Usporedba radnih okvira za razvoj višeplatformskih korisničkih sučelja

Kovačić, Nino

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:025347

Rights / Prava: In copyright/Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-29**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 613

USPOREDBA RADNIH OKVIRA ZA RAZVOJ VIŠEPLATFORMSKIH KORISNIČKIH SUČELJA

Nino Kovačić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 613

USPOREDBA RADNIH OKVIRA ZA RAZVOJ VIŠEPLATFORMSKIH KORISNIČKIH SUČELJA

Nino Kovačić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 613

Pristupnik: **Nino Kovačić (0036525448)**
Studij: Računarstvo
Profil: Programsko inženjerstvo i informacijski sustavi
Mentor: doc. dr. sc. Tomislav Jagušt

Zadatak: **Usporedba radnih okvira za razvoj višeplatformskih korisničkih sučelja**

Opis zadatka:

S povećanjem broja različitih računalnih i mobilnih platformi, razvoj višeplatformskih korisničkih sučelja postao je jedan od ključnih izazova u programskom inženjerstvu. U sklopu ovog diplomskog rada potrebno je analizirati i usporediti različite radne okvire namijenjene razvoju višeplatformskih aplikacija kako bi se identificirali optimalni alati za raznolike scenarije. Na primjeru jednostavne obrazovne aplikacije za učenje povijesti potrebno je prikazati postupak i specifičnosti razvoja u svakom od proučenih radnih okvira. Fokus diplomskog rada je na proučavanju radnog okvira Compose Multiplatform. Potrebno je osmisliti i implementirati aplikaciju koja će omogućiti nastavnicima unos lekcija iz povijesti koje će se učenicima prikazivati u obliku interaktivne vremenske lente. Potrebno je podržati različite sadržaje poput teksta, slika i videa, a kretanje po vremenskoj lenti i odabir lekcija mora biti slobodan i postignut uz korištenje interaktivnih animacija.

Rok za predaju rada: 28. lipnja 2024.

Želim se zahvaliti svojoj obitelji i pogotovo bratu Bonu na bezuvjetnoj ljubavi i podršci.

Sadržaj

1. Uvod . . . **3**

2. Višeplatformski radni okviri . . . **5**

2.1. Compose Multiplatform . . . 7

2.2. Flutter . . . 8

3. Korištene tehnologije . . . **10**

3.1. Kotlin . . . 10

3.1.1. Podrijetlo i razvoj . . . 10

3.1.2. Ključne značajke . . . 11

3.2. Gradle . . . 22

3.3. Ruby on Rails . . . 23

3.4. Dart . . . 23

4. Arhitektura aplikacije . . . **25**

4.1. Poslužitelj i baza . . . 26

4.2. Compose Multiplatform . . . 28

4.2.1. Kreiranje i struktura projetka . . . 28

4.2.2. Resursi i stilovi . . . 31

4.2.3. Umetanje ovisnosti . . . 33

4.2.4. Navigacija . . . 35

4.2.5. Mrežna komunikacija . . . 37

4.3. Flutter . . . 39

5. Korisničko sučelje . . . **43**

5.1. Ekran za prijavu korisnika . . . 43

5.2. Početni zaslon 47
5.3. Vremenska lenta 49
5.4. Detalji o događaju 53
5.5. Stanje bez internetske veze 57

6. Zaključak **60**

Literatura **62**

Sažetak **65**

Abstract **66**

1. Uvod

Razvojem sve većeg broja različitih tipova uređaja i operacijskih sustava koji se na njima pokreću, postaje sve teže razvijati programske proizvode koji bi se pokretali na više različitih uređaja. Tako dolazimo do situacije gdje, ako želimo na tržište plasirati proizvod koji bi se mogao pokretati na svakoj platformi, trebamo veliki broj programera koji su specijalizirani za pojedinu platformu. Time rastu troškovi razvoja i održavanja aplikacije i samim time je proizvod teže prodati zato što mu je cijena veća zbog povećanih troškova. Uz to, raste i potreba za koordinacijom i usklađivanjem razvojnih timova na pojedinim platformama, kao i mogućnost pojavljivanja grešaka pri samom razvoju. Iz navedenih razloga nastala je potreba za razvojem novih programskih rješenja i tehnologija koje će se moći izvršavati na više različitih platformi bez potrebe da programeri poznaju zamršene detalje razvoja tehnologije za svaku od tih platformi.

Tu na scenu stupaju radni okviri za razvoj višeplatformskih korisničkih sučelja. Tehnologije za višeplatformski razvoj (eng. *Multiplatform Development*) postale su jedne od glavnih fokusnih točaka razvoja budućih tehnologija i velike kompanije poput *Googlea*, *Facebooka* i *Microsofta* ulažu znatne količine resursa u razvoj istih. Razvojni inženjeri nastoje stvoriti aplikacije koje se mogu besprijekorno izvoditi na različitim platformama kao što su Android, iOS, Windows, macOS i web preglednici, bez potrebe za pisanjem i održavanjem zasebnih baza kodova (eng. *codebase*)[1] za svaku od njih. Ovaj pristup ne samo da štedi na vremenu i resursima potrebnim za razvoj aplikacija, već osigurava i konzistentno korisničko iskustvo na svim uređajima i platformama.

Primarni fokus ovog diplomskog rada je proučavanje višeplatformskog radnog okvira *Compose Multiplatform* koji je nedavno stupio na "višeplatformsku scenu". *Compose Multiplatform*[2] je moderni deklarativni višeplatformski radni okvir za razvoj korisničkog sučelja razvijen od tvrtke *JetBrains* sa sjedištem u Pragu. Izgrađen je na temeljima

Jetpack Composea[3], modernog alata za izradu nativnog Android sučelja. *Compose Multiplatform* proširuje ovu funkcionalnost na druge platforme, omogućujući razvojnim programerima korištenje jedne baze kodova za izradu aplikacija za Android, iOS, desktop i web.

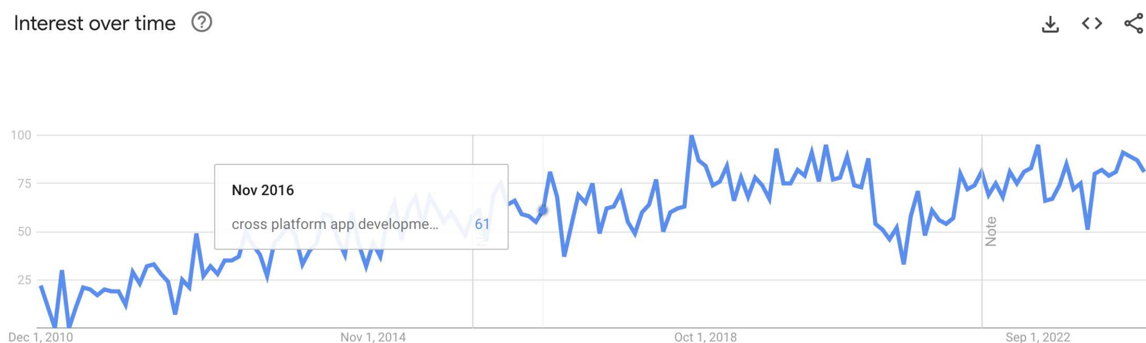
Kako bi mogli objektivno procijeniti trenutno stanje i mogućnosti razvoja u *Compose Multiplatformu*, usporedit ćemo ovaj radni okvir s drugim radnim okvirom razvijenim od *Googlea*, a to je *Flutter*[4]. Za razliku od tek nedavno razvijenog *Compose Multiplatforma*, *Flutter* je dobro uspostavljen i zreo radni okvir koji se kroz godine pokazao kao odličan odabir tehnologije za razvoj višeplatformskih aplikacija. Taj je status stekao zahvaljujući svom bogatom skupu unaprijed definiranih elemenata korisničkog sučelja (eng. *widget*), brzom razvojnom ciklusu i mogućnosti isporuke aplikacija visokih performansi.

Ovaj diplomski rad ima za cilj pružiti sveobuhvatnu usporedbu između *Compose Multiplatforma* i *Fluttera*, s primarnim fokusom na prvom radnom okviru. Udubit će se u temeljne tehnologije, arhitekturu i procese razvoja aplikacija oba okvira. Nudeći detaljnu analizu i praktične uvide, ovaj će rad poslužiti kao vodič programerima za procjenu potencijala i praktičnosti *Compose Multiplatforma* za razvoj svojih aplikacija.

2. Višeplatformski radni okviri

Kao što je već bilo rečeno u uvodu, višeplatformski radni okviri postali su jedan od najpopularnijih alata za izradu novih aplikacija. To se pogotovo odnosi na mobilne platforme gdje su uređaji najčešće sličnih dimenzija tako da se na svaki uređaj može primijeniti isti dizajn, ali zbog specifičnosti uređaja i operacijskih sustava koji se na njima nalaze, može doći do komplikacija pri implementaciji iste funkcionalnosti na različitim uređajima. Tu u igru stupaju višeplatformski radni okviri koji nude programerima mogućnosti pisanja aplikacija u jednoj bazi koda koja će se moći izvršavati na više različitih uređaja i operacijskih sustava (jednom riječju, platformama). Ovi okviri apstrahiraju detalje specifične za platformu, omogućujući programerima da jednom napišu kod i implementiraju ga u različitim okruženjima. Glavne prednosti ovog pristupa uključuju smanjene troškove razvoja i održavanja, brži izlazak na tržište i jedinstveno korisničko iskustvo na svim platformama.

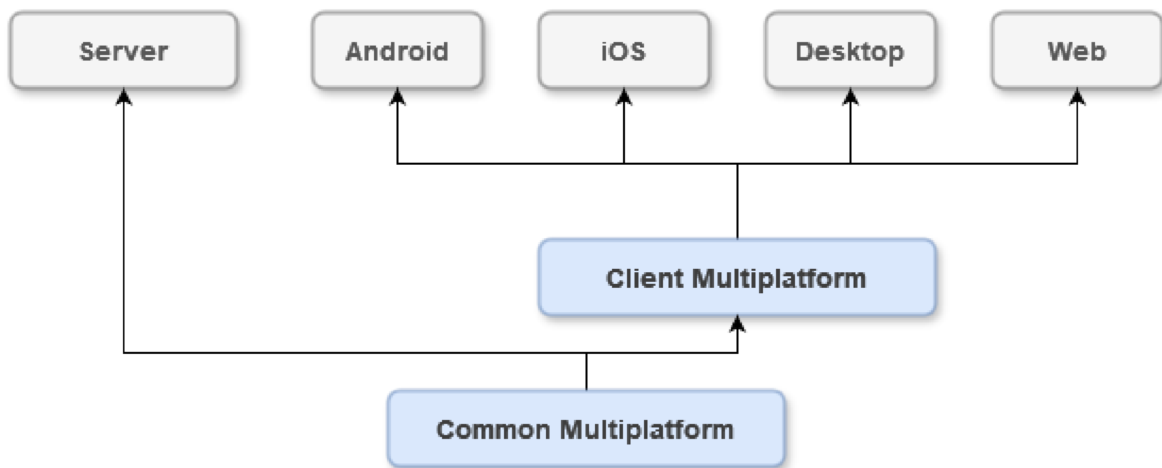
Iz navedenih razloga, logično je za očekivati da će interes za višeplatformske tehnologije i razvoj istih biti u konstantnom porastu kako na tržište izlazi sve veći broj novih uređaja i specifičnih operacijskih sustava. Od pojave prvih modernih višeplatformskih radnih okvira oko 2010. godine pa sve do danas, može se vidjeti stalni blagi porast interesa za višeplatformske tehnologije kao što je prikazano na slici 2.1.



Slika 2.1. *Google Trends* graf interesa za višeplatformske tehnologije

Rastuća popularnost višeplatformskih tehnologija rezultirala je pojavom mnogih novih alata na tržištu, svaki od kojih ima svoj set specifičnosti, a samim time i niz prednosti i mana. Iz tog razloga, neupućenim programerima, a i njihovim iskusnijim kolegama, može biti zahtjevno odabrati koji radni okvir je najbolje koristiti za aplikaciju koju žele razviti. Neki od popularnih višeplatformskih radnih okvira za razvoj korisničkih sučelja koji su u zrelom stadiju razvoja i koriste se već dugi niz godina su *Flutter*, *React Native*[5], *Ionic*[6] i *NET Multi-platform App UI*[7] (skraćeno *.NET MAUI*, evolucija prijašnjeg *Xamarina*[8]).

Uz navedene dugostojeće višeplatformske radne okvire, na tržište se plasiraju i novi okviri kojima je cilj nadmašiti konkurenciju. Jedan od njih je i *Kotlin Multiplatform*[9] (skraćeno KMP) koji je razvijen od *JetBrainsa* i koji omogućuje programerima stvaranje aplikacija za različite platforme i učinkovitu ponovnu upotrebu koda na njima uz zadržavanje svih prednosti nativnog programiranja. To je ujedno i najveća jedinstvena prodajna točka *Kotlin Multiplatforma*. Programeri imaju veliku fleksibilnost pri razvoju aplikacija u tom radnom okviru jer se mogu odlučiti na korištenje KMP-a samo u dijelu koda koji je zajednički svim platformama i zadržati implementacije korisničkih sučelja na nativnim stranama ili mogu odabrati potpuni prelazak na dijeljenje koda gdje će onda i kod za izgled korisničkog sučelja biti u okviru višeplatformske tehnologije.



Slika 2.2. Struktura projekta u *Kotlin Multiplatformu*

Tu u priču ulazi *Compose Multiplatform*. Ako se razvojni inženjeri odluče na korištenje *Kotlin Multiplatforma* s dijeljenim korisničkim sučeljem, to korisničko sučelje mora biti pisano u *Jetpack Composeu*, a cjelokupni radni okvir koji se koristi onda se zove *Compose Multiplatform*. U sljedećem odlomku ćemo detaljnije objasniti što je to *Compose Multiplatform*, kada i zašto je nastao i kakvo je trenutno stanje razvoja tog višeplatformskog radnog okvira.

2.1. Compose Multiplatform

Iz prethodnog odlomka mogli smo za zaključiti da *Compose Multiplatform* nije ništa drugo nego proširenje KMP-a koje koristi zajednički kod za oblikovanje korisničkog sučelja uz kod zajedničke logike koji se nalazi ispod u arhitekturi aplikacije. *Compose Multiplatform* je zapravo evolucija *Jetpack Composea* kojeg je Google predstavio 2019. godine i koji je ušao u produkciju 2021. godine. Od tada je *Compose* preporučeni alat za razvoj korisničkog sučelja od strane *Googlea* i veliki se napor ulaže u njegov razvoj. *Jetpack Compose* imao je za cilj pojednostaviti i ubrzati razvoj korisničkog sučelja na Androidu korištenjem modela deklarativnog programiranja[10].

Nadovezujući se na ovaj uspjeh, *JetBrains* je proširio *Compose* kako bi podržao više platformi, što je rezultiralo *Compose Multiplatformom*. Najavljen 2020., *Compose Multiplatform* koristi *Kotlin*, programski jezik poznat po sažetoj sintaksi i interoperabilnosti s *Javom*. Ovaj radni okvir još je uvijek u ranoj fazi, ali obećava u pružanju jedinstvenog

razvojnog iskustva na različitim platformama. Okvir trenutno podržava sve platforme, iako su stupnjevi razvoja na svakoj platformi dosta različiti. Android i desktop platforme su trenutno jedine stabilne, dok je iOS nedavno prešao iz alpha u beta verziju, a Web iz eksperimentalne u alpha verziju. U narednim poglavljima bit će još veći fokus na svakoj od platformi *Compose Multiplatforma* i njihovom stupnju razvoja.

2.2. Flutter

Flutter je višeplatformski radni okvir otvorenog koda kojeg je *Google* predstavio 2017. godine. *Flutter* se temelji na programskom jeziku *Dart*, jeziku koji je također razvio *Google* i koji je odabran zbog svojih brzih performansi, skalabilnosti i mogućnosti prevođenja koda unaprijed (eng. *Ahead-of-Time*, AOT). U poglavlju 3.4. bit će detaljnije opisana sintaksa i značajke *Darta*.

Od svog stupanja na IT scenu, *Flutter* je brzo postao popularan među programerima i korišten je za izradu širokog spektra aplikacija, od startupa do rješenja na razini poduzeća. Programere koji su upoznati s karakteristikama *Fluttera* to ne bi uopće trebalo čuditi jer *Flutter* odlikuje niz prednosti u odnosu na tradicionalne alate za izradu aplikacija. Neke od glavnih prednosti su:

- Jednostavnost: *Flutter* je relativno lagan radni okvir za učenje i korištenje, čak i za programere koji nemaju prethodnog iskustva s razvojem mobilnih ili drugih aplikacija. To ne znači da ne postoji krivulja učenja, ali nije potrebno prethodno znanje *Darta* ili razvoja mobilnih aplikacija za započeti učenje *Fluttera*.

