BITCOIN NOVČANIK ZA ANDROID

Skako, Josip

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Specijalistički diplomski stručni studij Informacijske tehnologije

JOSIP SKAKO

Z A V R Š N I R A D

BITCOIN NOVČANIK ZA ANDROID

Split, rujan 2021.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Specijalistički diplomski stručni studij Informacijske tehnologije

Predmet: Kriptovalute

Z A V R Š N I R A D

Kandidat: Josip Skako

Naslov rada: Bitcoin novčanik za Android

Mentor: Nikola Grgić, viši predavač

Split, rujan 2021.

Sadržaj

Sažetak

U ovom završnom radu opisan je postupak izrade funkcionalnog Bitcoin novčanika. Novčanik je kreiran u razvojnom okruženju Android Studio, programskome jeziku Kotlin, izrađen je za sve uređaje koji imaju operativni sustav Android 10 (API 29) ili više, pisan je u MVVM arhitekturi (engl. *Model–view–viewmodel*), a za verzioniranje je korišten sustav GitHub. Svrha ovog novčanika je primanje i slanje bitcoina na glavnoj i testnoj mreži te pregled povijesti transakcija. Za pohranjivanje privatnih i javnih ključeva i ostalih osjetljivih podataka se koristi *android keystore* i *shared preferences*. Podatci koji se dohvaćaju s lanca blokova (engl. *Blockchain)* se pohranjuju u SQLite bazu podataka na uređaju. Vizualno sučelje aplikacije je izrađeno u XML-u kroz razvojno okruženje Android Studio.

Ključne riječi: Android studio, Kotlin, Bitcoin novčanik, SPV

Summary

Bitcoin wallet for Android

This final paper describes the process of making a functional Bitcoin wallet. The wallet is created in the Android studio, Kotlin programming language, build for all devices running Android 10 (API 29) or higher, written in the MVVM architecture (Model - view - viewmodel), and the GitHub system is used for versioning. The purpose of this wallet is to receive and send bitcoin on the main and test network and to overview the history of transactions. Android keystore and shared preferences are used to store private and public keys and other sensitive data. Data retrieved from the blockchain is stored in the SQLite database on the device. The visual interface of the application is written in XML through the Android studio.

Keywords: Android studio, Kotlin, Bitcoin wallet, SPV

1. Uvod

Bitcoin je jedna od najbrže rastućih mreža i kriptovaluta koja doseže oko 200,000 transakcija dnevno, a sam pojam bitcoin se može odnositi i na digitalni novac. Konsenzus o tome tko posjeduje koji novčić postiže se kriptografski preko javnih čvorova umjesto da se oslanja na treću stranku poput banke. Novčići se čuvaju u digitalnom novčaniku s kojim se vrši slanje i primanje. Umjesto fizičke valute, digitalni novčanik će pohranjivati kriptografske informacije koje se koriste za pristup bitcoin adresama i slanje transakcije. Kriptografske informacije su generirane iz ključne rečenice (engl. *Seed phrase*) koju generira novčanik ili je upisuje korisnik. Ključna rečenica predstavlja niz od 12 do 24 nasumično generirane riječi koja je postala standard za deterministički novčanik. Deterministički novčanik je sustav generiranja ključeva iz jedne početne točke koja se naziva *seed*. Novčanici na mobilnim uređajima najčešće se implementiraju s pojednostavljenom provjerom plaćanja poznatom kao SPV (engl. *Simplified payment verification*).

