlju. Također, navigiranje aplikacije se izvodi na isti način kao u tom poglavlju. U tom poglavlju je spomenuto da se navigacija izvršava pomoću metode ChangeView() koja pruža mogućnost prosljeđivanja argumenata pogledu na standardni način i pomoću JSON vrijednosti.

Da bi bolje shvatili kako je realiziran MVC uzorak, fokusirat ćemo se na modul za semestre. Dijagram klasa za modul semestara se može vidjeti na slici 17.



Slika 17: MVC - dijagram klasa za modul semestri

U odnosu na modul za predmete, modul za semestre je nešto kompleksniji jer sadržava više pogleda te je nužno referenciranje na veći broj klasa u Model komponenti. Iako MVC implicira tri komponente (plus nužni ViewModel), u ovom poglavlju će se staviti fokus isključivo na Controller komponentu i na to što ona donosi jer su komponente Model i View iste kao u MVVM uzorku.

Obzirom da je komponenta Controller služi samo za navigaciju aplikacijom, važno je napomenuti da se ovaj modul sastoji od tri pogleda, to su UserSemestersPage koji sadrži prikaz svih semestara te s kojeg je moguće prijeći na pogled AddUserSemesterPage koji služi za

dodavanje i uređivanje semestra. S pogleda AddUserSemesterPage je moguće navigirati na pogled SelectCoursesPage gdje korisnik odabire predmete u tom semestru.

Komponenta Controller sadrži samo dvije klase, to su BaseController.cs koja je natklasa drugoj klasi UserSemestersController.cs. Klasa BaseController.cs je ista kao klasa Controller.cs navedena ranije u MVC primjeru, točnije sadrži samo jednu javnu metodu, ChangeView() koja je zadužena za navigiranje aplikacijom. Ta metoda pomoću refleksije dohvaća naziv klase i metode koja ju je pozvala te s tim podacima navigira na određeni pogled. Pored toga, metoda ChangeView() olakšava prosljeđivanje argumenata pogledu pomoću standardnog načina ili pomoću JSON-a.

Komponenta Controller, osim navigacije može poslužiti kao vezivno tkivo između View-a i Model-a što je iskorišteno u ovom modelu da bi se preskočio ViewModel u nekim slučajevima. Uzevši u obzir tu mogućnost, metode unutar klase UserSemestersController.cs se mogu podijeliti na metode koje služe za navigaciju i metode koje služe kao vezivno tkivo. Na dijelu koda 5.8 se mogu vidjeti dvije metode koje služe za navigaciju.

```
1         public async Task AddUserSemesterPage(string Id = "")
2         {
3             if (!string.IsNullOrEmpty(Id))
4             {
5                 var semester = await _userSemesterAppService.GetAsync(Id);
6                 await ChangeView(semester, true);
7             }
8             else
9             {
10                await ChangeView();
11            }
12        }
13
14        public async Task SelectCoursesPage(string Id)
15        {
16            await ChangeView(new EntityDto(Id));
17        }
```

Kod 5.8: Metode za navigaciju u klasi UserSemestersController.cs

Spomenuto je kako se ovaj modul sastoji od tri pogleda, međutim, na pogled UserSemestersPage se navigira isključivo iz navigacijskog modula što znači da nam ta metoda nije potrebna unutar Controller-a. Dakle, u Controller-u postoje dvije metode, jedna za navigiranje na pogled AddUserSemesterPage i druga za navigiranje na pogled SelectCoursesPage.

Metoda AddUserSemesterPage() prima argument Id koji je ključ od semestra koji se želi uređivati s tim da je ta vrijednost prazan string ako se dodaje novi predmet. Unutar metode se vidi da se provjerava je li taj string prazan te ako nije, onda se iz Model-a dohvaća cijeli objekt koji se prosljeđuje metodi ChangeView (pored tog objekta, argument "true" omogućuje da se taj objekt prosljeđuje kao JSON vrijednost). U slučaju da je vrijednost ključa prazan string, onda se poziva metoda ChangeView() bez ikakvih argumenata.

Nadalje, parametri koji su proslijeđeni stranici se mogu dohvatiti u ViewModel klasi. U slučaju da je to cijeli objekt koji je serijaliziran u JSON vrijednost, ViewModel tu vrijednost deserijalizira da bi dobio C# objekt koji se na kraju prikazuje na formi.