- *Hot Reload*: Jedna od glavnih prodajnih točaka *Fluttera* je mogućnost *Hot reloada*, što zapravo znači da programeri mogu vidjeti promjene u svom kodu pri spremanju koda dok se izvršava aplikacija, bez potrebe za ponovnim pokretanjem aplikacije. Ova značajka ubrzava razvoj aplikacije jer zaobilazi dugotrajan proces ponovnog pokretanja procesa izgradnje aplikacije koja nekad može trajati i minutama, npr. za Android aplikacije koje se grade s pomoću alata *Gradle*.

- Bogata kolekcija komponenti: *Flutter* nudi široku paletu ugrađenih komponenti UI-a, što uvelike olakšava izradu atraktivnih i funkcionalnih aplikacija i skraćuje vrijeme razvoja koje bi inače bilo potrošeno na pisanje vlastitih komponenti.

- Dokumentacija i zajednica: Jedna od ključnih značajki svakog alata za razvoj aplikacija je dobro napisana i opsežna dokumentacija. Dokumentacija pomaže početnicima, a i njihovim iskusnijim kolegama da nađu informacije koje su im potrebne za razvoj. Također, zbog velikog broja korisnika ovog radnog okvira od kojih isto tako veliki broj već dugi niz godina radi u *Flutteru*, nastala je aktivna zajednica programera koji održavaju svoje komponente otvorenog koda, dijele ih sa ostatkom zajednice i aktivni su na forumima na kojima programeri mogu postaviti pitanja o *Flutteru* kada naiđu na poteškoće u razvoju.

Sve navedene značajke *Fluttera* čine ga danas jednim od najpopularnijih radnih okvira za višeplatformski razvoj aplikacija, ali i jednim od najpopularnijih radnih okvira općenito. Iz tog razloga je upravo *Flutter* izabran kao radni okvir s kojim će se usporediti *Compose Multiplatform*.

3. Korištene tehnologije

3.1. Kotlin

Kotlin[11] je moderan, statički tipizirani programski jezik koji je stekao značajnu popularnost u zajednici softverskih programera od svog izdanja od strane *JetBrainsa* 2011. godine. Karakterizira ga njegova potpuna interoperabilnost s programskim jezikom *Java*. JVM verzija *Kotlinove* standardne biblioteke ovisi o Java Class Library.

3.1.1. Podrijetlo i razvoj

JetBrains je započeo s razvojem *Kotlina* u 2010. godini, a prvotno su ga objavili pod imenom *Project Kotlin* u srpnju 2011. godine. Trebali su jezik koji je sažet, elegantan, izražajan, a također interoperabilan s *Javom*, budući da je većina njihovih proizvoda razvijena u *Javi*, uključujući najpopularniji *Intellij IDEA*. Također, jedan od glavnih zahtjeva je bio da sve navedene promjene ne utječu na performanse samog jezika, već da one ostanu slične *Javinim* performansama.

JetBrains je otvorio kod za projekt pod licencom *Apache 2* u veljači 2012. godine kako bi još više potaknuo razvoj jezika i stvorio zajednicu. Prva verzija zvana *Kotlin 1.0* objavljena je u veljači 2016. godine te se ta verzija smatra prvim službenim izdanjem. S verzijom 1.2 izašla je prva podrška za višeplatformsko programiranje s opcijom dijeljenja koda između JVM i *Javascript* platformi, a s verzijom 1.4 donesene su velike promjene podrške za *Appleove* platforme, tj. za interoperabilnost s jezicima *Objective C* i *Swift*.

3.1.2. Ključne značajke

Kako je već bilo rečeno u prethodnom potpoglavlju, *Kotlin* po svojoj strukturi jako je vezan s *Javom*, a izgledom podsjeća na hibrid *Scale* i *Jave*. U ovom potpoglavlju bit će dokumentirane neke od glavnih značajki programskog jezika *Kotlina* koje ga čine specifičnim i neke koje ga razlikuju od ostalih programskih jezika. Uz to, bit će naveden niz usporedbi s *Javom* i opisane sličnosti i razlike *Kotlina* s matičnim programskim jezikom. Također, ovo bi potpoglavlje trebalo pomoći osobama koje čitaju ovaj diplomski rad da dobiju uvid u prednosti korištenja *Kotlina* u razvoju svojih aplikacija i kako pomoću njega mogu poboljšati kvalitetu svog koda i skratiti vrijeme potrebno za pisanje istog koda u drugim programskim jezicima.

S obzirom na to da je cilj razvoja *Kotlina* bilo napraviti programski jezik koji će biti "bolji jezik" od *Jave*, a opet jezik koji je skroz interoperabilan s *Javom*, nije ni čudo da *Kotlin* svojom strukturom i sintaksom jako podsjeća na *Javu*. Programeri koji prelaze s korištenja *Jave* na korištenje *Kotlina* neće imati velikih problema naviknuti se na sintaksu *Kotlina*, osim što će morati pripaziti na par sitnica u kojima se ta dva programska jezika razlikuju. Prvi primjer koji je odmah uočljiv svim programerima *Jave* je taj da u *Kotlinu* nije obavezno korištenje znaka točke sa zarezom (`;`) kao terminatora naredbe, ali to je tek prva od niza razlika između ova dva jezika. Te razlike će biti dokumentirane i opisane u narednim potpoglavljima koja će biti grupirana po logičkim cjelinama. Dijelovi koda korištenog u primjerima preuzeti su sa stranice *Baeldung*[12].

Varijable

Jedna od prvih razlika koje će programeri primjetiti je da se tip variable u *Kotlinu* ne mora eksplicitno naznačiti, već se koristi zaključivanje tipa (eng. *type inference*), osim u slučajevima gdje se tip ne može zaključiti implicitno ili da se pomogne drugom programeru koji čita kod da mu odmah bude jasno kojeg je tipa ta varijabla. Varijable u *Kotlinu* mogu biti samo za čitanje (eng. read-only) deklarirane ključnom riječi `val`, ili promjenjive, deklarirane ključnom riječi `var`, dok se konstante definiraju ključnom riječi `const`. Tip varijable definira se nakon naziva varijable odvojen sa znakom dvotočke (`:`). Primjeri definiranja varijabli i konstanti pokazan je u sljedećem primjeru koda.

Kod 3.1. Definicija varijabli

```
const MINUTES_IN_HOUR = 60
val message = "Hello World"
val floatVariable: Float = 24f
var intVariable = 5
// Postavljanje ove Integer variable na Long vrijednost baca gresku
intVariable = 12L
```

U *Kotlinu* sve varijable mogu biti nulabilnog tipa, ili drugim riječima, može im se postaviti *null* vrijednost. Kako bi neku varijablu označili kao nulabilnu, mora se staviti znak upitnika (`?`) nakon oznake tipa varijable. Pri svakom izvršavanju operacije nad nulabilnim objektom mora biti posvećena posebna pažnja kako ne bi došlo do greške. *Kotlinov* prevodilac ne dozvoljava da se nad nulabilnim objektima obavljaju iste operacije kao nad njihovim nenulabilnim pandanima. Stoga se pri svakom pristupu nulabilnom objektu i njegovim atributima mora izvršiti tzv. null provjera (eng. *null-check*). Null provjera se obavlja eksplicitno postavljanjem operatora uskličnik (`!`) nakon oznake tipa s čime se riskira dohvaćanje null vrijednosti i pogreške pri izvršavanju aplikacije ili tako da se koristi jedan od *Kotlinovih* null sigurnih (eng. *null-safe*) operatora:

- Operator `?.` (operator sigurne navigacije): Koristi se za siguran pristup metodi ili atributu objekta nulabilnog tipa. U slučaju da je vrijednost objekta null, metoda se neće pozvati i cijeli izraz će se protumačiti kao null.

- Operator `?:` (operator spajanja null vrijednosti): Koristi se za tumačenje izraza koji može imati null vrijednost. Ako vrijednost nije null, gleda se operand s lijeve strane izraza, a inače operand s desne strane izraza. Ovaj se operator često zove i *Elvisovim operatorom* jer podsjeća na emotikon Elvisa Presleya.

Poseban operator je operator `as?` (operator sigurne pretvorbe) koji se koristi za sigurnu pretvorbu objekta iz jednog tipa u drugi pri čemu, ako tipovi nisu kompatibilni, ne dolazi do nastupanja *ClassCastException* greške, već se vrijednost te varijable postavi na null.

Primjer navedenih svojstava nulabilnosti prikazan je u sljedećem kodu. Neki od dijelova koda bit će objašnjeni u sljedećim potpoglavljima; ovdje je primarni fokus na nulabilnosti varijabli.

Kod 3.2. Nulabilnost varijabli

```
// Primjer nulabilne i nenulabilne varijable
var nonNullableVariable: String = "Hello World"
var nullableVariable: String? = null
// Nije dozvoljeno
nonNullableVariable = null

// Pristup atributima nulabilnog objekta Car
class Wheel(
    val size: Int
)
class Car(
    val wheels: List<Wheel>
)

var car: Car? = Car(wheels = listOf(Wheel(size = 12), Wheel(size = 12)))

// Koristi se operator ?. za dohvacanje atributa "wheels" od objekta tipa "Car"
fun getFirstWheelSize() = car?.wheels?.first()?.size

// Koristi se operator ?: kako bi se dobila nenulabilna lista kotaca od auta
// U slucaju da je auto vrijednosti null, vraca se prazna list
fun getCarWheels(): List<Wheel> = car?.wheels ?: emptyList()

// Nije moguce pretvoriti auto u kucu,
// stoga se vrijednost varijable "house" postavlja na null
val house: House? = car as? House
```

Klase i sučelja

Kako smo već ustanovili da je *Kotlin* objektno orijentirani jezik, za očekivati je da je veliki fokus pri razvoju jezika bio na osmišljavanju jednostavnog i efikasnog načina rukovanjem klasama i sučeljima. Stoga je u *Kotlinu* prezentiran niz noviteta u odnosu na tradicionalno shvaćanje klasa u *Javi*.

Ako se ne navede drugačije, sve definirane klase su finalne, što znači da se iz njih ne mogu kreirati nove klase koje ih nasljeđuju, osim ako se ne navede ključna riječ `open` prije definiranja klase. Isto tako, svi članovi klasa su po definiciji javni i nije potrebno navođenje ključne riječi `public`, a vidljivost klase ili nekog člana klase može se ograničiti ključnim riječima `private`, `protected` ili `internal`. Uz to, kao i u *Javi* postoje apstraktne klase koje su označene ključnom riječi `abstract` i koje su otvorene po standardnim postavkama.

Kod 3.3. Vidljivost klasa i nasljeđivanje

```kotlin
// Teniski igrac ima dodatne atribute koje obicni igrac nema
class TennisPlayer(
    id: Int,
    name: String,
    val isRightDominantHand: Boolean
): Player(id, name) {
    // Uz ime teniskog igraca pise i oznaka njegove dominatne ruke
    override fun playerDisplay(): String {
        return "$name ${if (isRightDominantHand) "R" else "L"}"
    }
}

open class Player(
    val id: Int,
    val name: String
) {
    open fun playerDisplay(): String {
        return name
    }
}
```

Još jedan tip klase i sučelja koji može biti iznimno koristan u mnogim situacijama je tzv. *Sealed* klasa, tj. sučelje. *Sealed* klase i sučelja ograničavaju hijerarhije podklasa, što za posljedicu programerima donosi veću kontrolu nad hijerarhijom nasljeđivanja. Takav tip klase strukturom podsjeća na enumeracije i koristan je kada trebamo definirati određeni set klasa koje mogu postojati, a svaka od njih zahtjeva postojanje posebnih atributa. Primjer u kojem se mogu koristiti *Sealed* klase dan je u sljedećem kodu.

Kod 3.4. *Sealed* klasa

```kotlin
// Dijalog predstavlja komponentu sucelja koja se prikazuje preko ostalog sadrzaja
sealed class Dialog : ViewState() {
    // Oznacava da nije prikazan nijedan dijalog
    class None : Dialog()
    // Prikazan je djialog za brisanje stavke u listi s atributom
    // koji je identifikator te stavke
    class DeleteItem(val itemId: Long) : Dialog()
}
```

Na kraju dolazimo do posebnosti *Kotlina* koja je jedinstvena za taj jezik, a to su *Data klase*. U *Javi* se od svake klase očekuje da ima implementiran set osnovnih metoda kako bi klasa mogla funkcionirati s metodama i klasama koje pruža *Javina* standardna bibli-

oteka. Budući da je implementacija ovih metoda ručno ili s vanjskim alatima glomazna i često dovodi do ponovljenog ili šablonskog koda, postavlja se pitanje kako smanjiti sav taj šablonski kod. Tu u priču ulaze *Data* klase čija je osnovna svrha pohranjivanje podataka i koja je strukturom slična normalnim klasama osim što se ključne metode klasa `toString`, `equals` i `hashCode` automatski generiraju iz atributa klase. *Java* je u međuvremenu dodala svoju implemetaciju ove značajke koja se zove *Record* i dostupna je od verzije 14.

Objekt i pridruženi objekt

U *Kotlinu* kao i u svim drugim programskim jezicima koji se oslanjaju na JVM, koncept klase je u srži modela objektno orijentiranog programiranja. Međutim, *Kotlin* uvodi jedan novi koncept povrh toga. Posebna struktura podataka u *Kotlinu* je struktura koja se zove *Objekt* (eng. *Object*) i ima vrlo specifično ponašanje koje je moguće zaključiti iz samog imena strukture. Dok klasa opisuje strukture koje se mogu instancirati kako i kada programer želi i dopušta onoliko instanci koliko je potrebno, objekt umjesto toga predstavlja jednu statičnu instancu i nikada ne može imati više ili manje od ove jedne instance. Time je ponašanje *Objekta* vrlo slično oblikovnom obrascu *Singleton*. Objekti također nude punu podršku za modifikatore vidljivosti, omogućujući skrivanje podataka i enkapsulaciju kao i kod bilo koje druge klase. Također, objekti mogu nasljeđivati druge klase i implementirati sučelja čineći ih *singleton* primjercima roditeljskih klasa. Takvo ponašanje može biti vrlo korisno u slučajevima kada imamo implementaciju bez stanja i nema potrebe za stvaranjem novih primjeraka pri svakom pozivu, npr. neke funkcije unutar objekta.

Kod 3.5. Objekt koji nasljeđuje drugu klasu

```kotlin
object ReverseStringComparator : Comparator<String> {
    override fun compare(o1: String, o2: String)
        = o1.reversed().compareTo(o2.reversed())
}

val strings = listOf("Hello", "World")
// Comparator ce uvijek biti isti gdje god ga proslijedimo kao argument
val sortedStrings = strings.sortedWith(ReverseStringComparator)
```

Poseban tip objekta koji je samo varijacija na temu je tzv. *Companion Object* ili u prijevodu Popratni objekt. Popratni objekt služi kao "pomoćnik" nekoj klasi i u biti je isti

kao standardna definicija objekta, samo s nekoliko dodatnih značajki koje olakšavaju razvoj. Popratni objekt uvijek se definira unutar neke druge već postojeće klase i, iako može imati ime, ne treba ga imati i najčešće ga nema, već u tom slučaju automatski ima ime *Companion*. Popratni objekti dopuštaju pristup svojim članovima iz popratne klase bez navođenja naziva, a u isto vrijeme, vidljivim članovima se može pristupiti izvan klase kada ispred njih stoji naziv klase.

Glavna upotreba popratnih objekata je zamjena statičkih metoda/atributa poznatih iz *Jave*. Međutim, ta se polja ne generiraju automatski kao takva u rezultirajućoj datoteci klase. Ako nam je potrebno da ih generiramo, moramo staviti anotaciju `@JvmStatic` na traženu metodu/atribut koja će zatim generirati prikladni bajt kod za interoperabilnost s *Javom*.

Primjeri svih navedenih značajki popratnih objekata prikazani su u sljedećem kodu:

Kod 3.6. Definicija popratnog objekta

```
1  class OuterClass {
2      companion object {
3          // Stavljanjem anotacije je varijabla "publicVariable" prikladna
4          // za koristenje u Java kodu
5          @JvmStatic
6          val publicVariable = "You can see me"
7          // Nemoguce je pristupiti ovoj varijabli izvana
8          private val privateVariable = "You can't see me"
9      }
10     fun getSecretValue() = privateVariable
11 }
```

Proceduralno programiranje

Slično kao što je slučaj i u *Scali*, *Kotlin* je svojevrsni hibridni programski jezik koji kombinira pozitivne strane i objektno orijentiranog i proceduralnog programiranja. To radi na način da podržava sustav tipova na kojima počivaju koncepti proceduralnog programiranja, iako su, gledajući ispod haube, iste te funkcije u *Kotlinu* zapravo objekti.

Kotlinova sintaksa omogućuje definiranje lambda funkcija i funkcija najviše razine. Te su funkcije zapravo objekti, a sintaksa skriva da se zapravo na poziv funkcije poziva `invoke()` metoda na samom objektu. Kotlin također ima podršku za rekurzivne funkcije koje se sigurno izvršavaju na stogu, ali samo ako su te iste funkcije rekurzivne na

svojem kraju (eng. *tail-recursive*).