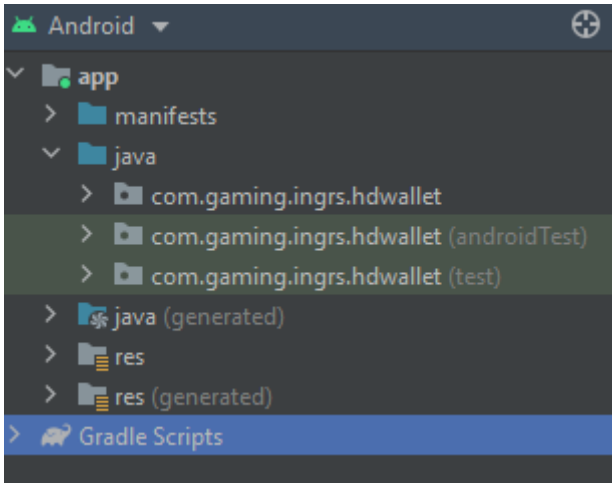
Rad je podijeljen na pet cjelina. U uvodu su kratko opisani motivacija pisanja rada, osnovni podatci o aplikaciji, krajnji cilj, i opis ostalih poglavlja. U drugoj cjelini su opisane korištene tehnologije vezane uz rad. Unutar trećeg poglavlja je detaljno izrađena inicijalizacija novčanika, od zaslona za učitavanje do odabira novčanika i postavljanja šifre. U četvrtom poglavlju je razrađeno plaćanje, primanje, povijest transakcija, oporavak novčanika, generiranje adresa, navigacija, gadgeti, i postavke. U petoj, ujedno i posljednjoj cjelini, donosi se zaključak teme.

2. Korištene tehnologije

Novčanik je izrađen u razvojnom okruženju Android Studio koristeći Kotlin programski jezik i biblioteke za rad s Bitcoinom, u MVVM arhitekturi. Android Studio je službeno integrirano razvojno okruženje za razvoj Android aplikacija temeljeno na IntelliJ IDEA. MVVM je uzorak koji razdvaja aplikaciju na tri dijela: *View*, *Model*, i *ViewModel*. Ova arhitektura je preporučena od *Googlea* kao jedna od najboljih za Android aplikacije. S ovom arhitekturom komponente korisničkog sučelja se drže odvojeno od poslovne logike i komponente su svjesne životnoga ciklusa [1]. Koristi se SQLite baza podataka u kojoj se pohranjuju kriptirani detalji dobiveni s lanca blokova i koje generira novčanik.