Pored metoda za navigaciju, UserSemestersController.cs sadrži još dvije metode koje se mogu vidjeti na dijelu koda 5.9.

```
1         public async Task<bool> DeleteUserSemester(string id)
2         {
3             return await _userSemesterAppService.DeleteAsync(id);
4         }
5
6         public async Task<bool> SetIsSelected(string id)
7         {
8             return await _userSemesterAppService.SetIsSelectedAsync(id);
9         }
```

Kod 5.9: Metode za vezanje View-a i Model-a u klasi UserSemestersController.cs

Ta funkcionalnost Controller-a je jasna, klasa UserSemestersController.cs pruža dvije metode koje se mogu pozvati direktno iz View-a da se preskoči ViewModel. Te metode pozivaju određenu metodu iz Model-a čime se, u ovim konkretnim slučajevima, semestar briše ili postavlja kao odabrani.

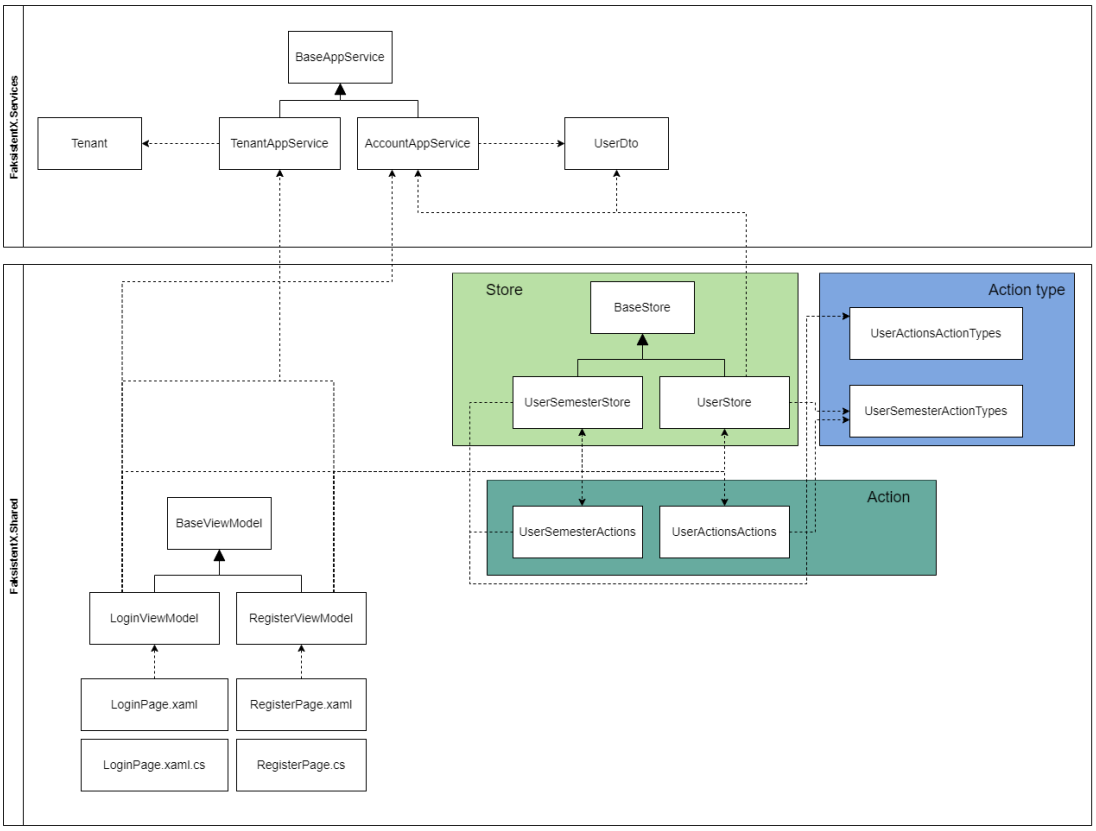
5.3.1.3. Flux

Pored korištenja uzoraka MVVM i MVC, u ovoj aplikaciji je se, u određenom obujmu, iskoristio i uzorak Flux. Konkretno za modul navigacije. U navigaciji je, osim stavki za navigiranje, bilo potrebno prikazati korisničko ime prijavljenog korisnika te naziv odabranog semestra. Te dvije vrijednosti su globalne što znači da se u jednom trenutku, u jednoj sesiji postoji samo jedan prijavljen korisnik sa samo jednim odabranim semestrom. Uzimajući u obzir te uvjete, uzorak Flux, konkretnije komponenta Store može uveliko pomoći za ostvarivanje navigacije.