Iz navedenih razloga, funkcionalno programiranje u *Kotlinu* je mnogo lakše i tečnije nego u *Javi*, ali svejedno je relativno šturo i neambiciozno u odnosu na strogo proceduralne programske jezike. Svakako treba imati na umu internu implementaciju proceduralnog programiranja u *Kotlinu* koja ispod haube skriva objektno orijentiranu implementaciju kako ne bi došlo do neočekivanih ponašanja pri izvršavanju aplikacije.

Imenovani i zadani parametri

Jedan od nepraktičnih izazova rukovanja s funkcijama je slučaj kada funkcije imaju veliki broj parametara koje primaju. Pozivanje takvih funkcija rezultira teško čitljivim kodom, a može doći i do pogrešaka u kojima se poredaka parametara slučajno zamijeni i dovodi do krivog izvođenja funkcije. Isto tako, kako povećavamo broj izbornih parametara funkcije, pružanje zadanih vrijednosti putem preopterećenja funkcije (eng. *overloading*) također može postati glomazno. Tu u pomoć dolazi *Kotlinova* podrška za imenovane i zadane parametre.

Kada pozivamo neku funkciju u *Kotlinu*, možemo imenovati jedan ili više argumenata. Naziv koji se koristi za argument mora odgovarati nazivu parametra navedenom u deklaraciji funkcije. Poredak argumenata pri pozivu funkcije ne mora odgovarati onome u deklaraciji, dok god su eksplicitno navedeni sva parametri. Moguće je navesti i samo neke od argumenata, ali u tom slučaju je potrebno održati poredak kakav je naveden u deklaraciji funkcije.

Ponekad postoji slučaj kada funkcija želi tretirati neke od svojih parametara kao opcionalne i preuzeti zadane vrijednosti za njih ako nije navedena neka druga vrijednost pri pozivu funkcije. Mnogi programski jezici uključujući *Javu* ne podržavaju ovu značajku, već se mora ili eksplicitno napisati kod koji opcionalnom parametru zadaje vrijednost ili se koristi preopterećenje funkcija kako bismo osigurali različite verzije iste metode pri čemu svaka verzija pozivateljima omogućuje preskakanje jednog ili više izbornih parametara. Međutim, kako se broj parametara povećava, preopterećenje metode može brzo izmaknuti kontroli i dovesti do većih poteškoća u identificiranju značenja parametara. Srećom, *Kotlin* nudi podršku za zadane vrijednosti parametara navođenjem simbola `=` iza tipa parametra u deklaraciji funkcije.

Kod 3.7. Imenovani i zadani parametri

```
// Primjer funkcije sa zadanim parametrima
fun connect(url: String, connectTimeout: Int = 1000, enableRetry: Boolean = true) {
    println(
        "The parameters are url = $url,
        connectTimeout = $connectTimeout,
        enableRetry = $enableRetry"
    )
}

// Poziv funkcije gdje navodimo samo jedan od opcionalnih parametara
// Ovaj poziv dovodi do greske
connect("http://www.mywebpage.com", false)
// Potrebno je eksplicitno imenovati parametar
connect("http://www.mywebpage.com", enableRetry = false)
```

Funkcije proširenja

Jedna od zanimljivih i korisnih značajki *Kotlina* je da uvodi koncept funkcija proširenja (eng. *extension functions*). Taj tip funkcija predstavlja praktičan način proširenja postojećih klasa novom funkcionalnošću bez korištenja nasljeđivanja ili bilo kojeg tipa obrasca *Dekorator*. Nakon definiranja funkcije proširenja, istu možemo koristiti kao da je bila dio originalnog koda. Primjerice, ako moramo definirati funkciju koja vraća broj kotača na nekom prijevoznom sredstvu, u *Javi* bi morali definirati metodu koja prima taj automobil kao parametar i vraća broj kotača kao rezultat. Za razliku od toga, u *Kotlinu* jednostavno možemo napisati funkciju proširanja na originalnu klasu prijevoznog sredstva koja ne treba primati instancu objekta kao parametar, već se poziva nad tom istom instancom bez argumenata.

Kod 3.8. Funkcija proširenja

```
// Postojeca klasa prijevozno sredstvo s atributom kotacima
abstract class Vehicle(
    private val wheels: List<Wheel>
)

// Definicija funkcije prosirenja u Kotlinu
fun Vehicle.getNumberOfWheels(): Int {
    // Kljucna rijec "this" nije nuzna, ali moze posluziti u svrhu jasnoce koda
    return this.wheels.size
}

// Nacin koristenja funkcije u Javi
Int numberOfWheels = Vehicle.getNumberOfWheels(car);
// Funkcija prosirenja u Kotlinu
val numberOfWheels = car.getNumberOfWheels()
```

Ako je potrebno definirati funkciju proširenja na nekom generičkom tipu, *Kotlin* i to omogućava. Potrebno je samo definirati generički tip koji se može ponašati isto kao i konkretni tip dok god zadovoljava tip koji je zadan. Unutar bloka funkcije vrijednost *this* je sigurna po tipu. Primjer definiranja funkcije proširenja za generički tip dan je sljedećem kodu.

Kod 3.9. Funkcija proširenja generičkog tipa

```
// Ako se drugacije ne navede, genericki tip "T" nasljedjuje tip "Any"
// "T" je ovdje predan kao primatelj i parametar funkcije
fun <T> T.concatAsString(b: T) : String {
    // Potrebno je koristiti vrijednosti koje svi izvedeni tipovi generickog imaju
    return this.toString() + b.toString()
}
```

Funkcije opsega

Ostalo je opisati još jedan tip generičkih funkcija proširenja koji vrijedi za sve tipove podataka i koji je dio *Kotlinove* standardne biblioteke. Takve funkcije se nazivaju funkcije opsega (eng. *scope functions*) i njihova je jedina svrha izvršavanje bloka koda unutar konteksta objekta. Kada se pozove takva funkciju na objektu s danim lambda izrazom, ona tvori privremeni opseg u kojem se može pristupiti objektu bez njegovog eksplicitnog naziva. Time je izbjegnuto stvaranje nepotrebnih varijabli i ponavljanja njihovih imena, a samim time i kompliciranjem koda za ostvarivanje jednostavne funkcionalnosti.

Kod 3.10. Korištenje funkcije opsega

```
// Primjer koristenja funkcije opsega
Car("Mercedes", "John Doe", 2021).let {
    println(it)
    it.setOwner("Michael Porter")
    it.renewRegistration()
    println(it)
}

// Isti kod bez koristenja funkcije opsega
val mercedes = Car("Mercedes", "John Doe", 2021)
println(mercedes)
mercedes.setOwner("Michael Porter")
mercedes.renewRegistration()
println(mercedes)
```

Postoji 5 takvih funkcija: `let`, `run`, `with`, `apply` i `also`. U osnovi, sve ove funkcije izvode istu radnju: izvršavaju blok koda na objektu. Ono što je drugačije je kako ovaj objekt postaje dostupan unutar bloka i koji je rezultat cijelog izraza. Razlike između pojedinih funkcija objašnjene su u sljedećoj tablici.

Naziv	Referenca na objekt	Povratna vrijednost	Funkcija proširenja?
let	it	Rezultat lambde	Da
run	this	Rezultat lambde	Da
run	-	Rezultat lambde	Ne; zove se bez konteksta objekta
with	this	Rezultat lambde	Ne; prima kontekst objekta
apply	this	Objekt konteksta	Da
also	it	Objekt konteksta	Da

Tablica 3.1. Razlike između funkcija opsega

Korutine

Za kraj ovom poglavlju o *Kotlinu* valja spomenuti jednu od najmoćnijih i najistaknutijih značajki ovog jezika. Riječ je o korutinama (eng. *coroutines*), funkcijama koje su osmišljene da olakšaju pisanje jednostavnog, čitljivog i učinkovitog asinkronog koda.

Tradicionalno, rukovanje asinkronim zadacima oslanjalo se na povratne pozive (eng. *callbacks*), dretve (eng. *threads*) i funkcije obećanja (eng. *futures/promises*). Međutim, ove metode često dovode do složenog i teško čitljivog koda, poznatog kao *"callback hell"*.

Kako bi riješio te probleme, Kotlin je uveo korutine kao prvoklasnu značajku u verziji 1.3. Korutine omogućuju pisanje koda koji izgleda sinkrono, ali se izvršava asinkrono, čime se pojednostavljuje upravljanje asinkronim zadacima. Inspiracija za korutine dolazi iz jezika poput C# i Python, ali Kotlinova implementacija nudi jedinstvene prednosti i fleksibilnost korištenja.

Korutina je laka, neblokirajuća konstrukcija za upravljanje asinkronim kodom. Može se smatrati funkcijom koja se može pauzirati i nastaviti kasnije, omogućujući efikasno upravljanje vremenski zahtjevnim operacijama kao što su mrežni zahtjevi i ulazno-izlazne operacije.

Suspendirajuće funkcije (eng. *suspend functions*) su temelj korutina. Označene ključnom riječi `suspend`, ove funkcije mogu pauzirati svoje izvršavanje bez blokiranja dretvi i kasnije se nastaviti. Ključna riječ `suspend` signalizira da funkcija može biti pozvana unutar korutine ili druge suspendirajuće funkcije.

Kotlin pruža nekoliko načina za pokretanje korutina, od kojih su najčešće `launch` i `async`. Funkcija `launch` pokreće novu korutinu koja ne vraća vrijednost, dok `async` pokreće korutinu koja vraća rezultat putem `Deferred` objekta. Korutine se uvijek izvršavaju unutar određenog konteksta (eng. *context*) i područja (eng. *scope*). Kontekst pruža informacije poput dispečera (eng. *dispatcher*) koji određuje na kojoj dretvi će se korutina izvršavati. Primjeri dispečera su `Dispatchers.Main` za glavnu dretvu, `Dispatchers.IO` za ulazno-izlazne operacije i `Dispatchers.Default` za CPU-intenzivne zadatke.

Kod 3.11. Korutine

```kotlin
// Osiguravanje da se korutina izvrsi na pozadinskoj dretvi
val bgScope: CoroutineScope = CoroutineScope(Dispatchers.Default + SupervisorJob())

fun main() = bgScope.launch {
    val deferred1 = async { fetchDataFromServer1() }
    val deferred2 = async { fetchDataFromServer2() }

    val result1 = deferred1.await()
    val result2 = deferred2.await()

    println("Results: $result1 i $result2")
}

suspend fun fetchDataFromServer1(): String {
    delay(1000L) // Simulacija mreznog zahtjeva
    return "Data 1"
}

suspend fun fetchDataFromServer2(): String {
    delay(1000L) // Simulacija mreznog zahtjeva
    return "Data 2"
}
```

3.2. Gradle

Automatizacija izgradnje je praksa izgradnje softverskih sustava na relativno nenadziran način. Izrada je konfigurirana za izvođenje uz minimalnu ili nikakvu interakciju programera i bez upotrebe osobnog računala programera. *Gradle*[13] je jedan od najpopularnijih alata za automatizaciju izgradnje projekta, posebno u ekosustavu JVM jezika kao što su *Java*, *Kotlin* i *Groovy*[14]. Njegova fleksibilnost, učinkovitost i sposobnost da upravlja složenim projektima čine ga ključnim alatom za moderne razvojne timove.

Gradle je nastao 2007. godine kao odgovor na potrebe za fleksibilnijim i snažnijim alatom za izgradnju od tada dominantnih *Apache Ant* i *Apache Maven*. Hans Dockter, autor *Gradlea*, imao je viziju alata koji kombinira najbolje karakteristike *Anta* i *Mavena*, ali bez njihovih ograničenja. Prva verzija *Gradlea* službeno je objavljena 2009. godine. *Gradle* je brzo stekao popularnost zahvaljujući svojoj fleksibilnosti i učinkovitosti. Danas se koristi u mnogim industrijama i projektima, od malih timova do velikih korporacija. Za kontekst ovog rada bitno je naglasiti da je *Google* 2013. godine objavio da će *Gradle*

biti službeni alat za izgradnju Android projekata, što je značajno doprinijelo njegovoj popularnosti.

Gradle koristi skripte za definiranje procesa izgradnje. Ove skripte mogu biti napisane u *Groovyu* ili *Kotlin DSL*-u (skraćeno od *Domain Specific Language*). Glavna skripta za izgradnju je datoteka `build.gradle` ili `build.gradle.kts`, ovisno o odabranom skriptnom jeziku. Više riječi o samim konfiguracijama skripti bit će u poglavlju, o arhitekturi *Compose Multiplatform* projekata.4.2.

3.3. Ruby on Rails

Ruby on Rails[15], često nazivan samo *Rails*, je popularni web razvojni okvir otvorenog koda napisan u programskom jeziku *Ruby*[16]. Dizajniran da olakša i ubrza razvoj web aplikacija, Rails slijedi principe kao što su "Konvencija umjesto konfiguracije" (eng. *Convention over Configuration*) i "Ne ponavljaj se" (eng. *Don't Repeat Yourself - DRY*). Upravo zbog toga je brzo stekao popularnost u svijetu razvoja web aplikacija i od svoje prve verzije koja je izašla 2004. godine prošao je kroz mnoge verzije i unapređenja, postavši jedan od najkorištenijih web razvojnih okvira na svijetu.

Upravo zbog tih razloga je *Ruby on Rails* odabran kao radni okvir za razvoj poslužiteljske logike za ovu aplikaciju. *Ruby on Rails* je iznimno praktičan za brzi razvoj aplikacija, a kako za potrebe ove aplikacije nije potrebna kompleksna poslužiteljska logika, *Ruby on Rails* se pokazao kao savršeni odabir.

Više riječi o arhitekturalnim obrascima u *Ruby on Railsu* i implementaciji poslužiteljske logike bit će u poglavlju 4.1.

3.4. Dart

Dart[17] je moderan, objektno-orijentirani programski jezik razvijen od strane *Googlea*, dizajniran da zadovolji potrebe razvoja web i mobilnih aplikacija. *Dart* je stvoren s ciljem da bude brz, siguran i produktivan, pružajući programerima jednostavan način za izradu visokokvalitetnih aplikacija.

Dart je zamišljen kao jezik koji bi mogao poboljšati razvoj modernih web aplikacija

zamjenjujući ili nadopunjujući *JavaScript*, ali tijekom godina se razvio i prilagodio, postavši temeljni jezik za *Flutter* radni okvir koji omogućuje razvoj nativnih mobilnih i web aplikacija s jednom bazom koda. Iako je u početku naišao na pomiješane reakcije zajednice, njegov značaj je porastao s razvojem *Fluttera*. Danas, kombinacija *Darta* i *Fluttera* je popularan izbor za razvoj višeplatformskih aplikacija s rastućom zajednicom i podrškom velikih tvrtki.

Što se tiče nekih od ključnih značajki *Darta* kao programskog jezika, valja istaknuti da je *Dart* potpuno objektno-orijentirani jezik, što znači da su svi tipovi podataka, od jednostavnih tipova do složenih struktura, zapravo objekti. Ta značajka omogućava dosljedan i intuitivan način rada s podacima.

Dart koristi statičku tipizaciju, ali omogućava dinamičnu tipizaciju gdje je potrebno. Ovo pruža ravnotežu između sigurnosti tipova i fleksibilnosti. Statička tipizacija pomaže u otkrivanju pogrešaka u vrijeme kompajliranja, čime se smanjuje broj runtime pogrešaka.

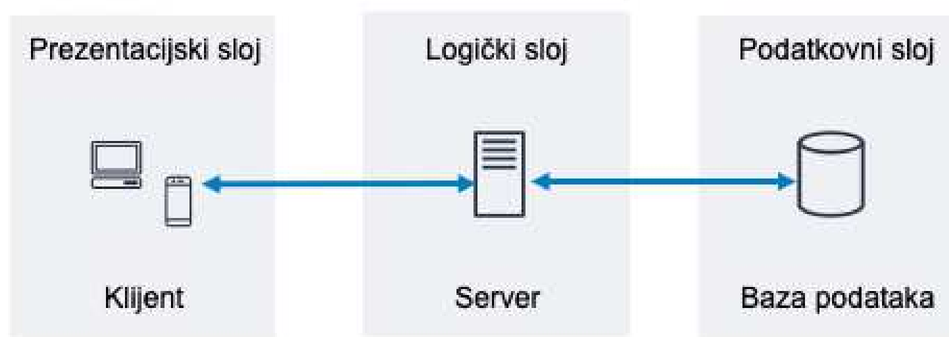
Također valja napomenuti da *Dart* ima ugrađenu podršku za asinkrono programiranje pomoću `Future` i `Stream` klasa. Ovo olakšava rad s vremenski zahtjevnim operacijama kao što su mrežni zahtjevi i ulazno-izlazne operacije. Primjer rukovanja s asinkronim podacima dan je sljedećem kodu:

Kod 3.12. Future klasa u *Dartu*

```
1  import 'dart:async';
2
3  void main() {
4    print('Pokretanje programa...');
5    fetchUserOrder();
6    print('Kraj programa.');
7  }
8
9  Future<void> fetchUserOrder() async {
10   print('Simulacija mreznog zahtjeva...');
11   await Future.delayed(Duration(seconds: 2));
12   print('Mrezni zahtjev zavrsen.');
13 }
```

4. Arhitektura aplikacije

U ovom poglavlju bit će detaljno opisana arhitektura demonstracijske aplikacije koja služi za pokazivanje velikog broja značajki razvoja aplikacije u *Compose Multiplatformu* i za usporedbu s razvojem iste aplikacije u *Flutteru*. Arhitektura aplikacije sastoji se od 3 dijela: baze podataka, poslužitelja i korisničkog sučelja. Opisana arhitektura naziva se troslojna arhitektura gdje baza podataka čini podatkovni sloj i zadužena je za pohranjivanje i dohvaćanje podataka, poslužitelj čini logički sloj i sadrži aplikacijsku logiku, poslovna pravila i obradu podataka, a korisničko sučelje čini prezentacijski sloj na kojemu korisnici vrše interakcije s aplikacijom. Jednostavan vizualni prikaz takve arhitekture dan je na slici 4.1.