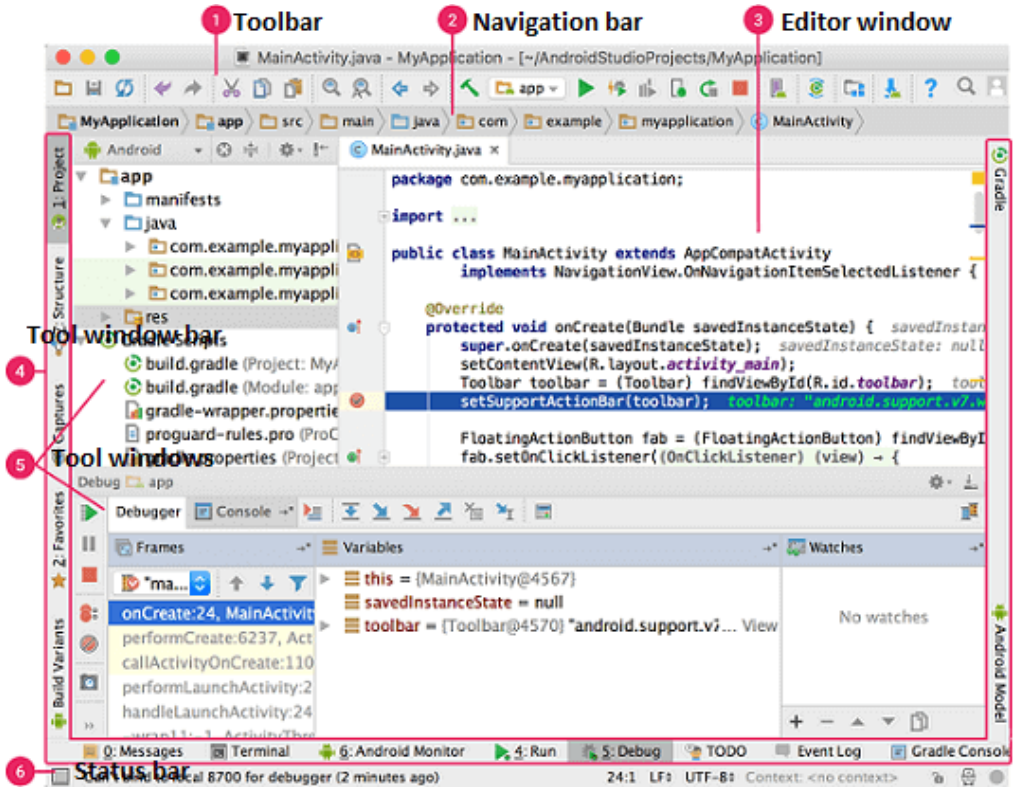
2.1. Android Studio

Razvojno okruženje Android Studio nudi više značajki koje povećavaju produktivnost pri izradi Android aplikacija. Najavljen je 16. svibnja 2013. na *Google* konferenciji i započeo je s verzijom 0.1. Prva stabilna verzija je objavljena u 12. mjesecu 2014. Neke od značajki su da ima fleksibilan sustav gradnje temeljen na *Gradleu*, ima brz emulator bogat značajkama za testiranje aplikacija, mogućnost razvijanja za sve android uređaje, ažuriranje koda na uređaju bez potrebe za resetiranjem aplikacije, napredni alat za testiranje, ima podršku za C++ i NDK, jednostavno je integrirati *Googleove* servise,…



Slika 1: Struktura unutar razvojnog okruženja Android Studio

Slika 1 prikazuje strukturu mapa unutar razvojnog okruženja Android Studio koje su podijeljene po modulima. Pomoću njih pristupamo ključnim datotekama projekta. Unutar `manifest` mape se nalazi `AndroidManifest.xml` koji sadrži ključne informacije o aplikaciji, `java` mapa sadrži izvorni kôd aplikacije uključujući i kôd za testiranje, i na kraju `res` mapa sadrži sve resurse koji nisu kodirani te *xml* datoteke korisničkog sučelja.



Slika 2: Sučelje Android studija [1]

Slika 2 prikazuje sučelje razvojnog okruženja Android Studio. Alatna traka (engl. *Toolbar*) nudi širok raspon radnji, uključujući pokretanje aplikacije i Android alata. Navigacijska traka (engl. *Navigation bar*) omogućuje kompaktan prikaz strukture. Prozor za uređivanje (engl. *Editor window*) je mjesto gdje se kreira i mijenja kôd. Traka prozora (engl. *Tool window bar*) sadrži gumbe koji omogućuju proširenje i sužavanje pojedinih prozora alata. Alatna traka (engl. *Tool windows*) omogućuje pristup određenim zadacima poput pretraživanja, upravljanja projektima, kontroli verzija i drugima. Statusna traka (engl. *Status bar*) prikazuje status projekta i samog razvojnog okruženja, kao i sve poruke ili upozorenja. Razvojno okruženje je moguće organizirati skrivanjem ili pomicanjem alatnih traka kako bismo dobili više prostora.

2.2. Kotlin

Kotlin je besplatni programski jezik otvorenoga koda. U početku je bio dizajniran za Java virtualni stroj i Android. On kombinira objektno orijentirane i funkcionalne značajke programiranja. Usredotočen je na interoperabilnost, sigurnost, jasnoću i podršku alata. Kotlin je nastao u JetBrainsu, tvrtki koja stoji iza IntelliJ IDEA, 2010. godine. Kotlin na prvi pogled izgleda kao sažetija i pojednostavljena verzija Jave. Programski kod u Kotlinu je kompatibilan s Javom tako da jedna klasa može biti pisana u Kotlinu a druga u Javi te će to uredno raditi. Iako je Kotlin punopravni funkcionalni programski jezik, on je sačuvao većinu objektno orijentirane prirode Jave kao alternativnog stila programiranja, što je vrlo zgodno pri pretvaranju postojećeg Java koda u Kotlin. [2]. Slika 3 prikazuje prednosti Kotlina nad Javom.