Ako se prisjetimo poglavlja gdje je se definirao Flux, znamo da se za postavljanje vrijednosti u komponentu Store izvršava pomoću pozivanja akcija. Tu se može vidjeti još jedna prednost Flux-a za navigaciju jer se korisničko ime i naziv semestra mogu postavljati u različitim slučajevima. Tako se korisničko ime postavlja prilikom prijave ili registracije, a vrijednost naziva semestra prilikom prijave (ako je u nekoj ranijoj sesiji postavljen zadani semestar) ili prilikom postavljanja zadanog semestra.

Obzirom da prijavljivanje u aplikaciju obuhvaća oba slučaja pozivanja akcije, da bi bolje razumjeli implementaciju Flux-a u ovom primjeru, prvo će se prikazati dijagram klasa za prijavu i registraciju koji se može vidjeti na slici 18.

Na dijagramu klasa su označene samo komponente koje su bitne za uzorak Flux, a to su Store, Action i Action type (Dispatcher je izostavljen zbog jednostavnosti). Modul prijave i registracije ne dohvaća nikakve podatke iz Store-a, nego samo poziva akcija pomoću koji se učitavaju podaci u Store koji se kasnije čitaju.



Slika 18: Flux - dijagram klasa za prijavu i registraciju

Prilikom uspješne prijave, pozivaju se dvije akcije na način koji se može vidjeti na dijelu koda 5.10.

```
1         _userSemesterActions.GetCurrentSemester();
2
3         _userActions.SetUser(user);
```

Kod 5.10: Pozivanje akcija iz ViewModel-a

Može se zaključiti da je dio za koji je ViewModel odgovoran jako jednostavan. Pozivaju se dvije akcije, jedna unutar klase UserSemesterActions.cs, a druga unutar klase UserActions.cs. Tu završava sva odgovornost ViewModel komponente te daljnju logiku preuzimaju Flux komponente. U slučaju klase UserActions.cs, poziva se metoda SetUser() s argumentom prijavljenog korisnika. Ta metoda se može vidjeti na dijelu koda 5.11.

```
1         public void SetUser(UserDto input)
2         {
3             DependencyService.Get<Dispatcher>().Invoke<object>
4                 (UserActionTypes.SET_USER, input);
5         }
```

Kod 5.11: Metoda SetUser unutar klase UserActions.cs

Ta metoda je, također, jednostavna te sve što radi je dohvaća Dispatcher.cs klasu te poziva njezinu metodu Invoke s parametrima naziva akcije (naziv je definiran u klasi UserActionTypes.cs) i s parametrom korisnika koji je prijavljen. Za Dispatcher znamo da akcije prosljeđuje

svim Store-ovima te klasa UserStore.cs je jedini Store koji reagira na tu akciju. UserStore.cs klasa se može vidjeti na dijelu koda 5.12.

```
1         public override async Task Invoke<TData>(string eventType, TData
              data)
2         {
3             switch (eventType)
4             {
5                 case UserActionTypes.GET_USER:
6                     await GetUser();
7                     break;
8                 case UserActionTypes.SET_USER:
9                     await SetUser(data as UserDto);
10                    break;
11            }
12        }
```