Slika 4.1. Troslojna arhitektura

Ta arhitektura je odabrana jer služi za razdvanje odgovornosti u smislu da je svaki sloj neovisan o drugome i može se razvijati, upravljati i održavati zasebno, što arhitekturu čini modularnom i lakšom za upravljanje. Također, promjene u jednom sloju (kao što je ažuriranje korisničkog sučelja) ne moraju nužno utjecati na druge slojeve, a kako je cilj ovog rada demonstrirati upravo razvoj korisničkog sučelja, to čini ovu arhitekturu savršenim odabirom za aplikaciju.

U sljedećim potpoglavljima bit će opisan svaki od slojeva u detalje, od odabira tehnologije do arhitekture svakog sloja, a najveći fokus bit će na arhitekturi *Compose Multiplatforma*.

4.1. Poslužitelj i baza

Funkcija poslužitelja u sklopu ovog diplomskog rada je minimalna i služi kako bi aplikacija činila jednu skladnu cjelinu sastavljenu od korisničkog sučelja, baze podataka i poslužitelja, međutim za potrebe ovog diplomskog rada bili bi skoro dovoljni i lažni podaci. Jedina funkcija poslužitelja koja se nije mogla simulirati lažnim podacima je provjera spajanja na internet i dohvat podataka putem mrežnih zahtjeva.

Poslužitelj je implementiran u obliku REST servisa što znači da je svaka funkcionalnost izložena putem RESTful API-ja, omogućujući klijentima (prezentacijskom sloju) komunikaciju s poslužiteljem putem HTTP zahtjeva.

Sam servis na poslužitelju slijedi arhitekturalni obrazac *Model-View-Controller* (MVC) gdje svaka od tri navedene komponente ima svoju ulogu. *Model* predstavlja strukturu podataka i daje pogled u strukturu baze podataka. Kako ne bi morali sami pisati svoj SQL kod za pristup i manipulaciju podacima u bazi, *Ruby on Rails* nam pruža funkcionalnost koja se zove *Active Record*, a predstavlja ORM (*Object-Relational Mapping*) sloj *Railsa* koji omogućava jednostavno mapiranje objekata na relacijske baze podataka. Koristeći *Active Record*, programeri mogu raditi s bazama podataka koristeći *Ruby* objekte i metode, umjesto pisanja SQL koda.

Još jedna u nizu značajki koje ubrzavaju razvoj u *Railsu* su generatori koji automatski generiraju kod za modele, kontrolere, migracije i testove pomoću jedne naredbe unesene putem terminala.

Kod 4.1. Naredba generiranja modela *Student*

```
1 rails generate model Student firstname:string lastname:string
```

Pokretanjem prethodno navedene naredbe kreira se prikladna migracija u *Railsu*. Migracije u *Railsu* su dio *Active Recorda* i omogućuju jednostavno upravljanje promjenama u bazi podataka. One omogućuju programerima da definiraju promjene u strukturi baze

podataka pomoću *Ruby* koda, što olakšava praćenje i povrat promjena. Nakon generiranja svih migracija, potrebno je izvršiti te migracije i prebaciti bazu na sljedeću verziju upisom naredbe `rails db:migrate` u lokalni terminal.

Sljedeći sloj arhitekture poslužitelja je *View* koji je odgovoran za prikaz podataka korisniku. To bi u jednoj samostojećoj web stranici napravljenoj preko *Railsa* mogao biti prikaz HTML stranice preko kojeg bi korisnici komunicirali s poslužiteljem, međutim, budući da je ovo REST servis, pogledi su zapravo JSON odgovori koji se šalju klijentu. Za pretvorbu podataka iz baze koji se dohvaćaju u *Railsu* preko *Active Recorda* u JSON objekte koji se šalju korisniku, koristimo *Blueprinter*. *Blueprinter* je serijalizator za *Ruby* koji uzima objekte i rastavlja ih na jednostavne hashove te ih serijalizira u JSON. Dovoljno je da serijalizatoru kažemo koja polja i koje veze s drugim objektima želimo serijalizirati i on će obaviti sav posao za nas.

Kod 4.2. Serijalizator lekcija

```
1  class LessonSerializer < Blueprinter::Base
2    identifier :id
3
4    fields :name
5    fields :startDate
6    fields :endDate
7    fields :imageUrl
8
9    association :subject, blueprint: SubjectSerializer
10 end
```

Zadnji sloj arhitekture poslužitelja je *Controller* čija je zadaća obraditi dolazne HTTP zahtjeve, vršiti interakciju s modelima te vratiti odgovarajuće odgovore klijentima. Kontroleri mogu podržati sve standardne HTTP metode (GET, POST, PUT, DELETE) za interakciju s resursima. Implementacija REST servisa u *Ruby on Railsu* uključuje definiranje ruta, kontrolera i odgovarajućih metoda za rukovanje HTTP zahtjevima. Popis svih ruta definira se u datoteci `routes.rb` i sastoji se od mapiranja dolaznog zahtjeva na prikladnu metodu kontrolera kojemu je zadatak da rukuje s tim zahtjevom.

Kod 4.3. Primjer ruta u datoteci routes.rb

```
Rails.application.routes.draw do
  namespace :api do
    get "schools", to: "login#schools"
    get "studentLessons/:studentId", to: "lessons#student_lessons"
    get "getLesson/:lessonId", to: "lessons#get_lesson"
    get "lessonTimeline/:lessonId", to: "lessons#lesson_timeline"
    get "checkConnection", to: "connection#check_connection"
  end
end
```

Kod 4.4. Kontroler za dohvacanje i prikaz lekcija

```
class Api::LessonsController < ApplicationController
  def student_lessons
    classroom = Student.find(params[:studentId]).classroom
    teachers = classroom.teachers
    lessons = Lesson.where(teacher_id: teachers.map(&:id))
    render json: LessonSerializer.render(lessons)
  end

  def get_lesson
    lesson = Lesson.find(params[:lessonId])
    render json: LessonSerializer.render(lesson)
  end

  def lesson_timeline
    timeline_entries = TimelineEntry.where(lesson_id: params[:lessonId])
    render json: TimelineEntrySerializer.render(timeline_entries)
  end
end
```

4.2. Compose Multiplatform

Prije nego što bude opisana sama arhitektura projekta napravljenog u *Compose Multiplatformu*, ovo poglavlje dotaknut će se nakratko kreiranja i postavljanja projekta u ovoj tehnologiji kako bi čitatelju bilo jasno koji su to koraci potrebni za kreiranje projekta u *Compose Multiplatformu* i koji alati su potrebni za razvoj aplikacije.

4.2.1. Kreiranje i struktura projetka

Prije samog kreiranja projekta potrebno je instalirati odgovarajuće alate za razvoj. Ti alati su *Android Studio*[18], jednu od verzija *Jave* (preporuća se najnovija stabilna ver-

zija), *Kotlin Multiplatform Plugin* koji je dodatak za *Android Studio* i verziju *Kotlina* koja je kompatibilna s *Kotlin Multiplatform Pluginom*. Također, ako će se razvijati i aplikacija za web preglednik u sklopu projekta, potrebno je instalirati web preglednik koji ima podršku za *Web Assembly Garbage Collection*[20]. Ako se za razvoj koristi računalo s *macOS* operacijskim sustavom, može se provjeriti jesu li ispravno instalirani svi alati pokretanjem alata *KDoctor*. Dovoljno je instalirati alat pomoću *Homebrew* upravitelja paketa naredbom `brew install kdoctor` i pokrenuti analizu upisom naredbe `kdtoctor` u terminal.

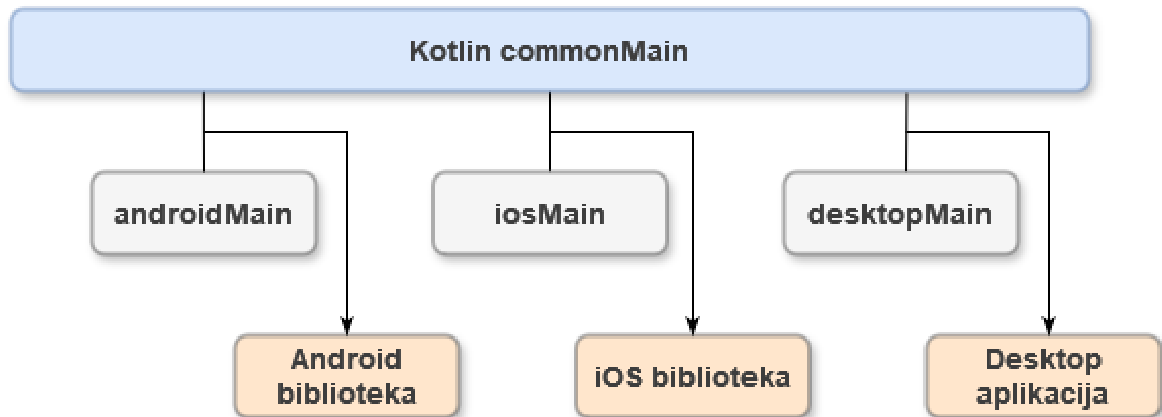
Nakon uspješne instalacije svih potrebnih alata, može se nastaviti s inicijalizacijom projekta. Najlakši način za stvaranje novog višeplatformskog projekta je uz korištenje čarobnjaka[21] za *Kotlin Multiplatform*. U čarobnjaku moguće je odabrati ime projekta, paket projekta i koje su ciljane platforme za razvoj. Također, moguće je odabrati neke od predložaka koji opisuju najčešće kreirane projekte. Za kraj je potrebno stisnuti na gumb za preuzimanje komprimiranog projekta koji se kasnije treba raspakirati u direktorij za razvoj.

Nakon što je projekt raspakiran u željeni direktorij, valja proučiti sadržaj projekta. Projekt sadrži dva modula:

- **composeApp:** Modul koji sadrži logiku koju dijele Android, iOS, desktop i web aplikacije, tj. kod koji se koristi na svim platformama. Obuhvaća poslovnu logiku, podatkovne modele i većinu korisničkog sučelja. Koristi *Gradle* kao sustav za automatsku izradu aplikacije.
- **iosApp:** Modul u obliku *Xcode* projekta koji se izgrađuje kao *iOS* aplikacija. Ovisi o dijeljenom modulu opisanom gore.

Modul `composeApp` sastoji se od niza skupova izvornog koda koji, ovisno o odabiru željenih platformi, mogu biti ovi: `androidMain`, `iosMain`, `desktopMain` i `wasmJsMain` zajedno uz skup `commonMain` koji je zajednički za sve tipove projekata i odabranih platformi. Skup izvornog koda je koncept koji opisuje niz logičko grupiranih datoteka po platormi gdje svaka grupa ima svoj niz ovisnosti koje su specifične za tu platformu, zajedno uz skup datoteka koje su zajedničke svim platformama. Skup izvornog koda `commonMain` koristi zajednički *Kotlin* kod, dok ostali skupovi koriste *Kotlin* kod specifičan za svaku

platformu. Kada se dijeljeni modul ugradi u Android biblioteku, zajednički *Kotlin* kod tretira se kao *Kotlin/JVM*. Kada je ugrađen u okvir *iOS*-a, uobičajeni *Kotlin* kod se tretira kao *Kotlin/Native*, dok kada se dijeljeni modul ugradi u web aplikaciju, zajednički *Kotlin* kod tretira se kao *Kotlin/Wasm*. Struktura opisana u prethodnim rečenicama vizualno je prikazana na slici 4.2.



Slika 4.2. Struktura *Compose Multiplatform* projekta

Kod 4.5. Dodavanje ovisnosti u *build.gradle.kts* datoteku

```
sourceSets {
    androidMain.dependencies {
        implementation(project.dependencies.platform(libs.compose.bom))
        implementation(libs.ktor.client.okhttp)
        implementation(libs.coroutines.android)
        // Ostale ovisnosti...
    }
    commonMain.dependencies {
        implementation(compose.runtime)
        implementation(libs.bundles.ktor)
        implementation(libs.kamel.image)
        implementation(libs.multiplatform.settings)
        // Ostale ovisnosti...
    }
    desktopMain.dependencies {
        implementation(compose.desktop.currentOs)
        implementation(libs.ktor.client.okhttp)
    }
    iosMain.dependencies {
        implementation(libs.ktor.client.darwin)
    }
}
```

Sada kad znamo kako inicijalizirati i kako izgleda strukutra jednog *Compose Multiplatform* projekta, možemo krenuti u opisivanje same strukture projekta i postavljanja svih potrebnih stvari koje bi trebale jednoj aplikaciji.

Ulazna točka projekta je *Composable* funkcija `App()` koja se implementira umetanjem na prikadan kod u svakoj platformi. Iz ove točke slažu se svi daljnji ekrani i cijeli izgled aplikacije. Svaka platforma implementira ovu funkciju na svoj način koji omogućuje da se aplikacija pokrene na svakoj platformi s istim izgledom i funkcionalnošću. Ulazni kod za pokretanje aplikacije za desktop platformu prikazan je u sljedećem isječku.

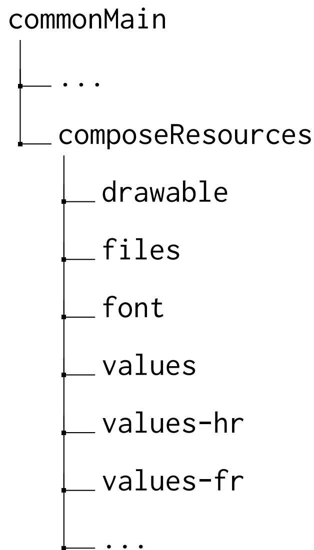
Kod 4.6. Ulazna točka u desktop aplikaciju

```
1 fun main() = application {
2     Window(onCloseRequest = ::exitApplication, title = "Diplomski Compose") {
3         App()
4     }
5 }
```

4.2.2. Resursi i stilovi

Cilj svake aplikacije je da bude vizualno privlačna i intuitivna korisniku za korištenje kako bi privukli nove korisnike i zadržali stare. Iz tog razloga su dizajn aplikacije i korisničko iskustvo u pravilu jedne od najvažnijih stvari o kojima treba razmišljati pri razvoju projekta. U ovom poglavlju opisat će se postavljanje osnovnih resursa i stilova koji će se koristiti kroz cijelu aplikaciju.

Compose Multiplatform nudi posebnu implementaciju za pružanje resursa kako bi bili dostupni za korištenje na svim platformama. Pod pojam resursi spadaju slike, fontovi i tekstovi koji se moraju prevoditi na različite jezike. Svaki od ovih resursa mora biti postavljen u odgovarajući direktorij unutar direktorija `composeResources` koja se nalazi u paketu `commonMain`. Također, svi tipovi resursa osim *"raw"* definiranih resursa u `files` direktoriju mogu imati više svojih varijanta ovisno o rezoluciji slika, tematskim bojama, fontovima i jezicima na koje se tekstovi prevode. Ispravna struktura direktorija prikazana je na slici 4.3.



Slika 4.3. Struktura direktorija za dijeljene resurse

Pri radu s resursima u *Compose Multiplatformu* valja obratiti pažnju na nekoliko stvari koje opisuju trenutne uvjete u ovom okviru. Tako je jedna od tih stvari da se gotovo svi resursi čitaju sinkrono u dretvi pozivatelja gdje su jedina iznimka *"raw"* datoteke i svi resursi na JS platformi koji se čitaju asinkrono. Također, čitanje velikih *"raw"* podataka, poput dugih videozapisa, u obliku toka podataka (eng. *stream*) još nije podržano, već se treba koristiti funkcija `getUri()` za prosljeđivanje zasebnih datoteka sistemskom API-ju, na primjer knjižnici *kotlinx-io*[22].

Za stilove i teme, pak, nije potrebna nikakva biblioteka, već se koriste standardne *Composeove* funkcije i klase. Moguće je definirati boje, tipografiju, temu i druge funkcionalnosti koje su potrebne jednoj aplikaciji. *Google* potiče korištenje boja i tema koje su u skladu s *Material*[23] smjernicama. Zadnja aktualna verzija *Material* smjernica je *Material3*. Postoji alat preko kojega je moguće generirati temu za aplikaciju, odabrati boje i fontove i generirati ih za niz tehnologija koje se podržane. Taj alat se zove *Material Theme Builder*[24] i, između ostalog, podržava generiranje tema za *Compose* projekte.