Features	Java	Kotlin
Fully OOP(Object-Oriented Programming)	Not pure OOP	Fully OOP
Null Safety	No	Yes
Checked Exception	Yes	No
Invariant Array	No	Yes
Smart Casts	No	Yes
Lambda Expression	No	Yes
Singletons Object	Yes	Easily create Singleton objects
Functional Reactive Programming	No	Yes

Slika 3: Prednosti Kotlina nad Javom [3]

2.3. SQLite

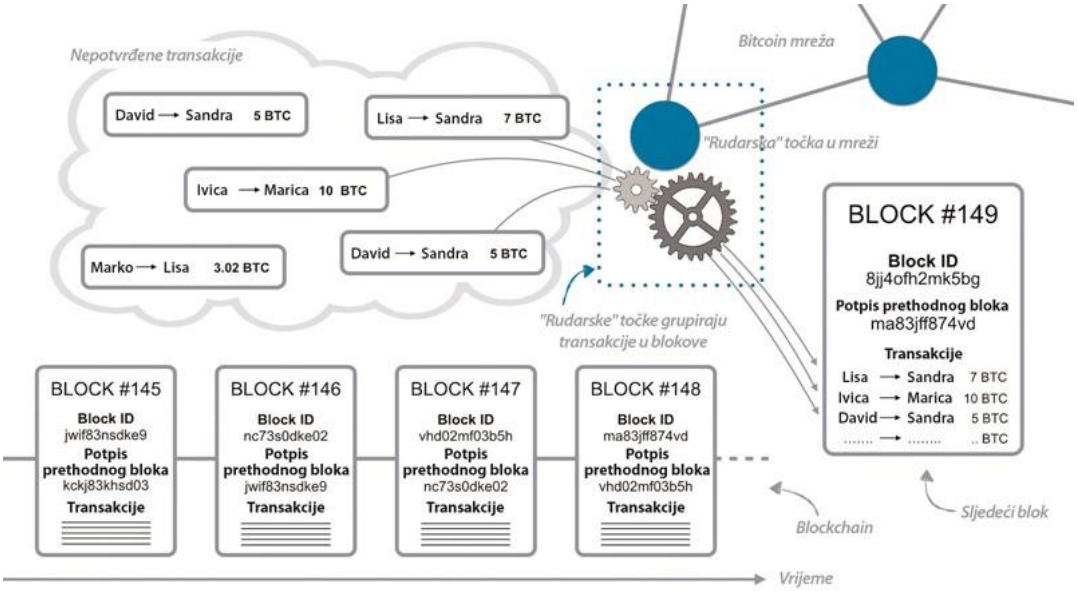
SQLite je programski paket koji pruža upravljanje relacijskom bazom podataka. Relacijske baze podataka koriste se za pohranjivanje korisničkih podataka u međusobno povezane tablice. Osim obrade podataka može obraditi složene naredbe koje kombiniraju podatke iz više tablica za generiranje izvješća. SQLite je lagan što se tiče složenosti postavljanja, administrativnih troškova i korištenja resursa. Neke od prednosti SQLite su da ne zahtijeva zaseban poslužiteljski proces ili sustav za rad, stvaranje instance baze podataka jednostavno je poput otvaranja datoteke, radi na svim platformama, transakcije su usklađene s ACID svojstvima i omogućuju siguran pristup iz više procesa i niti, tim SQLite se ozbiljno bavi testiranjem i provjerom koda,… Na kraju, SQLite pruža vrlo funkcionalno i fleksibilno okruženje relacijske baze podataka koje troši minimalne resurse i stvara minimalne probleme za programere i korisnike [4].

2.4. Github

Github je mrežna stranica i usluga zasnovana na oblaku (cloud) koja pomaže programerima pohranjivati i upravljati svojim kodom, kao i pratiti i kontrolirati promijene u kodu. Potrebno je znati dva principa, sustav za upravljanje verzijama i Git. Sustav za upravljanje verzijama pomaže programerima pratiti i upravljati promjenama. Kako softverski projekt raste, upravljanje verzijama postaje bitno. U slučaju da glavni razvojni programer želi raditi na jednom određenom dijelu koda, ne bi bilo sigurno ili učinkovito da izravno modificira glavni izvorni kôd. Umjesto toga, kontrola verzija omogućuje programerima siguran rad kroz grananje i spajanje. Uz grananje, programer duplicira dio izvornog koda te ga mijenja bez posljedica. Zatim, kada programer bude zadovoljan svojim kodom te kada kôd radi ispravno može spojiti taj dio koda natrag u glavni izvorni kôd. Sve se promjene prate te se mogu poništiti ako je potrebno. Git je distribuirani sustav kontrole verzija, što znači da je cijeli kôd i njegova povijest dostupna na računalu svakog razvojnog programera, što omogućuje jednostavno grananje i spajanje. Razvio ga je Linus Torvalds 2005 [5].