Kod 5.12: Klasa UserStore.cs

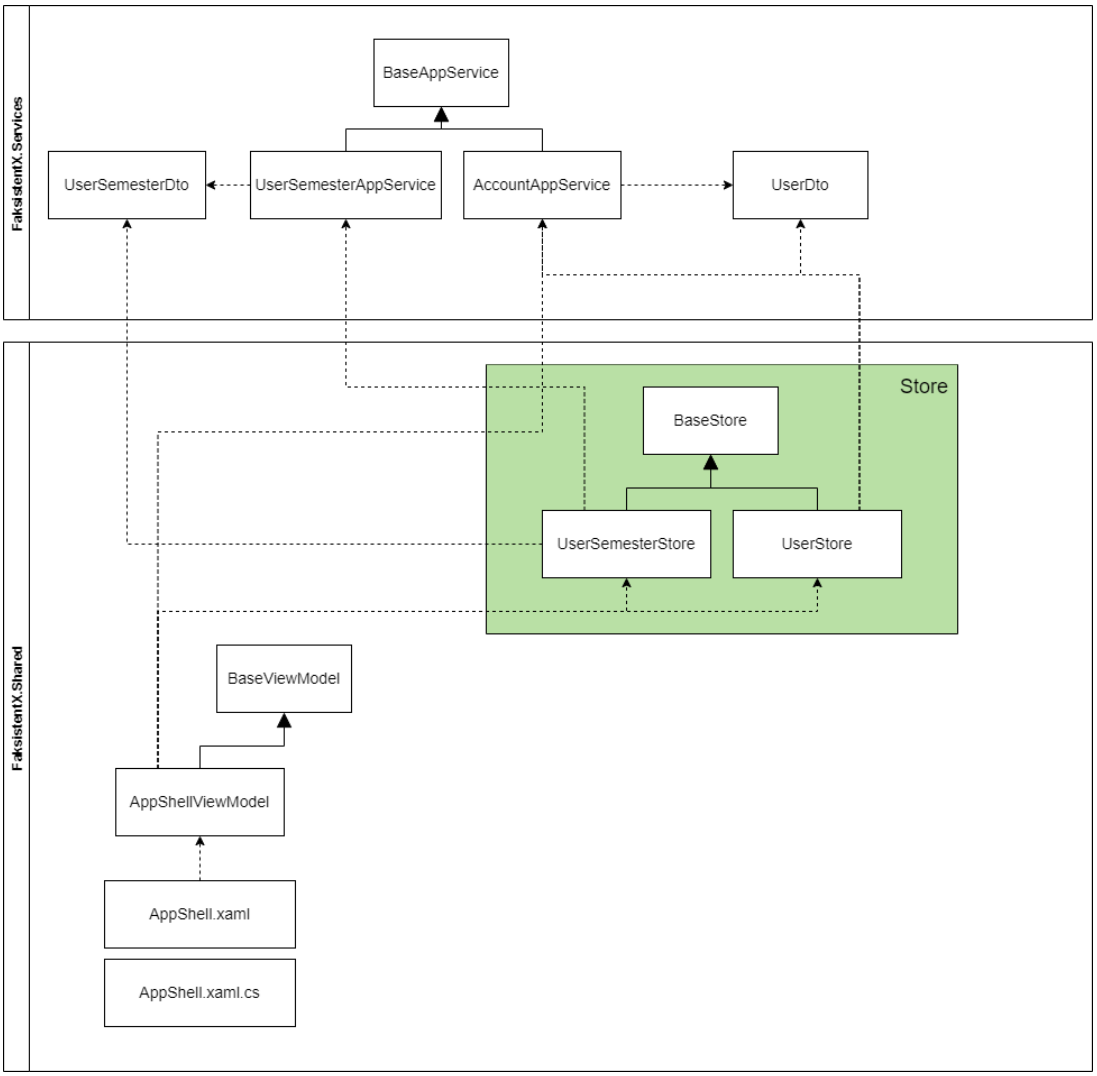
Iz prethodnog primjera za Flux znamo da svaki Store ima metodu Invoke koja se poziva iz Dispatcher-a. Unutar te metode se može vidjeti da, u slučaju da je akcija jednaka "SET_USER" akciji, onda se poziva metoda SetUser() koja postavlja vrijednost varijable Data na dobiveni parametar. U ovome trenutku je ta varijabla postavljena te je dostupna globalno cijeloj aplikaciji.

Spomenuto je da je modul navigacija jedini modul koji koristi podatke iz Store-ova. Dijagram klasa za taj modul je moguće vidjeti na slici 19.