Nakon generiranja željenih boja i fontova, moguće ih je spremiti putem alata i prekopirati njihov sadržaj u datoteke `Color.kt` i `Typography.kt` koje se ručno generiraju. Boje dolaze u dvije sheme: svijetla i tamna. Svaka od tih paleta boja može se postaviti putem funkcija `lightColorScheme()` i `darkColorScheme()` s bojama koje su generirane putem alata. Tipografija se postavlja korištenjem klase `Typography` i

postavljanjem željenih tipova teksta. Nakon što su boje i tipografija postavljene, trebaju se kreirati tema aplikacije koja je `Composable` funkcija i koja poziva *Composeovu* `MaterialTheme()` funkciju s odgovarajućim parametrima. Postavljanje teme prikazano je u sljedećem isječku koda.

Kod 4.7. Tema aplikacije

```
@Composable
fun AppTheme(
    useDarkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable () -> Unit
) {
    val colors = if (!useDarkTheme) {
        LightColors
    } else {
        DarkColors
    }

    MaterialTheme(
        colorScheme = colors,
        typography = getTypography(),
        content = content
    )
}
```

Kasnije se u kodu, gdje god je to potrebno, mogu dohvatiti boje i tipografija korištenjem objekta *MaterialTheme* i dohvatom parametara `colors` ili `typography`.

4.2.3. Umetanje ovisnosti

Umetanje ovisnosti (eng. *Dependency Injection* - DI) je oblikovni obrazac koji pomaže u odvajanju ovisnosti unutar aplikacije, čineći kod lakšim za održavanje, testiranje i proširivanje. U Compose Multiplatform projektima, korištenje DI-a može značajno pojednostaviti upravljanje ovisnostima i poboljšati modularnost aplikacije. U sklopu ovog projekta odabran je okvir *Koin*[25] kao okvir za umetanje ovisnosti zbog njegove jednostavnosti i intuitivnosti, kao i lakog povezivanja sa samim *Compose Multiplatform* radnim okvirom i bibliotekom za navigaciju *Voyager* koje će biti opisana malo kasnije.

Za početak rada s *Koinom* potrebno je dodati prikladne ovisnosti u `build.gradle.kt` datoteku. Kako se u sklopu ovog radnog okvira koristi *Compose* kao alat za opisivanje korisničkog sučelja, tako moramo dodati verziju *Koina* koja podržava rad s *Composeom*.

Kod 4.8. Dodavanje ovisnosti *Koina*

```
implementation("io.insert-koin:koin-androidx-compose:$versionRef")
```

Koin odvaja ovisnosti na module, a svi moduli spremaju se u listu modula koji se predaju tzv. *KoinApplication* klasi koja predstavlja ulaznu točku aplikacije. U sklopu ovog projekta odvojili su se moduli za postavljanje mreže i HTTP klijenta, modul za repozitorij s popisom mrežnih zahtjeva i modul za *view modele* koji predstavljaju poslovnu logiku unutar aplikacije. Svi navedeni moduli spremaju se u listu `appModule()`.

Kod 4.9. *Koin* moduli u aplikaciji i primjer definicije modula

```
fun appModule() = listOf(
    provideHttpClientModule,
    provideRepositoryModule,
    provideViewModelModule,
    provideUtilModule
)

val provideViewModelModule = module {
    single<LoginViewModel> {
        LoginViewModel(get())
    }
    single<HomeViewModel> {
        HomeViewModel(get())
    }
    single<LessonViewModel> {
        LessonViewModel(get())
    }
}
```

Svaki od *view modela* prima `NetworkRepository` kao argument kroz konstruktor. *Koin* je dovoljno pametan da sam zaključi od kuda treba dohvatiti taj parametar i zaključi da postoji modul `provideRepositoryModule` koji pruža upravo repozitorij za mrežne zahtjeve. Također, svaki primjerak repozitorija zahtjeva `HttpClient` kao svoj parametar kojeg može primiti iz `provideHttpClientModule` modula. Sada postaje jasno da se implementacijom umetanja ovisnosti uvelike olakšala implementacija svih dijelova aplikacije i smanjila količina šablonskog koda koja bi se u suprotnom pojavila.

Kod 4.10. *KoinApplication* ulazna točka

```
@Composable
fun App() {
    KoinApplication(moduleList = { appModule() }) {
        AppTheme {
            ...
        }
    }
}
```

4.2.4. Navigacija

U trenutku pisanja ovog diplomskog rada još ne postoji stabilna verzija *Compose Multiplatforma* koja podržava nativnu *Compose* navigaciju koja se koristi u nativnim Android projektima. Kako se *Compose Multiplatform* brzo razvija, moguće je da će se u skorijem trenutku dodati i implementacija za nativnu *Compose* navigaciju, međutim vjerojatno će je pratiti male greške u implementaciji po platformama i razne druge dječje bolesti. Iz tog razloga programeri koji razvijaju *Compose Multiplatform* moraju pribjeći korištenju vanjskih biblioteka za navigaciju ili pokušati kreirati vlastitu navigaciju što se može pokazati kao iznimno zahtjevan zadatak.

Nadovezujući se na poglavlje o umetanju ovisnosti, u sklopu ovog projekta koristi se biblioteka za navigaciju koja ima mogućnosti iskoristiti sve pogodnosti umetanja ovisnosti i time uvelike pojednostaviti posao programerima. Navedena biblioteka zove se *Voyager*[26] i koristi pragmatičan pristup navigaciji koji uvelike olakšava razvoj aplikacija.

Voyager je višeplatformska navigacijska biblioteka izgrađena za *Jetpack Compose* koja je neprimjetno integrirana s njim i koja omogućuje kreiranje skalabilnih aplikacija unutar jednog prozora u kojemu se mogu izmjenjivati ekrani po potrebi. U *Voyageru* svaki ekran nasljeđuje baznu klasu `Screen` i mora implementirati apstraktnu metodu `Content()` koja je u stvari *Composable* funkcija i predstavlja sam izgled ekrana. Svaki ekran je stoga definiran kao klasa sa svojim argumentima koji se mogu proslijediti kroz konstruktor pri pozivanju ekrana u navigaciji.

Navigacija u *Voyageru* započinje definiranjem `Navigator` *Composable* funkcije koja prima početni ekran (ili listu ekrana) i tip tranzicija koje će se koristiti pri prijelazima iz

ekrana u ekran.

Kod 4.11. *Voyager* ulazna točka

```
@Composable
fun App() {
    KoinApplication(moduleList = { appModule() }) {
        AppTheme {
            ...
            Navigator(LoginScreen()) {
                FadeTransition(it)
            }
        }
    }
}
```

Za dohvaćanje navigatora i prijelaz na druge ekrane iz trenutnog ekrana, koristi se `LocalNavigator` koji se može dohvatiti iz *Composable* konteksta. Korištenjem lokalno dohvaćenog navigatora, dodavanje novog ekrana obavlja se putem metode `push()` ako želimo dodati novi ekran uz ostavljanje prethodnih ekrana na stogu ili putem metode `replace()` ako želimo sve prethodne ekrane pobrisati i ostaviti samo novi ekran. Za navigaciju na prethodni ekran na stogu koristi se metoda `pop()`.

Isto kao i s navigacijom, u trenutku pisanja ovog rada još ne postoji podrška za *view modele*, već programeri moraju razviti svoja rješenja ili koristiti vanjke biblioteke. Srećom, *Voyager* također nudi i pragmatičan pristup arhitekturi koja nudi ponašanje slično onome od *view modela* kakvo je prisutno na nativnoj Android platformi. *Voyagerova* implementacija *view modela* zove se `ScreenModel` i, poput *ViewModela*, dizajniran je za pohranjivanje i upravljanje podacima povezanim s korisničkim sučeljem, svjestan životnog ciklusa aplikacije. Također omogućuje podacima da prežive promjene konfiguracije kao što su rotacije zaslona. Za razliku od *ViewModela*, `ScreenModel` je samo sučelje koje treba implementirati i neovisan je o Android platformi te ne zahtijeva *Activity* ili *Fragment* za postojanje.

Kao što je već rečeno, *Voyager* nudi mogućnost interoperabilnosti s *Koin* okvirom za umetanje ovisnosti. Dohvaćanje *view modela* na ovaj način postaje vrlo jednostavno i obavlja se pozivanjem metode `getScreenModel()` s odgovarajućim *view modelom* kao parametrom. U pozadini te funkcije obavlja se dohvat *Koin* modula i traži se *view model* koji je predan kao parametar funkcije u modulu *view modela*.

Kod 4.12. Dohvat navigatora i *view modela*

```
data class HomeScreen(private val studentId: Int) : Screen {

    @Composable
    override fun Content() {
        val navigator = LocalNavigator.currentOrThrow
        val viewModel = getScreenModel<HomeViewModel>().also {
            it.getStudentLessons(studentId)
        }
        val lessonsData by viewModel.lessonsDataFlow.collectAsState()
    }
}
```

4.2.5. Mrežna komunikacija

U prethodnim poglavljima već je opisano da su HTTP klijent i repozitorij mrežnih zahtjeva implementirani kao moduli koji se putem umetanja ovisnosti mogu pružiti na svim potrebnim mjestima u aplikaciji, međutim još nije opisana sama implementacija klijenta i repozitorija. Za implemetaciju klijenta i repozitorija koristi se okvir *Ktor*[27], fleksibilni i moćni asinkroni alat za mrežnu komunikaciju razvijen od strane *JetBrainsa* koji pruža izvrsnu podršku za razvoj višeplatformskih aplikacija.

Prije samog početka rada s *Ktorom* potrebno je postaviti potrebne ovisnosti za ovaj alat u datoteci `build.gradle.kts`. S obzirom svaka platforma ima svoj način komunikacije s mrežom, potrebno je odvojiti ovisnosti po platformama.

Kod 4.13. Potrebne ovisnosti za *Ktor*

```
sourceSets {
    androidMain.dependencies {
        implementation(libs.ktor.client.okhttp)
    }
    commonMain.dependencies {
        implementation(libs.bundles.ktor)
    }
    desktopMain.dependencies {
        implementation(libs.ktor.client.okhttp)
    }
    iosMain.dependencies {
        implementation(libs.ktor.client.darwin)
    }
}
```

Nakon što su uspješno dodane sve ovisnosti, moguće je inicijalizirati HTTP klijent i postaviti rute korištene u mrežnoj komunikaciji. Pri inicijalizaciji HTTP klijenta potrebno mu je predati parametre o baznom URL-u s kojeg će se dohvaćati zahtjevi, protokol koji će se koristiti te port na kojem je otvoren servis. Također, treba definirati format sadržaja koji će se slati i primati sa servisa što je u našem slučaju JSON.

Kod 4.14. Inicijalizacija HTTP klijenta

```
val provideHttpClientModule = module {
    single {
        HttpClient {
            install(DefaultRequest) {
                url {
                    protocol = URLProtocol.HTTP
                    host = "diplomski-api.com"
                    port = 3000
                }
                accept(ContentType.Application.Json)
                contentType(ContentType.Application.Json)
                header("Cache-Control", "max-age=0")
            }

            install(ContentNegotiation) {
                json(
                    Json {
                        ignoreUnknownKeys = true
                        prettyPrint = true
                        explicitNulls = false
                    }
                )
            }
        }
    }
}
```

Nakon postavljanja HTTP klijenta još ostaje definirati zahtjeve na mrežu koji će se koristiti u aplikaciji. Definira se klasa `NetworkRepository` koja prima prethodno definirani HTTP klijent kao parametar u konstruktoru putem umetanja ovisnosti. Unutar repozitorija popisane su rute u obliku `suspend` funkcija i unutar njih odgovarajući REST tipovi zahtjeva na mrežu i dohvaćanje njihovih rezultata. Za lakše rukovanje HTTP zahtjevima implementirana je posebna klasa zvana `Response` na koju se mapiraju svi rezultati zahtjeva na mrežu i koja može biti tipa `Success` i `Error` ovisno o tome je li zahtjev uspješno izvršen ili ne.

Kod 4.15. Primjer rute u repozitoriju i klasa Response

```
class NetworkRepository(private val httpClient: HttpClient) {
    suspend fun getSchools(): Response<List<School>> = safeResponse {
        httpClient
            .get("/api/schools")
            .body()
    }
}

sealed interface Response<out T> {
    class Success<T>(val data: T) : Response<T>
    class Error(val e: Exception) : Response<Nothing>
}
```

4.3. Flutter

O ovom poglavlju bit će opisano postavljanje projekta i struktura aplikacije napisane u *Flutteru*. Sama arhitektura neće biti opisana u detalje kao u prethodnom odlomku jer je fokus ovog rada ipak na *Compose Multiplatformu*, ali cilj je dati dojam o općoj strukturi aplikacije i zahtjevnosti implementacije projekta u *Flutteru* u odnosu na projekt u *Compose Multiplatformu*.

Za početak razvoja aplikacije u *Flutteru* potrebno je skinuti *Flutter SDK* (Software Development Kit) koji treba raspakirati u željeni direktorij i dodati ga u globalnu PATH varijablu radne okoline. Za integrirano razvojno okruženje mogu se koristiti mnogi alati, ali najpraktičniji su *Android Studio* i *Visual Studio Code* koji podržavaju *plugin* za podršku razvoja *Flutter* aplikacija. Pokretanjem naredbe `flutter doctor` u terminalu može se provjeriti jesu li uspješno instalirane sve komponente potrebne za razvoj. Ako su sve komponente uspješno instalirane, moguće je kreirati novi projekt u *Flutteru* uz pomoć već navedenog *plugina*.

U ovom dijelu će se ukratko opisati struktura kreiranog projekta u *Flutteru*. Ovisno o odabranim platformama pri kreiranju projekta, moguće je vidjeti sljedeće direktorije i datoteke na vrhu strukture projekta:

- **lib**: Jezgra *Flutter* aplikacije koja sadrži sav zajednički kod, poslovnu logiku i ulaznu točku aplikacije `main.dart`. Sadržaj ovog direktorija dijeli se na modele, ekrane, widgete, servise i druge mape koje se po potrebi dodaju i organiziraju.

- **assets**: Direktorij koji sadrži resurse koji mogu biti slike, fontovi i druge medijske datoteke.

- **platforme**: Ovisno o odabranim platformama, mogu se vidjeti sljedeći direktoriji: `android`, `ios`, `web` i desktop direktoriji `windows`, `linux` i `macos`. Svaka platforma sadrži specifičan kod potreban za kreiranje aplikacije na toj platformi i kod koji poziva zajedničku aplikaciju lociranu u **lib** direktoriju.

Ulazna točka aplikacije je funkcija `main()` kojoj je glavna uloga pokretanje druge funkcije zvane `runApp()` koja kao argument prima implementaciju aplikacije u obliku *Widgeta*. Widgeti su ekvivalent *Composable* funkcijama i njihova je uloga prikaz korisničkog sučelja i praćenje korisnikove interakcije. Dijele se na *Stateless* (bez stanja) i *Stateful* (sa stanjem) widgete, a razlika između tih tipova je da prvi ne prati stanje aplikacije i nema sposobnost praćenja korisnikove interakcije, dok drugi ima.

Kod 4.16. Main funkcija u *Flutteru*

```
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Diplomski Flutter',
      theme: ThemeData(
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple),
        useMaterial3: true,
      ),
      home: const LoginScreen(),
    );
  }
}
```

Navigacija u *Flutteru* slijedi slične pragmatične principe kao i navigacija preko vanjske biblioteke *Voyager* te je, poput navigacije u *Voyageru*, jednostavnija i intuitivnija za korištenje od nativne *Compose* navigacije (koja u trenutku pisanja nije implementirana za *Compose Multiplatform*). `Navigator` je widget koji nudi metode za dodavanje i brisanje ekrana sa stoga navigacije koji pamti stanje prethodno postavljenih ekrana. Ko-

rištenjem funkcija `push()` i `pop()` mogu se dodavati i micati ekrani sa stoga. Jedina razlika od *Voyagerove* navigacije je da ne postoji funkcija za micanje svih ekrana sa stoga i dodavanje novog, ali se taj slučaj može pokriti izradom funkcije ekstenzije nad stanjem navigatora kako je prikazano u sljedećem isječku koda.

Kod 4.17. Primjer navigacije u *Flutteru*

```
extension NavigationExtensions on NavigatorState {
  void pushAndRemoveAll(MaterialPageRoute route) {
    pushAndRemoveUntil(
      route,
      (route) => false,
    );
  }
}

class _HomeScreenState extends State<HomeScreen> {
  void logoutFunction() {
    final MyAppState appState = Provider.of<MyAppState>(context, listen: false);
    appState.logoutUser();
    Navigator.of(context).pushAndRemoveAll(
        MaterialPageRoute(builder: (context) => const LoginScreen())
    );
  }
}
```

Uz to, zanimljivo je napomenuti da je u *Flutteru* implementirana nativna navigacija unazad za iOS uređaje koja bi se u *Compose Multiplatformu* morala nativno implementirati u *Swift* datotekama koje pokreću zajedničku aplikaciju. Takve stvari nisu velike, ali mogu značajno utjecati na korisničko iskustvo.