2.5. Bitcoin

Bitcoin se može gledati na dva načina, jedan je da je Bitcoin anonimna platna mreža koja je decentralizirana i distribuirana, a drugi je da je bitcoin virtualna valuta koju ta platna mreža koristi. Satoshi Nakamoto je pseudonim za osobu ili osobe koji su napisali originalnu Bitcoin specifikaciju i taj identitet vezuje se uz izum samog Bitcoina. Klasične valute kontrolira određena institucija, a to bi značilo da sva pravila oko te valute donosi jedno tijelo. Prednost Bitcoina je ta što je on jedna od najvećih decentraliziranih mreža, a to bi značilo da se valuta kao takva ne mora nikome povjeravati jer ne postoji središnje državno tijelo. Za decentralizacija Bitcoina je zaslužna tehnologija koju koristi pod nazivom lanac blokova.



Slika 4: Prikaz lanaca blokova i rudarenja novih [7]

Slika 4 prikazuje primjer lanca blokova i dodavanje novih blokova. Za svaki blok u lancu blokova se izračuna *hash* koji je jedinstven kao i otisak prsta. Ako se izmjeni bilo koji bit toga bloka identifikacijska oznaka se mijenja, a blok više nije valjan. Svaki sljedeći blok sadrži identifikacijsku oznaku prethodnoga bloka. U bloku se nalaze transakcije koje potvrđuje decentralizirana mreža nepoznatih računala na bazi specifičnoga algoritma. Transakcije u blokove stavljaju rudari koji će biti nagrađeni prilikom svake potvrde. Rudari su pojedinci koji rješavanjem složenih matematičkih algoritama dolaze do bitcoina pa bi bitcoin bio ništa drugo nego rezultat neke matematičke operacije. Svaki bitcoin je zaključan privatnim ključem s kojim se potvrđuje posjedovanje dok se u bloku nalazi samo javni ključ.

Na lancu blokova se zasniva većina kriptovaluta no on se ne mora koristiti samo u tome kontekstu već se može se koristiti za bilo koje digitalno zapisivanje. Broj bitcoina je ograničen na 21 milijun [6].

2.6. Biblioteke za programiranje aplikacija za Bitcoin

Android biblioteke strukturno su iste kao i modul aplikacije za Android što je vidljivo na slici 1 te uključuju sve što je potrebno za izradu aplikacije, a to su izvorni kôd, datoteke resursa, i manifest za Android. No, umjesto da se kompilira u APK koji radi na uređaju, Android biblioteka se kompilira u datoteku *Android Archive* (AAR) koja se koristiti kao *dependency* za modul aplikacije [8]. Za izradu novčanika koriste se dvije biblioteke, [9] *BitcoinJ* i [10] *bitcoin-kit-android*. *BitcoinJ* je biblioteka za rad s Bitcoin protokolom, a pruža mnoge mogućnosti bez potrebe za lokalnom kopijom Bitcoin Corea. Implementirana je u Javi te nije optimizirana za rad s mobilnim uređajima. Bitcoin-kit-android biblioteka je izrađena u Kotlinu isključivo za korištenje na mobilnim uređajima s Android operativnim sustavom. Obje datoteke podržavaju punu implementaciju SPV novčanika, slanje i primanje, P2PKH, P2PK, P2SH, P2WPKH, P2WPKH-SH, izračun naknade, generiranje ključne rečenice.