Obzirom da modul za navigaciju ne okida nikakve akcije, nego samo čita iz Store-a, na dijagramu je prikazana samo komponenta Store. Ranije u tekstu je navedeno u kojim slučajevima se Store popunjava podacima te znamo da u trenutku učitavanja navigacije, ti podaci su se već učitali. Dakle, posao komponente ViewModel u ovom modulu je pročitati te podatke. Način na koji se ti podaci čitaju se može vidjeti na dijelu koda 5.13.

```
1         public async void OnAppearing()
2         {
3             IsBusy = true;
4
5             _userSemesterStore =
                  DependencyService.Get<UserSemesterStore>();
6             _userStore = DependencyService.Get<UserStore>();
7
8             SemesterItem = "Semestri (" + (_userSemesterStore.Data?.Name
                  ?? "Nije odabrano") + ")";
9             LogoutItem = "Odjava (" + (_userStore.Data?.UserName ?? "Nije
                  odabrano") + ")";
10
11            IsBusy = false;
12        }
```

Kod 5.13: Čitanje podataka iz Store-a iz ViewModel klase



Slika 19: Flux - dijagram klasa za navigaciju

U metodi OnAppearing(), koja se okida prilikom učitavanja navigacije, se može vidjeti da se prvo pomoću injektiranja ovisnosti dohvaćaju dva potrebna Store-a nakon čega se iz tih Store-ova dohvaća varijabla Data u kojoj se nalazi objekt prijavljenog korisnika, odnosno odabranog semestra. Nakon toga se ti podaci spremaju u varijable u ViewModel-u koje View može čitati te prikazivati na pogledu.

Prikazivanje tih podataka na navigaciji je se moglo odraditi na velik broj načina, međutim, u usporedbi s ostalim obrađenim uzorcima, Flux je pružio najljepše rješenje gdje se pomoću akcija jednostavno mogu nanovo učitati ti podaci koji će se, na koncu prikazivati u navigaciji.

6. Zaključak

Vraćajući se na uvodna poglavlja ovog rada, jasno je da su dvije stvari stavljene u glavni fokus. To su Xamarin kao tehnologija te korištenje različitih arhitekturalnih uzoraka u Xamarinu.

Za Xamarin je spomenuto da pruža veliku prednost u odnosu na nativne tehnologije, a to je ta da je moguće sa samo jednim projektom napraviti aplikacije i za Android i za iOS. Jasno to dovodi do nekih nedostataka koliko zapravo Xamarin tu olakšava posao i je li bolji s konkurentnim tehnologijama kao što su Flutter ili React Native.

Spomenuto je također da je Microsoft predstavio .NET MAUI koji je nadogradnja na Xamarin te se postavlja pitanje o isplativosti razvoja u Xamarinu obzirom da se može očekivati da .NET MAUI preuzme velik dio projekata. U trenutku pisanja ovog rada, .NET MAUI još nema stabilne verzije. Jedan način gledanja na izlazak .NET MAUI-a je taj da će, vremenom, istisnuti Xamarin s tržišta, ali postoji i drugi način. Tehnologija .NET MAUI koristi iste jezike i iste okvire kao Xamarin što znači da će prilagodba biti svedena na minimum te druga prednost je ta da će se svi Xamarin projekti moći relativno jednostavno pretvoriti u .NET MAUI projekte [35].

U radu su definirana četiri arhitekturalna uzorka dizajna, MVVM, MVC, VIPER i Flux te je, uz jednostavne praktične primjere, zaključeno da uzorak MVVM nudi najviše prednosti za razvoj u Xamarinu te da taj uzorak ima neke nedostatke koje mogu nadoknaditi drugi uzorci.

Ti nedostatci MVVM-a su utjecali na odluku da se u praktičnom dijelu rada uz MVVM, iskoriste neki principi iz MVC i Flux uzorka. Uzorak VIPER je se pokazao kao uzorak koji ne pruža ništa što već spomenuta tri uzorka već ne pružaju.

Za potrebe praktičnog dijela rada je izrađena aplikacija "Faksistent" koja služi studentima za praćenje uspjeha u toku semestra te su u ovome radu prikazani načini na koji su iskorišteni pojedini arhitekturalni uzorci. Nakon toga je prikazana cijela aplikacija, modul po modul.

Source kod serverske aplikacije koja je korištena u praktičnom dijelu se može pronaći na poveznici https://github.com/bayo04/FaksistentAPI, a Xamarin aplikacija se može pronaći na poveznici https://github.com/bayo04/FaksistentX.