Zadnja usporedba *Compose Multiplatforma* i *Fluttera* koja će se opisati je mrežna komunikacija. U ovom pogledu *Compose Multiplatform*, tj. alat *Ktor* nadmašuje *Flutter* po pitanju jednostavnosti i količini šablonskog koda. Za razliku od implementacije serijalizacije u *Compose Multiplatformu* gdje je dovoljno staviti anotaciju `@Serializable` na željenu klasu modela, kako bi omogućili slanje i primanje definiranih modela s poslužitelja u *Flutteru* potrebno je raditi vlastite implementacije serijalizacije u JSON. Alternativa je da se koriste vanjske biblioteke, ali ni to nije idealan slučaj jer većina biblioteka također zahtjeva neku vrstu šablonskog koda za implementaciju. Također, biblioteka u *Flutteru* za mrežne pozive `http` je dosta štura i prikladna samo za manje projekte, dok je za kompleksnije slučajeve bolje koristiti neki drugi alat, poput `Dio` biblioteke.

Kod 4.18. Vlastita implementacija serijalizacije modela

```
class School {
  ...
  factory School.fromJson(Map<String, dynamic> json) {
    var clasroomsList = json["classrooms"] as List;
    List<Classroom> classrooms = clasroomsList.map((clasroomJson) =>
        Classroom.fromJson(clasroomJson)).toList();

    return School(
      id: json["id"] as int,
      name: json["name"] as String,
      classrooms: classrooms
    );
  ...
}
```

Kod 4.19. Mrežni pozivi unutar repozitorija

```
const _scheme = "http";
const _host = "diplomski-api.com";
const _port = 3000;

extension on Uri {
  Uri fromPath(String path) {
    return Uri(scheme: _scheme, host: _host, port: _port, path: path);
  }
}

Future<List<School>> getSchools() async {
  var uri = Uri().fromPath("/api/schools");
  final response = await http.get(uri);

  if (response.statusCode == 200) {
    Iterable iterable = jsonDecode(response.body);
    return List<School>.from(iterable.map((model) => School.fromJson(model)));
  } else {
    throw Exception('Failed to load schools');
  }
}
```

5. Korisničko sučelje

U ovom poglavlju bit će detaljno opisan razvoj aplikacije odvojen po cjelinama gdje svaka cjelina predstavlja jedan ekran u aplikaciji. Fokus ovog poglavlja je na implementaciji korisničkog sučelja u tehnologijama *Compose Multiplatform* i *Flutter*, razlikama između implementacija i usporedbi prednosti i mana svake tehnologije.

Prije nego što krenemo u pregled implementacije korisničkog sučelja po ekranima, valja objasniti koje su funkcionalnosti implementirane u ovoj pokaznoj aplikaciji i koja je svrha aplikacije. Aplikacija koja je napravljena u sklopu ovog rada omogućuje korisnicima (učenicima) da se prijave u aplikaciju odabirom svoga imena i prezimena i dobiju pregled svih lekcija povijesti koje su njihovi učitelji unijeli u aplikaciju. Svaka lekcija implementirana je u obliku interaktivne vremenske lente na kojoj postoji više događaja koji se mogu otvoriti i proučiti. Ostatak detalja bit će opisan u narednim poglavljima gdje će se opisivati tok aplikacije i implementacija korisničkog sučelja po ekranima.

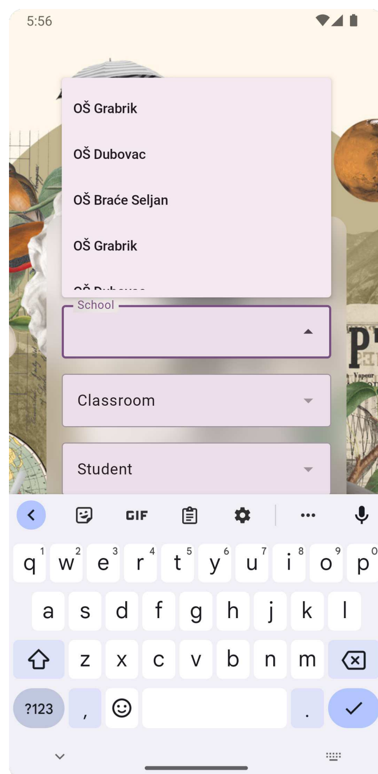
5.1. Ekran za prijavu korisnika

Pri instalaciji aplikacije, prvi ekran koji se učeniku prikazuje je ekran za prijavu. Učenik treba odabrati svoju školu, razred i ime i prezime kako bi se prijavio u aplikaciju. Gumb za prijavu učenika postane omogućen tek kada odabere sva potrebna polja. Za potrebe ovog rada nije bilo potrebno implementirati autentifikaciju korisnika jer se nigdje ne spremaju osjetljivi podaci niti učenik ima mogućnost mijenjanja podataka u bazi kroz sučelje aplikacije.

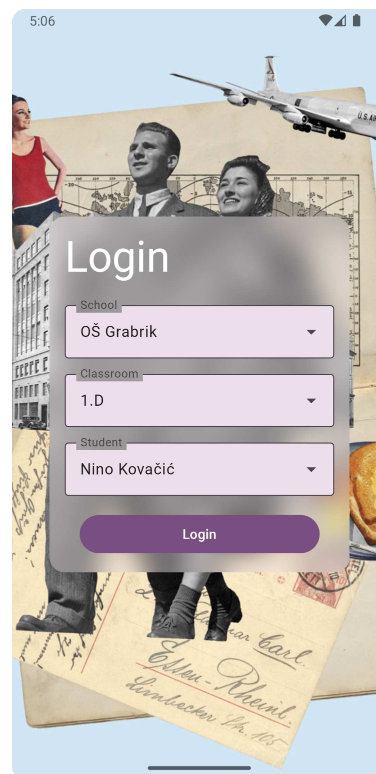
Početni ekran sastoji se od forme za unos podataka i pozadine. Slike na pozadini su spremljene kao URL veze, stoga je potrebno koristiti neku vrstu *Composable* funkcije koja podržava asinkrono učitavanje slika s interneta. Trenutno ne postoji nativna

podrška za učitavanje slika s interneta za *Compose Multiplatform*, stoga se za potrebe ovog projekta koristi *Kamel*. *Kamel*[28] je biblioteka za asinkrono učitavanje medija za *Compose Multiplatform* koja omogućuje, između ostalog, dohvat slika s interneta preko URL-a kojeg proslijedimo funkciji *KamelImage*. *Kamel* prema zadanim postavkama koristi *Ktor* klijent za učitavanje resursa koji već imamo postavljen kao ovisnost u aplikaciji.

Originalna ideja bila je učitati GIF sliku putem alata za asinkrono učitavanje slika s interneta, međutim *Kamel* ne podržava taj tip medija, stoga se u *Compose Multiplatform* aplikaciji svakih 5 sekundi izmjenjuju slike u pozadini. U *Flutteru* nema tih problema jer njegova standardna biblioteka za učitavanje slika već podržava sve formate tako da se može učitati i slika u GIF formatu.



(a) Odabir škole kroz padajući izbornik



(b) Popunjena forma za prijavu

Slika 5.1. Ekran za prijavu korisnika

Forma se sastoji od 3 polja za unos teksta s padajućim izbornikom. U *Composeu* postoji samo eksperimentalna verzija ove komponente koja se zove `ExposedDropdownMenuBox` i u koju je potrebno umetnuti polje za unos zvano `OutlinedTextField` zajedno s padajućim izbornikom `DropdownMenu` koji se pojavljuje kada ima barem jedan ponuđeni

predmet u listi. Lista ponuđenih predmeta filtrira se ovisno o tekstu koji je upisan u polju za unos.

Istu funkcionalnost u *Flutteru* moguće je implementirati korištenjem widgeta `TypeAheadField` koji omata `TextField` i ima mogućnost upravljanja s prikazivanjem padajućeg izbornika i filtriranjem sadržaja ovisno o unesenom tekstu, kao i u prethodno spomenutom *Compose Multiplatform* ekvivalentu.

Još jedna funkcionalnost koja je implementirana u sklopu ovog ekrana je okvir koji omeđuje formu. Okvir je dizajniran da izgleda kao prozirno staklo kroz koje se može vidjeti pozadina, ali zamućena. Željeni efekt se zove efekt mat stakla (eng. *frosted glass effect*). Implementacija tog efekta u *Composeu* se pokazala kao jako veliki izazov jer ovisi o platformski specifičnim načinima manipuliranja grafike, stoga se koristi vanjska biblioteka za postizanje upravo tog efekta. Ta biblioteka zove se *Haze*[29], a korištenje nje svodi se na postavljanje *Modifiera* `haze()` za element koji želimo da se zamuti i `hazeChild()` za elemente koje želimo da budu oni koji će zamutiti pozadinu. Također, moguće je odabrati nekoliko opcija za jačinu zamućivanja pozadine, od *"Ultra Thin"* do *"Ultra Thick"*. Kako bi se uspješno izvršila operacija zamućivanja, potrebno je i pozadini i elementu ispred pozadine predati isti i `HazeState`.

Kod 5.1. Korištenje *Haze* modifikatora

```
val hazeState = remember { HazeState() }
// Pozadina koja ce se zamutiti
KamelImage(
    modifier = Modifier.haze(
        hazeState,
        style = HazeMaterials.ultraThin()
    ),
    resource = asyncPainterResource(data = backgroundImage)
)
// Okvir forme
Column(
    modifier = Modifier
        .align(Alignment.Center)
        .fillMaxWidth(0.8f)
        .hazeChild(
            state = hazeState,
            shape = RoundedCornerShape(16.dp)
        )
)
```

U *Flutteru* se ista funkcionalnost može napraviti ručno, bez korištenja vanjskih biblioteka. Potrebno je koristiti widget `BackdropFilter` koji ima mogućnost zamućivanja pozadine s podesivom jačinom ovisno o predanim parametrima. Unutar njega treba se dodati widget `Container` s dekoracijom `BoxDecoration` bijele boje i željene transparentnosti koja se može postaviti preko funkcije `withOpacity()` koja prima *"alphu"* boje kao parametar. Time se u par jednostavnih koraka implementirala funkcionalnost koja je u *Compose Multiplatformu* vrlo teško izvediva i zahtjeva puno vremena za implementaciju.

Zadnja stvar koju treba pokriti na ovom ekranu je što se događa kada korisnik unese podatke u formu i pritisne gumb za prijavu. Aplikacija treba nekako lokalno pratiti koji je trenutnu prijavljeni korisnik kako bi se pri sljedećim paljenjima aplikacije otvorio početni ekran, a ne ekran za prijavu. Za implementaciju te funkcionalnosti koristi se vanjska biblioteka zvana *Multiplatform Settings*[30] koja omogućuje spremanje parova ključ-vrijednost (eng. *key-value pair*) u zajedničkom kodu. Za potrebe spremanja podataka o trenutno prijavljenom korisniku, napravljen je objekt `UserAccount` u kojem se nalaze podaci o prijavljenom korisniku i metode za prijavu i odjavu korisnika.

Kod 5.2. Praćenje trenutno prijavljenog korisnika

```
private const val LOGGED_OUT_USER_ID = -1
private const val USER_ID = "USER_ID"

object UserAccount {
    private val settings: Settings = Settings()
    var userId = settings.getInt(USER_ID, LOGGED_OUT_USER_ID)
        set(value) {
            field = value
            settings.putInt(USER_ID, value)
        }
    fun login(userId: Int) {
        this.userId = userId
    }
    fun logout() {
        this.userId = LOGGED_OUT_USER_ID
    }
    fun isUserLoggedIn(): Boolean = userId != LOGGED_OUT_USER_ID
}
```

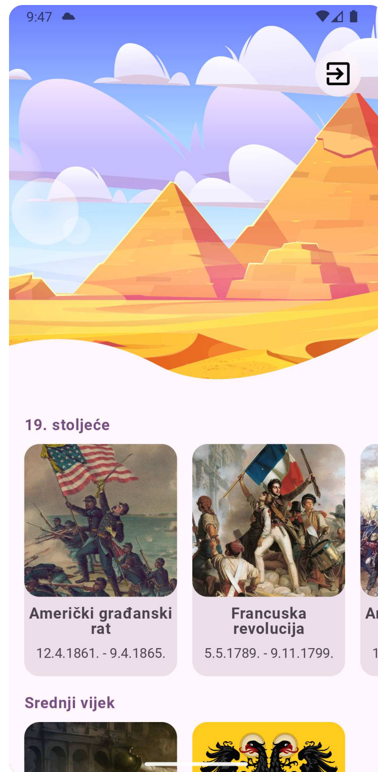
Sada se pri pokretanju aplikacije može provjeriti je li korisnik prijavljen i ovisno o tome pokazati ekran za prijavu ili početni ekran.

Kod 5.3. Postavljanje početkog ekrana

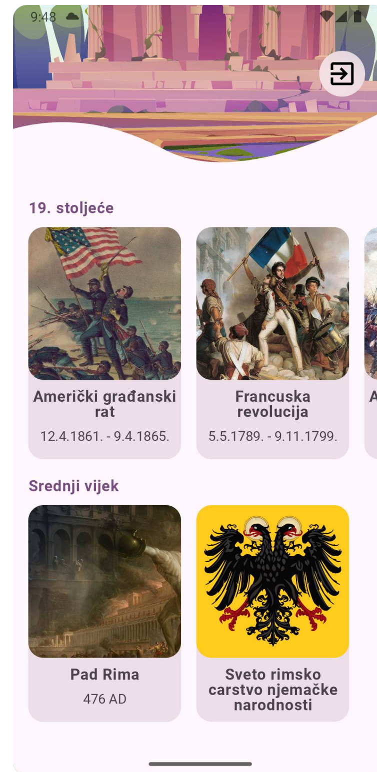
```
@Composable
fun App() {
    KoinApplication(moduleList = { appModule() }) {
        AppTheme {
            val startScreen = if (UserAccount.isUserLoggedIn()) {
                HomeScreen(UserAccount.userId)
            } else {
                LoginScreen()
            }
            Navigator(startScreen) {
                FadeTransition(it)
            }
        }
    }
}
```

5.2. Početni zaslon

Prvi ekran koji učenik vidi nakon uspješne prijave u aplikaciju je ujedno i glavni ekran aplikacije. Na početnom zaslonu prikazan je popis svih lekcija od svih učitelja kod kojih prijavljeni učenik sluša sate. Lekcije su grupirane po temama i prikazane su u horizontalno skrolabilnoj listi. Također, na vrhu ekrana prikazano je zaglavlje (eng. *header*) koje predstavlja neku od povijesnih slika i sadrži ikonicu za odjavu iz aplikacije. Lekcije su predstavljene u obliku kartica sa slikom i imenom lekcije i trajanjem tog povijesnog događaja ako je trajanje navedeno u bazi. Klikom na neku od lekcija otvara se vremenska lenta za tu lekciju, ali više riječi o tome bit će u sljedećem poglavlju.



(a) Početni ekran



(b) Početni ekran pomaknut prema dolje

Slika 5.2. Početni ekran s lekcijama grupiranim po temama

Za postavljanje oblika zaglavlja kakav je prikazan na slikama 5.2.a i 5.2.b potrebno je napraviti vlastitu implementaciju `Shape` sučelja koje se koristi za oblikovanje slike putem modifikatora `clip()`. Implementacija koristi Bezierovu krivulju[31] za oblikovanje željenog valovitog oblika. Implementacija istog oblika u *Flutteru* izgleda vrlo slično jer *Dart* pruža iste metode koje pruža i *Kotlin* za manipulaciju putanje oblika.