3. Inicijalizacija novčanika

3.1. Zaslon za učitavanje (Engl. *Splash screen*)

Slika 5: Logo

Slika 5 prikazuje izgled loga novčanika, a prikazuje se prilikom prvog pokretanja aplikacije. U pozadini će se nakon dvije sekunde pozvati provjera uređaja koja će provjeriti da operativni sustav nije modificiran. Modificiranje sustava, iako legalno, predstavlja sigurnosni propust na Android uređajima zbog mogućnosti dohvaćanja privatnih podataka od strane treće osobe stoga je pokretanje aplikacije dozvoljeno samo onim uređajima koji nisu modificirani.

```
// TCP/HTTP/DNS (depending on the port, 53=DNS, 80=HTTP, etc.)
private fun isOnline(): Boolean {
    val runtime = Runtime.getRuntime()
    try {
        val ipProcess = runtime.exec("/system/bin/ping -c 1 8.8.8.8")
        val exitValue = ipProcess.waitFor()
        return exitValue == 0
    } catch (e: IOException) {
        e.printStackTrace()
    } catch (e: InterruptedException) {
        e.printStackTrace()
    }
    return false
}
```

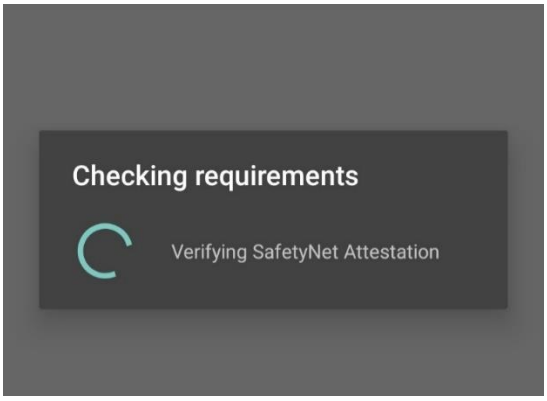
Ispis 1: Provjera dostupnosti mreže

Programskim kodom u ispisu 1 provjerava se je li dostupna internetska konekcija tako da se radi *ping* Google DNS poslužitelja. Metoda će vratiti pozitivan odgovor ako uspješno ostvari kontakt s poslužiteljem, a negativan u suprotnome slučaju.

```
Private fun checkSafetyNet() {
    if(isOnline()) {
        val safetyNet = SafetyNetAttestation()
        safetyNet.sendSafetyNetRequest(this@SplashActivity, this)
    } else{
        Toast.makeText(this, "No internet connection, application will be
closed.", Toast.LENGTH_LONG)
            .show()
        val runnable = Runnable {
            finishAffinity()
        }
        Handler().postDelayed(runnable, 2000)
    }
}
```

Ispis 2: Pozivanje metode za provjeru uređaja

Programski kôd u ispisu 1 prikazuje način pozivanja metode za provjeru uređaja (engl. *Safetynet attestation*) nakon što se obavi provjera dostupnosti mreže. Ako je mreža dostupna metodi se šalju dva parametra, a to su aktivnost i kontekst, u ostalom se ispisuje poruka korisniku te se aplikacija zatvori.



Slika 6: Zaslon prilikom provjere uređaja

Provjera uređaja je jednostavno i skalabilno Googleovo rješenje koje će provjeriti sigurnost uređaja te štititi osjetljive podatke unutar aplikacije. Slika 6 prikazuje zaslon nakon pozivanja metode `sendSafetyNetRequest` koja pokreće traku za napredak (engl. *Progress bar*) te će trajati sve dok provjera ne vrati rezultat ili dok ne istekne 60 sekundi. Nakon isteka aplikacija se prisilno zatvara jer se smatra da nije mogla doći do odgovora.

```
fun sendSafetyNetRequest(myActivity: Activity, myContext: Context) {
    this.context = myContext
    this.activity = myActivity
    //Show Loading Spinner
    loadingSpinner.startLoadingSpinner(
        context,
        "Verifying SafetyNet Attestation",
        "Checking requirements"
    )
    val nonceData = "Safety Net Sample: " + System.currentTimeMillis()
    val nonce: ByteArray = getRequestNonce(nonceData)

    val client: SafetyNetClient = SafetyNet.getClient(activity)
    val task: Task<AttestationResponse> = client.attest(nonce, API_KEY)
    task.addOnSuccessListener(activity, mSuccessListener)
        .addOnFailureListener(activity, mFailureListener)
}
```