Na kraju se može zaključiti da se Xamarin i dalje isplati učiti i koristiti u projektima unatoč prednostima konkurentnih tehnologija. To se ponajviše odnosi na razvojne programere koji već imaju iskustva rada sa .NET razvojnim okvirom. Izlazak platforme .NET MAUI se može promatrati kao pozitivna stvar za Xamarin jer predstavlja nužno osvježenje tehnologije te razvojni programeri se trebaju minimalno prilagoditi s tim da već postojeće Xamarin projekte je moguće lako prebaciti na .NET MAUI tehnologiju. Tako da sve vrijeme utrošeno na stjecanje znanja u Xamarinu i sve vrijeme utrošeno u razvoj aplikacija se može iskoristiti u narednim godinama, samo što se to znanje i te aplikacije neće koristiti u Xamarinu nego u sličnom .NET MAUI-u.

Popis literature

[1] „Developer Guides," Google. (10. srpnja 2021.), adresa: https://developer.android.com/guide (pogledano 14. 5. 2022.).

[2] „Bring Your Ideas to Life," Apple Inc. (2022.), adresa: https://developer.apple.com/develop/ (pogledano 14. 5. 2022.).

[3] „React Native," Meta Platforms, Inc. (2022.), adresa: https://reactnative.dev/ (pogledano 14. 5. 2022.).

[4] „Flutter," flutter-dev@. (2022.), adresa: https://docs.flutter.dev/ (pogledano 15. 5. 2022.).

[5] „What is Xamarin?" Microsoft. (2021.), adresa: https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin (pogledano 21. 7. 2021.).

[6] „Xamarin.Forms release notes and roadmap," Microsoft. (29. ožujka 2022.), adresa: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/release-notes/ (pogledano 14. 5. 2022.).

[7] A. D. Gustavo Hartmann Geoff Stead, „Cross-platform mobile development," 2011. adresa: https://wss.apan.org/jko/mole/Shared%20Documents/Cross-Platform%20Mobile%20Development.pdf.

[8] M. Reynolds, „What is Mono?" *Xamarin Mobile Application Development for Android*, 2014., str. 23–24.

[9] „A tour of the C# language," Microsoft. (18. ožujka 2022.), adresa: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ (pogledano 7. 7. 2022.).

[10] „The history of C#," Microsoft. (16. lipnja 2022.), adresa: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history (pogledano 7. 7. 2022.).

[11] „What is .NET?" Microsoft. (2022.), adresa: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet (pogledano 7. 7. 2022.).

[12] „What is Xamarin.Forms?" Microsoft. (8. srpnja 2021.), adresa: https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin-forms (pogledano 19. 5. 2022.).

[13] D. Kermek, „Uzorci dizajna za strukturirane i distribuirane sustave," *Predavanja iz kolegija Uzorci dizajna*, Fakultet Organizacije i Informatike, 2020.

[14] U. Z. Paris Avgeriou. „Architectural Patterns Revisited – A Pattern Language.” (2005.), adresa: http://eprints.cs.univie.ac.at/2698/1/ArchPatterns.pdf (pogledano 17. 7. 2021.).

[15] M. I. M. Erik Sørensen. „Model-View-ViewModel (MVVM) Design Pattern using Windows Presentation Foundation (WPF) Technology.” (2010.), adresa: http://megabyte.utm.ro/articole/2010/info/sem1/InfoStraini_Pdf/1.pdf (pogledano 21. 8. 2021.).

[16] „Get started with WPF,” Microsoft. (2021.), adresa: https://docs.microsoft.com/en-us/visualstudio/designers/getting-started-with-wpf?view=vs-2019 (pogledano 21. 8. 2021.).

[17] „The Model-View-ViewModel Pattern,” Microsoft. (2021.), adresa: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm (pogledano 25. 8. 2021.).

[18] T. Reenskaug. „MVC XEROX PARC 1978-79.” (1979.), adresa: https://folk.universitetetiosl no/trygver/themes/mvc/mvc-index.html (pogledano 13. 8. 2021.).