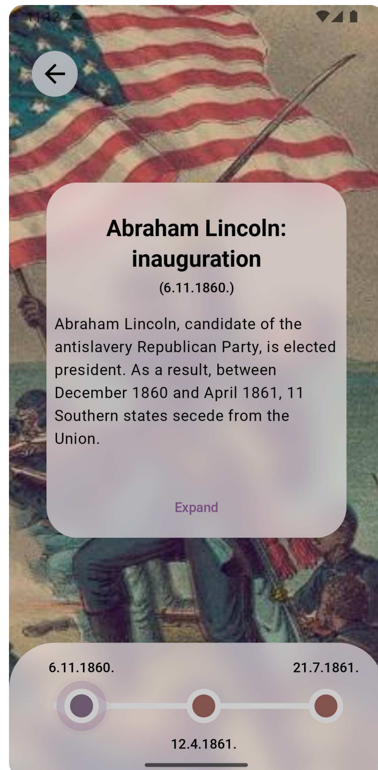
Kod 5.4. Prilagođeni oblik zaglavlja

```
private class HeaderShape : Shape {
    override fun createOutline(
        size: Size,
        layoutDirection: LayoutDirection,
        density: Density
    ): Outline {
        val firstCP = Offset(size.width * 0.2f, size.height * 0.8f)
        val secondCP = Offset(size.width * 0.6f, size.height)
        val middle = Offset(size.width * 0.4f, size.height * 0.9f)
        val path = Path().apply {
            lineTo(0f, size.height * 0.85f)
            quadraticBezierTo(firstCP.x, firstCP.y, middle.x, middle.y)
            quadraticBezierTo(secondCP.x, secondCP.y, size.width, size.height * 0.8f)
            lineTo(size.width, 0f)
        }
        return Outline.Generic(path)
    }
}
```

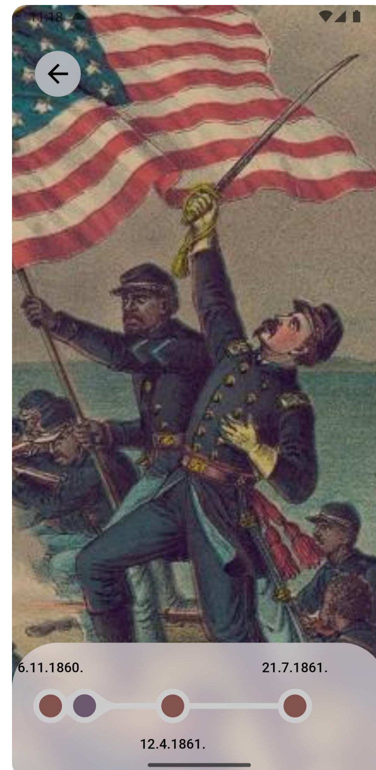
5.3. Vremenska lenta

Kao što je spomenuto u prethodnom poglavlju, odabirom bilo koje od lekcija, učeniku se otvara prikaz vremenske lente za tu lekciju koja se sastoji od niza događaja. Učenik može navigirati po lenti klikom na jedan od "kružića" koji predstavljaju događaj u lenti ili potezanjem prsta, tj. kursora lijevo-desno čime se vremenska lenta pomiče na sljedeći događaj. Ako se navigira po vremenskoj lenti potezanjem lijevo-desno, kartica u sredini se ne prikazuje dok se izvršava potezanje, već se prikaže kada potezanje prestane i kad se odabere neki od događaja. Kada je neki događaj na lenti odabran, prikazat će se pulsirajući efekt nad čvorom tog elementa u lenti.

U pozadini vremenske lente prikazana je slika lekcije koja je ista kao i ona koja je vidljiva na početnom ekranu u kartici za tu lekciju. U sredini ekrana pozicionirana je kartica sa sažetkom odabranog događaja na lenti. U kartici nalaze se naslov događaja, vremensko razdoblje u kojemu se taj događaj odvijao te sažetak događaja. Na dnu kartice nalazi se gumb za proširenje prikaza događaja. Klikom na gumb otvara se detaljan prikaz tog događaja koji će biti opisan u sljedećem poglavlju.



(a) Odabran događaj i pulsirajući efekt



(b) Stanje vremenske lente za vrijeme potezanja ekrana

Slika 5.3. Ekran s vremenskom lentom lekcije

Najveći izazov pri implementaciji ove aplikacije pokazala se upravo vremenska lenta. Ponašanje koje je trebalo dobiti bilo je to da se korisnik može pomicati po lenti jednako na svim platformama. Tu izazov predstavljaju desktop i web platforme kod kojih ne vrijede ista pravila kao kod mobilnih platformi. Komponenta za prikaz liste s velikim broj elemenata u *Composeu* zove se `LazyList` i ima dvije varijante ovisno o orijentaciji: `LazyColumn` za vertikalne liste i `LazyRow` za horizontalne liste.

Na mobilnim platformama se kretanje po listi odvija pritiskom prsta i pomicanjem prsta u smjeru kojem želimo ići, dok se na desktop/web platformama koriste miševi i *touchpadovi*. Za kretanje po horizontalnoj listi kakva je ova vremenska lenta, potrebno je ili s dva prsta kretati se po *touchpadu* ili koristiti miš s pritisnutom *Shift* tipkom. Nijedan od ovih pristupa nije intuitivan korisnicima, pogotovo korisnicima koji nemaju iskustva s korištenjem tehnologije, stoga se morao koristiti drukčiji pristup kretanju po vremenskoj lenti.

Odluka koja je donesena je da će se ujednačiti ponašanje na svim platformama s onim kakvo je na mobilnim platformama. To znači da će kretanje po listi na desktop i web platformama također biti omogućeno povlačenjem sadržaja prstom ili mišem, bez potrebe za dodatnim tipkama ili gestama. Na taj način, korisnici će imati dosljedno iskustvo bez obzira na uređaj koji koriste, što će značajno poboljšati intuitivnost i pristupačnost aplikacije.

Jedina mana ovog pristupa je što zahtjeva prilagođenu implementaciju koja ne iskorištava sve mogućnosti `LazyList` komponente, već manipulira njezino stanje s vanjskim parametrima. Svaka `LazyList` komponenta ima svoj objekt stanja `ListState` koji se može podići van konstruktora komponente kako bi se promatralo i kontroliralo pomicanje liste. Objekt stanja ima razne parametre i metode koje se mogu koristiti za manipulaciju pozicije liste od kojih će se izdvojiti one koje su se koristile u sklopu ovog zadatka:

- `firstVisibleItemIndex`: Varijabla koja označava prvi vidljivi element u listi koji je trenutno iscrtan na ekranu. Ona je ključna za ovu implementaciju jer se na vremenskoj lenti pomicanjem odabire prvi vidljivi element.

- `scrollBy()`: Suspendirajuća funkcija koja pomiče listu za odabrani broj pixela koji se predaje kao argument funkcije.

- `animateScrollToItem()`: Suspendirajuća funkcija koja prima indeks elementa u listi i obavlja animaciju pomicanja do tog elementa.

Koristeći ove metode za manipulaciju stanja liste i još neke druge modifikatore, moguće je postići prethodno opisanu funkcionalnost. Prvo, potrebno je kreirati `LazyRow` i na njega postaviti modifikator `draggable` koji reagira na povlačenje komponente i sadrži metode za praćenje stanja povlačenja, reagiranje na početak i reagiranje na kraj povlačenja. Dok je u tijeku operacija povlačenja, šaljemo količinu obavljenog povlačenja objektu stanja `ListState` koristeći metodu `scrollBy()`. Također, na početku povlačenja sakrijemo karticu u sredini, dok na kraju pozicioniramo listu na trenutno najbliži prvi vidljivi element i ponovno prikažemo karticu. Za kraj, potrebno je onemogućiti normalno ponašanje liste postavljanjem parametra `userScrollEnabled`. Dio implementacije ovog ponašanja demonstriran je u sljedećem isječku koda.

Kod 5.5. Implementacija povlačenja vremenske lente

```
LazyRow(
    modifier = Modifier.draggable(
        orientation = Orientation.Horizontal,
        state = rememberDraggableState { delta ->
            coroutineScope.launch {
                listState.scrollBy(-delta)
            }
        },
        onDragStarted = {
            isCardVisible = false
            isCardExpanded = false
        },
        onDragStopped = {
            listState.animateScrollToItem(listState.firstVisibleItemIndex)
            selectedNodeIndex = listState.firstVisibleItemIndex
            isCardVisible = true
        }
    ),
    userScrollEnabled = false,
    state = listState
)
```

Iscrtavanje pozadinske crte na vremenskoj lenti obavlja se putem modifikatora `drawWithContent` koji omogućuje manipulaciju iscrtavanja same komponente i dodatnog sadržaja oko nje. Svaki čvor na lenti iscrtan je kao zasebna komponenta korištenjem komponente `Canvas`, a trenutno odabrani element još ima i pulsirajući efekt nacrtan oko njega.

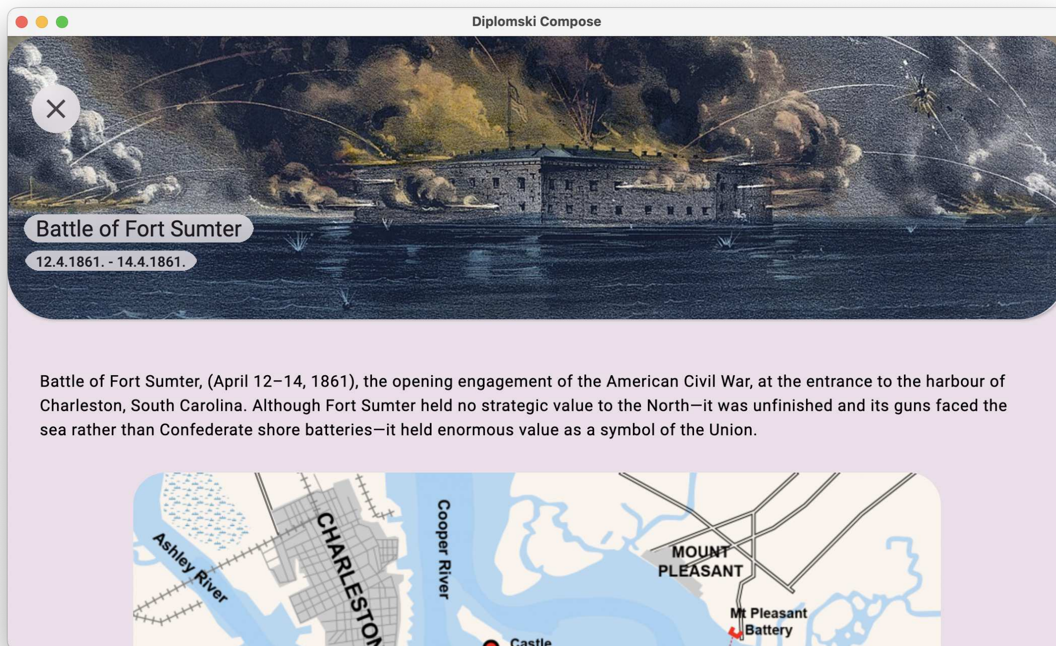
Pulsirajući efekt kreiran je da imitira ponašanje kružnih valova na vodi gdje se obavlja animacija triju koncentričnih krugova. Efekt je kreiran korištenjem beskonačnih tranzicija koje animiraju promjenu veličine i transparentnosti kako se krugovi šire dalje od središta. Korištenjem dvije beskonačne animacije možemo postići željeno ponašanje. Animacije veličine i transparentnosti prikazane su u sljedećem isječku koda.

Kod 5.6. Animacija pulsirajućeg efekta

```
@Composable
private fun pulsarBuilder(
    targetRadius: Float,
    initialRadius: Float,
    delayMillis: Int
): Pair<Float, Float> {
    val infiniteTransition = rememberInfiniteTransition(label = "")
    val radius by infiniteTransition.animateFloat(
        initialValue = initialRadius,
        targetValue = targetRadius,
        animationSpec = InfiniteRepeatableSpec(
            animation = tween(3000),
            initialStartOffset = StartOffset(delayMillis),
            repeatMode = RepeatMode.Restart
        ), label = "radiusAnimation"
    )
    val alpha by infiniteTransition.animateFloat(
        initialValue = 0.6f,
        targetValue = 0f,
        animationSpec = InfiniteRepeatableSpec(
            animation = tween(3000),
            initialStartOffset = StartOffset(delayMillis + 100),
            repeatMode = RepeatMode.Restart
        ), label = "alphaAnimation"
    )
    return radius to alpha
}
```

5.4. Detalji o događaju

Zadnji ekran u aplikaciji je ekran detalja o događaju. Ulazak u taj ekran obavlja se klikom na gumb "Proširi" (eng. "Expand") unutar kartice nekog događaja na vremenskoj lenti kao što je prikazano na slici 5.3.a Nakon klika gumba potrebno je animirati otvaranje detalja o događaju na način da novi ekran "nastane" animacijom proširenja iz kartice i zauzme cijeli mogući prostor. Detalji o nekom događaju sastoje se od kombinacije slika i teksta koji se dohvaćaju s mreže u obliku liste JSON datoteka. JSON datoteka za slikovni sadržaj sastoji se od URL poveznice na resurs na webu.



Slika 5.4. Detalji o događaju na desktop platformi

Animacija prijelaza iz jednog u drugi ekran bi u pravilu trebao biti zadatak navigacije, ali kako za *Compose Multiplatform* ne postoji nativna implementacija navigacije, a biblioteka *Voyager* ne podržava prilagođene animacije od ekrana do ekrana, morala se napraviti vlastita implementacija animacije kako bi se postiglo željeno ponašanje.

Animacija koja je implementirana zapravo nije animacije prijelaza s jednog u drugi ekran, već je samo animacija proširenja i smanjenja veličine kartice događaja. Kartici se predaje modifikator popunjenja postotka širine i visine ekrana, a taj postotak mijenja se ovisno o tome je li kartica u proširenom ili skupljenom stanju.

Kako bi se animirala promjena tih postotaka, koristi se *Composable* funkcija `animateFloatAsState` koja animira promjene između predanih stanja i može joj se predati tip animacije koja se koristi. Kartica zauzima 80% širine i 50% visine dok je skupljena, a cijeli slobodni prostor dok je proširena. Implementacija ove animacije prikazana je u sljedećem isječku koda.

Kod 5.7. Animacija proširenja kartice

```
val width by animateFloatAsState(
    targetValue = if (isCardExpanded) 1f else 0.8f,
    animationSpec = tween(500)
)
val height by animateFloatAsState(
    targetValue = if (isCardExpanded) 1f else 0.5f,
    animationSpec = tween(500)
)
TimelineCard(
    timelineEntry = it,
    modifier = Modifier
        .fillMaxHeight(height)
        .fillMaxWidth(width)
        .hazeChild(
            hazeState,
            shape = RoundedCornerShape(32.dp)
        ),
    isExpanded = isCardExpanded,
    onExpand = { isCardExpanded = true },
    onCollapse = { isCardExpanded = false }
)
```

Za razliku od implementacije u *Compose Multiplatformu* gdje se moralo raditi prilagođeno rješenje, *Flutter* već ima podršku za ovakve situacije. `Hero` [32] je specijalni widget koji omogućava kreiranje animacija prijelaza između ekrana. Kada korisnik navigira s jednog ekrana na drugi, `Hero` widget omogućava animaciju prijelaza određenog elementa korisničkog sučelja, čime se stvara efekt prijelaza gdje element "leti" s jednog ekrana na drugi.

Za implementaciju animacije, potrebno je oba ekrana omotati s `Hero` widgetom. Ključni dio implementacije je postavljanje jedinstvene oznake (eng. *tag*) na oba ekrana kako bi widget znao prepoznati koji su ekrani za koje treba napraviti prijelaz. Animacija se pokreće kada korisnik dodirne widget na prvom ekranu i navigira na drugi ekran. U našem primjeru to se dogodi na promjenu ekrana putem `Navigator.push()` metode. Također, moguće je promijeniti tip animacije, ali u ovom slučaju je pretpostavljena animacija upravo ono što nam treba.

Za kraj ovog poglavlja o detaljima događaja opisat će se funkcionalnost zatvaranja ekrana. Ekran se može standardno zatvoriti klikom na ikonicu za zatvaranje u gornjem

lijevom kutu ekrana, međutim zbog praktičnosti dodana je još jedna funkcionalnost koja olakšava navigaciju kroz aplikaciju. Desktop korisnicima je intuitivno koristiti *Escape* gumb za povratak na prethodni ekran dok je Android korisnicima prirodno koristiti *Back* gumb ili gestu za povratak unazuad, međutim to u *Compose Multiplatformu* nije omogućeno prema zadanim postavkama, stoga je bilo potrebno implementirati vlastito rješenje.

Prije nego što se objasni sama implementacija ove funkcionalnosti, treba pojasniti kako se implementiraju funkcionalnosti koje su specifične za svaku platformu. Takav se slučaj *Kotlin Multiplatformu* (a samim time i u *Compose Multiplatformu*) rješava korištenjem očekivanih i stvarnih vrijednosti. U zajedničkom kodu se definira `expect` (očekivana) vrijednost koja definira zajedničko sučelje i koja se implementira na svakoj platformi u obliku `actual` (stvarne) vrijednosti.

Tako ćemo za implementaciju ove funkcionalnosti kreirati *Composable* funkciju `BackHandler` kojoj je zadatak pratiti kada je korisnik napravio akciju vraćanja na prethodni ekran. Implementacija za Android platformu već postoji, tako da će implementacija za tu platformu samo pozivati nativnu komponentu. Za implementaciju na desktop platformi potrebno je osluškivati kada korisnik pritisne *Escape* tipku na tipkovnici. Za tu svrhu kreirano je sučelje `ActionStore` koje se ponaša kao oblikovni obrazac "Promatrač" (eng. *Observer*) i kojemu se javlja da se neka akcija dogodila kako bi on mogao javiti pretplatnicima da obave prikladnu reakciju na tu akciju. Za potrebe funkcionalnosti vraćanja na prethodni ekran dovoljno je kreirati samo jedan tip akcije koja se zove `OnBackPressed`. Sada stvarna vrijednost komponente `BackHandler` za desktop platformu može osluškivati dolazeće akcije i, ako je akcija `OnBackPressed`, prijeći na prethodni ekran.

Kod 5.8. BackHandler komponenta za desktop platformu

```
1 @Composable
2 expect fun BackHandler(isEnabled: Boolean = true, onBack: () -> Unit)
3
4 @Composable
5 actual fun BackHandler(isEnabled: Boolean, onBack: () -> Unit) {
6     LaunchedEffect(isEnabled) {
7         actionStore.events.collect { if (isEnabled) onBack() }
8     }
9 }
```

Kod 5.9. Implementacija promatrača akcija

```
// Promatrac se definira na razini aplikacije
val actionStore = CoroutineScope(SupervisorJob()).createActionStore()

sealed interface Action {
    data object OnBackPressed: Action
}

interface ActionStore {
    fun send(action: Action)
    val events: SharedFlow<Action>
}

fun CoroutineScope.createActionStore(): ActionStore {
    val events = MutableSharedFlow<Action>()

    return object : ActionStore {
        override fun send(action: Action) {
            launch {
                events.emit(action)
            }
        }
        override val events: SharedFlow<Action> = events.asSharedFlow()
    }
}
```

5.5. Stanje bez internetske veze

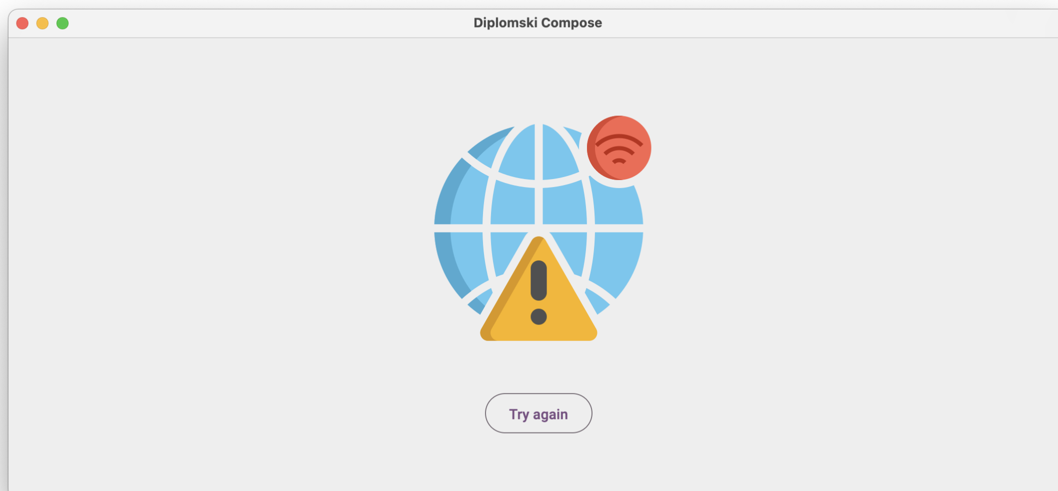
Za kraj će se opisati rukovanje slučajem kada zbog bilo kojeg razloga veza s mrežom nije dostupna. Taj se slučaj može dogoditi zbog nedostupnosti mreže, uključivanja zrakoplovnog načina rada na mobitelu ili drugih razloga, ali važno je da korisnik ima informaciju da veza s mrežom nije uspjela i da mu se ponudi mogućnost da ponovi zahtjev na mrežu. U sklopu toga potrebno je napraviti poseban ekran za slučaj kada je mreža nedostupna i refaktorirati postojuću strukturu ekrana da svaki ekran može podržati provjeru dostupnosti interneta.

Prvo je potrebno na svaki poziv na mrežu dodati provjeru je li poziv uspio ili ne. U svrhu toga stvara se poseban objekt stanja `NetworkState` koji je vidljiv u cijeloj aplikaciji i koji sadrži *Flow* koji pamti je li mreža dostupna ili nedostupna. Uz to je dodan i poseban tip mrežnog poziva koji se poziva svakih **n** sekundi iz trenutno prikazanog ekrana, a služi samo za provjeru je li poslužitelj dostupan.

Kod 5.10. Provjera dostupnosti mreže pri svakom pozivu

```
object NetworkState {
    val connectionStateFlow = MutableStateFlow(ConnectionState.Available)
}
// Ekstenzija za slanje stanja mreze nakon svakog poziva
suspend inline fun <T> safeResConn(crossinline block: suspend () -> T): Response<T> {
    return safeResponse(block).also {
        it.onSuccess { connectionStateFlow.emit(ConnectionState.Available) }
        it.onError { connectionStateFlow.emit(ConnectionState.Unavailable) }
    }
}
```

Nakon implementacije pozadinskog koda potrebnog za prikaz stanja bez interneta, potrebno je refaktorirati postojeće ekrane kako bi mogli podržati ovu funkcionalnost.



Slika 5.5. Stanje bez interneta na desktopu

Prvo treba napraviti baznu ekran koji će naslijeđivati *Voyagerovu* `Screen` klasu. `BaseScreen` je apstraktna klasa koja definira metode koje svi ostali ekrani moraju implementirati i razdvaja ekrane na slučajeve kada postoji i ne postoji veza s mrežom.

Kod 5.11. Razdvajanje ekrana po tipu u baznom ekranu

```
abstract class BaseScreen : Screen {
    @Composable
    abstract fun HasInternetContent()
    @Composable
    private fun NoInternetContent() {
        ...
    }
    @Composable
    override fun Content() {
        networkUtil = getKoin().get<NetworkUtil>()
        networkUtil.periodicallyCheckConnection()
        val connectionState by connectionStateFlow.collectAsState()
        if (connectionState == ConnectionState.Available) {
            HasInternetContent()
        } else {
            NoInternetContent()
        }
    }
    abstract fun retryNetworkCall()
}
```

Kao što se vidi iz prikazanog koda, postoje dva tipa ekrana ovisno o stanju mreže, a svaki ekran mora implementirati ekran kada je aplikacija povezana na mrežu. Svaki ekran mora implementirati i funkciju koja se poziva na ručni pokušaj ponovnog uspostavljanja veze. Također, svakih 10 sekundi se provjerava stanje mreže i ovisno o tome postavlja prikladan ekran. Pomoćna klasa `NetworkUtil` koja obavlja akciju provjere umeće se u bazni ekran korištenjem alata za umetanje ovisnosti *Koin*.

6. Zaključak

U sklopu ovog rada demonstriran je cjelokupni razvoj projekta u *Compose Multiplatformu* zajedno s funkcionalnostima koje ovaj višeplatformski radni okvir pruža. Opisani su svi koraci postavljanja projekta u *Compose Multiplatformu*, a na razvijenoj demonstracijskoj aplikaciji prikazan je veliki broj funkcionalnosti koje se mogu implementirati u ovom radnom okviru. Također, ovaj novi višeplatformski radni okvir uspoređen je sa široko prihvaćenim radnim okvirom *Flutter* koji se godinama koristi za veliki broj projekata i koji ima dugu potporu od *Googlea* i programerske zajednice s ciljem da se naglase prednosti i mane svakog okvira.

Compose Multiplatform još je u ranoj fazi razvoja, ali ima priliku postati jedan od najpopularnijih višeplatformskih radnih okvira zbog svoje svestranosti i praktičnosti pri razvoju. Kako JetBrains nastavlja ulagati u svoj razvoj, možemo očekivati povećanu stabilnost, poboljšanja performansi i bogatiji skup značajki. Integracija okvira s Kotlin Multiplatform vjerojatno će se proširiti, pružajući još bolju podršku za dijeljenje koda i ponovnu upotrebu na različitim platformama.

Na uspjeh bilo kojeg razvojnog okvira uvelike utječu njegova zajednica i ekosustav. Kako sve više programera krene usvajati *Compose Multiplatform*, možemo očekivati da ćemo vidjeti sve veći broj biblioteka, dodataka i alata koji nadograđuju njegove funkcionalnosti. Osim toga, doprinosi zajednice, vodiči i radovi poput ovog pomoći će novim razvojnim programerima da izgrade kvalitetne i robusne aplikacije. Da bi *Compose Multiplatform* dobio široku primjenu, mora dokazati svoju pouzdanost i performanse u velikim projektima. Kako sve više tvrtki bude eksperimentiralo s ovim radnim okvirom i dijelilo svoja iskustva, stjecat ćemo uvid u njegove snage i slabosti. Uspješne studije slučaja i pozitivne povratne informacije od prvih korisnika bit će presudni u uvjeravanju više poduzeća da razmotre *Compose Multiplatform* za svoje projekte.

Neki od nedostataka *Compose Multiplatforma* u trenutnom stadiju razvoja su:

- Rana faza razvoja: Budući da je relativno nova tehnologija, *Compose Multiplatform* može naići na probleme sa stabilnošću i performansama. Razvojni programeri mogli bi se suočiti s strmijom krivuljom učenja zbog ograničenih resursa i podrške zajednice u usporedbi s uspostavljenijim okvirima.

- Nedostatak nativnih komponenti: Ova se točka nadovezuje na prethodnu u smislu da je sam radni okvir još u ranoj fazi razvoja te se još nisu sve komponente, koje su dostupne u nativnim Android projektima, prepisale na višeplatformski način rada. Stoga se u trenutnoj ranoj fazi razvoja može pretpostaviti veća ovisnost o vanjskim bibliotekama i duže vrijeme razvoja zbog pisanja vlastitih komponenti.

- Prilagodbe specifične za platformu: Osigurati da aplikacije izgledaju i osjećaju se izvorno na svakoj platformi može biti izazovno. Programeri moraju uložiti vrijeme u prilagodbe specifične za platformu i testiranje kako bi postigli željeno korisničko iskustvo. Takva situacija nije idealna za višeplatformski razvoj u kojemu je generalno cilj smanjiti količinu platformski ovisnog koda.

Za programere i druge dionike koji razmatraju korištenje *Compose Multiplatforma* za svoje projekte, važno je obratiti pažnju na niz faktora pri odabiru ovog radnog okvira od kojih su najvažniji zahtjevi projekta, stručnost razvojnog tima i dugoročna održivost. Kako je *Compose Multiplatform* još u ranoj fazi razvoja, vjerojatno nije prikladan odabir za velike i važne projekta, već za manje projekte u kojima si timovi mogu dozvoliti mala kašnjenja u razvoju i nastanak potencijalnih grešaka koje se mogu dogoditi pri učestaloj nadogradnji tehnologije. Za razvojne timove koji su već upoznati s razvojem Android aplikacija, ovaj radni okvir je odličan odabir jer je prelazak s nativnog Android razvoja na *Compose Multiplatform* vrlo jednostavan i intuitivan.

Pažljivim razmatranjem svih navedenih čimbenika, razvojni programeri i ostali dionici mogu donijeti informirane odluke koje najbolje odgovaraju njihovim specifičnim potrebama i ciljevima.

Literatura

[1] Robert Sheldon, “What is a codebase (code base)?” https://techtarget.com/whatis/definition/codebase-code-base, [Pristupljeno 30. svibnja 2024.].

[2] JetBrains, “Compose Multiplatform”, https://www.jetbrains.com/lp/compose-multiplatform/, [Pristupljeno 31. svibnja 2024.].

[3] Google, “Jetpack Compose”, https://developer.android.com/jetpack/compose, [Pristupljeno 31. svibnja 2024.].

[4] ——, “Flutter”, https://flutter.dev, [Pristupljeno 31. svibnja 2024.].

[5] Facebook, “React Native”, https://reactnative.dev/, [Pristupljeno 31. svibnja 2024.].

[6] Drifty, “Ionic”, https://ionicframework.com/, [Pristupljeno 31. svibnja 2024.].

[7] Microsoft, “.NET Multi-platform App UI”, https://dotnet.microsoft.com/en-us/apps/maui, [Pristupljeno 31. svibnja 2024.].

[8] ——, “Xamarin”, https://dotnet.microsoft.com/en-us/apps/xamarin, [Pristupljeno 31. svibnja 2024.].

[9] JetBrains, “Kotlin Multiplatform”, https://www.jetbrains.com/kotlin-multiplatform/, [Pristupljeno 31. svibnja 2024.].

[10] Google, “Why adopt compose?” https://developer.android.com/develop/ui/compose/why-adopt, [Pristupljeno 31. svibnja 2024.].

[11] JetBrains, “Kotlin”, https://kotlinlang.org/, [Pristupljeno 9. lipnja 2024.].

[12] Baeldung, “Baeldung: Kotlin”, https://www.baeldung.com/kotlin/, [Pristupljeno 9. lipnja 2024.].

[13] Gradle Inc., “Gradle”, https://gradle.org/, [Pristupljeno 9. lipnja 2024.].

[14] Apache, “Groovy”, https://www.groovy-lang.org/, [Pristupljeno 9. lipnja 2024.].

[15] Rails, “Rails”, https://rubyonrails.org/, [Pristupljeno 9. lipnja 2024.].

[16] Ruby, “Ruby”, https://www.ruby-lang.org/en/, [Pristupljeno 9. lipnja 2024.].

[17] Google, “Dart”, https://dart.dev/, [Pristupljeno 9. lipnja 2024.].

[18] Google, JetBrains, “Android Studio”, https://developer.android.com/studio, [Pristupljeno 11. lipnja 2024.].

[19] JetBrains, “Kotlin Multiplatform Plugin”, https://kotlinlang.org/docs/multiplatform-plugin-releases.html, [Pristupljeno 11. lipnja 2024.].

[20] Web Assembly, “Web Assembly Garbage Collection”, https://github.com/WebAssembly/gc, [Pristupljeno 11. lipnja 2024.].

[21] JetBrains, “Kotlin Multiplatform Wizard”, https://kmp.jetbrains.com/?_gl=1*1xayb7e*_gcl_au*MTUzNTAwNjA2Ny4xNzE4MTM3MzMw*_ga*ODI4NjQ1NDAyLjE3MTgxMzczMjE.*_ga_9J976DJZ68*MTcxODEzNzMyMS4xLjEuMTcxOD&_ga=2.92384861.1521443548.1718137330-828645402.1718137321, [Pristupljeno 11. lipnja 2024.].

[22] ——, “kotlinx-io”, https://github.com/Kotlin/kotlinx-io, [Pristupljeno 15. lipnja 2024.].

[23] Google, “Material Design”, https://m3.material.io/, [Pristupljeno 15. lipnja 2024.].

[24] ——, “Material Theme Builder”, https://material-foundation.github.io/material-theme-builder/, [Pristupljeno 15. lipnja 2024.].

[25] Arnaud Giuliani, “Koin”, https://insert-koin.io/, [Pristupljeno 15. lipnja 2024.].

[26] Adriel Cafe, “Voyager”, https://voyager.adriel.cafe/, [Pristupljeno 15. lipnja 2024.].

[27] JetBrains, “Ktor”, https://ktor.io/, [Pristupljeno 15. lipnja 2024.].

[28] Kamel Media, “Kamel Image Loader”, https://github.com/Kamel-Media/Kamel, [Pristupljeno 16. lipnja 2024.].

[29] Chris Banes, “Haze”, https://chrisbanes.github.io/haze/, [Pristupljeno 16. lipnja 2024.].

[30] Russell Wolf, “Multiplatform Settings”, https://github.com/russhwolf/multiplatform-settings, [Pristupljeno 16. lipnja 2024.].

[31] Wikipedia, “Bezier Curve”, https://en.wikipedia.org/wiki/B%C3%A9zier_curve, [Pristupljeno 16. lipnja 2024.].

[32] Flutter Docs, “Hero: Animations”, https://docs.flutter.dev/ui/animations/hero-animations, [Pristupljeno 16. lipnja 2024.].

Sažetak

Usporedba radnih okvira za razvoj višeplatformskih korisničkih sučelja

Nino Kovačić

U ovom radu istražuju se i uspoređuju dvije tehnologije za razvoj višeplatformskih aplikacija: *Compose Multiplatform* i *Flutter*. *Compose Multiplatform* novi je višeplatformski radni okvir koji proširuje mogućnosti *Jetpack Composea*, deklarativnog alata za razvoj korisničkih sučelja za Android aplikacije. Fokus je stavljen na *Compose Multiplatform* zbog njegove perspektive i trenutnog stanja razvoja koje još nije spremno za velike projekte, ali pokazuje veliki potencijal. U radu je također provedena analiza kako obje platforme rade "iza kulisa", usporedba arhitektonskih obrazaca te opis procesa izrade aplikacija. Cilj ovog rada je pružiti jasne smjernice i uvid u proces razvoja aplikacija koristeći *Compose Multiplatform*, omogućujući čitateljima da sami procijene je li ova tehnologija prikladna za njihove projekte.

Ključne riječi: višeplatformski, multiplatformski, radni okvir, korisničko sučelje, Compose Multiplatform, Kotlin, Android, Flutter

Abstract

Comparison of Frameworks for Developing Cross-Platform User Interfaces

Nino Kovačić

This thesis explores and compares two technologies for multiplatform application development: *Compose Multiplatform* and *Flutter*. *Compose Multiplatform* is a new multiplatform framework that extends the capabilities of Jetpack Compose, a declarative tool for developing user interfaces for Android applications. Emphasis is placed on *Compose Multiplatform* due to its perspective and current development state, which is not yet ready for large projects, but shows great potential. The study also delves into the behind-the-scenes workings of both platforms, comparing architectural patterns and the application development process. The goal of this thesis is to offer clear guidance and insights into the application development process using *Compose Multiplatform*, enabling readers to determine if this technology is suitable for their projects.

Keywords: multiplatform, framework, user interface, Compose Multiplatform, Kotlin, Android, Flutter