[19] „MVC,” Mozilla Corporation. (2021.), adresa: https://developer.mozilla.org/en-US/docs/Glossary/MVC (pogledano 13. 8. 2021.).

[20] S. T. P. Glenn E. Krasner. „A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System.” (1988.), adresa: http://heaveneverywhere.com/stp/PostScript/mvc.pdf (pogledano 14. 8. 2021.).

[21] „ASP.NET Core,” Microsoft. (2022.), adresa: https://github.com/dotnet/aspnetcore (pogledano 22. 1. 2022.).

[22] J. Gilbert. „Architecting iOS Apps with VIPER.” (2014.), adresa: https://www.objc.io/issues/13-architecture/viper/ (pogledano 12. 8. 2021.).

[23] A. Pelykh. „Designing the Architecture of Your Mobile Product: 4 Patterns To Choose.” (2018.), adresa: https://medium.com/@brainbeanapps/designing-the-architecture-of-your-mobile-product-4-patterns-to-choose-35613564b8d3 (pogledano 13. 8. 2021.).

[24] SuavePirate. „Xamarin.Flux.” (2019.), (pogledano 2. 1. 2022.).

[25] J. Kagga. „Understanding React Components.” (2018.), (pogledano 11. 1. 2022.).

[26] „Flux,” Facebook. (2021.), (pogledano 11. 1. 2022.).

[27] D. Foundation. „What is the ASP.NET Boilerplate?” (2021.), (pogledano 13. 4. 2022.).

[28] Microsoft. „Xamarin.Forms 5.0.0.2196 (5.0.0 Service Release 6) Release Notes.” (29. ožujka 2022.), (pogledano 13. 4. 2022.).

[29] Newtonsoft. „Json.NET Documentation.” (2022.), (pogledano 13. 4. 2022.).

[30] Microsoft. „Using SQLite.NET with Android.” (2022.), (pogledano 13. 4. 2022.).

[31] „CollectionView,” Developer Express Inc. (2022.), (pogledano 24. 4. 2022.).

[32] „Data Editors,” Developer Express Inc. (2022.), (pogledano 24. 4. 2022.).

[33] „Navigation,” Developer Express Inc. (2022.), (pogledano 14. 4. 2022.).

[34] D. E. Inc. „What Is Multi-Tenancy?” (2022.), (pogledano 14. 4. 2022.).

[35] „Migrate your app from Xamarin.Forms,” Microsoft. (5. ožujka 2022.), adresa: `https://docs.microsoft.com/en-us/dotnet/maui/get-started/migrate` (pogledano 11. 6. 2022.).

Popis slika

1. Dijagram o načinu rada Xamarina [5] 4
2. Zadana struktura datoteka i direktorija 6

3. MVVM [17] 9
4. MVVM - dijagram klasa 10
5. MVVM - dijagram aktivnosti za navigaciju 13
6. MVC [19] 14
7. MVC - dijagram klasa 16
8. MVC - dijagram aktivnosti za navigaciju 22
9. Xamarin i ASP.NET - dijagram klasa 23
10. VIPER [23] 26
11. VIPER - dijagram klasa 29
12. VIPER - dijagram aktivnosti za navigaciju 33
13. Flux [26] 35
14. Flux - dijagram klasa 37

15. Izgled pogleda za prikaz detalja korisničkog predmeta 52
16. MVVM - dijagram klasa za modul predmeta 56
17. MVC - dijagram klasa za modul semestri 60
18. Flux - dijagram klasa za prijavu i registraciju 63
19. Flux - dijagram klasa za navigaciju 65

Popis tablica

1. Prosječno vrijeme učitavanja pogleda sa zapisima 44
2. Broj klasa 46
3. Broj linija koda i referenci na druge klase - ukupno 47
4. Broj linija koda i referenci na druge klase - View 47
5. Broj linija koda i referenci na druge klase - ViewModel 48
6. Broj linija koda i referenci na druge klase - poslovna logika 48
7. Broj linija koda i referenci na druge klase - ostale komponente 49
8. Korišteni paketi u praktičnom rješenju 53

Prilozi

1. Poveznica za source kod serverske aplikacije:
   https://github.com/bayo04/FaksistentAPI
2. Poveznica za source kod Xamarin aplikacije:
   https://github.com/bayo04/FaksistentX