Ispis 3: Metoda za provjeru uređaja

Programskim kodom u ispisu 3 prikazuje se način pozivanja metode `sendSafetyNetRequest`. Metoda se nalazi u klasi `SafetyNetAttestation` koja nema svoju aktivnost (engl. *Activity*) i kontekst (engl. *Context*) pa se primaju kao parametri iz prošle metode. Kontekst i aktivnost su metode koje daju pristup određenim resursima poput tema, dodataka, te funkcionalnosti koje pruža Android *framework*.

```
fun startLoadingSpinner(
    context: Context,
    message: String,
    title: String,
    exitTimer: Long = 60
) {
    nDialog = ProgressDialog(context)
    nDialog.setMessage(message)
    nDialog.setTitle(title)
    nDialog.isIndeterminate = false
    nDialog.setCancelable(false)
    nDialog.show()
    exitTimer(exitTimer)
}

//If stuck on LoadingSpinner for 60 seconds exit app
private fun exitTimer(exitTimer: Long) {
    handler.postDelayed(Runnable {
        exitProcess(-1)
    }, 1000 * exitTimer) //after amount of exitTimer seconds

}
```

Ispis 4: Inicijalizacija trake za napredak

Programski kôd u ispisu 4 prikazuje način na koji se inicijalizira traka za napredak. Kao parametri se primaju kontekst, naslov i poruka koji će biti prikazani, te izlani brojčanik koji će odrediti koliko će vremena aplikacija čekati odgovor metode za provjeru uređaja.

Prije poziva upita za provjeru uređaja postavlja se *nonce* koji štiti privatnu komunikaciju sprječavajući napad odgovorima (engl. *Replay attacks*) te kako bi zahtjev prema poslužitelju bio jedinstven. *Nonce* se kreira tako da se na nasumični tekst nadoda trenutno vrijeme i kao takav pretvori u bajtove. Ispred dobivenih bajtova nadodaju još 24 nasumično generirana bajta koja se pohranjuju u niz (engl. *Array*).

```
val client: SafetyNetClient = SafetyNet.getClient(activity)
val task: Task<AttestationResponse> = client.attest(nonce, API_KEY)
task.addOnSuccessListener(activity, mSuccessListener)
    .addOnFailureListener(activity, mFailureListener)
```

Ispis 5: Slanje zahtjeva za provjeru uređaja

Programskim kodom u ispisu 5 se prikazuje način slanja zahtjeva za provjeru uređaja Google poslužitelju. Za pristup poslužitelju koji vrši provjeru uređaja je potrebno napraviti API ključ koji se zatraži preko *gmail* korisničkog računa. Problem kod provjere uređaja putem poslužitelja je *Man In The Middle* napad. [11] Kod ovakve vrste napada u komunikacijskom kanalu između pošiljatelja i primatelja spaja se treća osoba koja uz pomoć alata manipulira komunikacijskim kanalom i neprimjetno nadzire komunikaciju. Nakon slanja upita se čeka rezultat od poslužitelja te se prema rezultatu provodi idući korak. Ako je operativni sustav na uređaju modificiran poslužitelj vraća poruka da provjera uređaja nije te se aplikacija neće pokrenuti. U slučaju pozitivnog odgovora se provjerava je li novčanik inicijaliziran te se korisnik prema rezultatu provjere preusmjeri na sljedeći zaslon. Provjera uređaja se radi prilikom svakog pokretanja zbog mogućnosti modificiranja operativnog sustava i nakon inicijalizacije.

```
var checker: Boolean = Operations().checkPinExists(activity)
Log.e(TAG, "PIN = $checker")
```

Ispis 6: Pozivanje metode za provjeru PIN koda

```
fun checkPinExists(activity: Activity): Boolean {
    val encryptedPin = Operations().getHashMap(PIN_LOC, activity)
    return encryptedPin != null
}
```

Ispis 7: Metoda za provjeru PIN koda

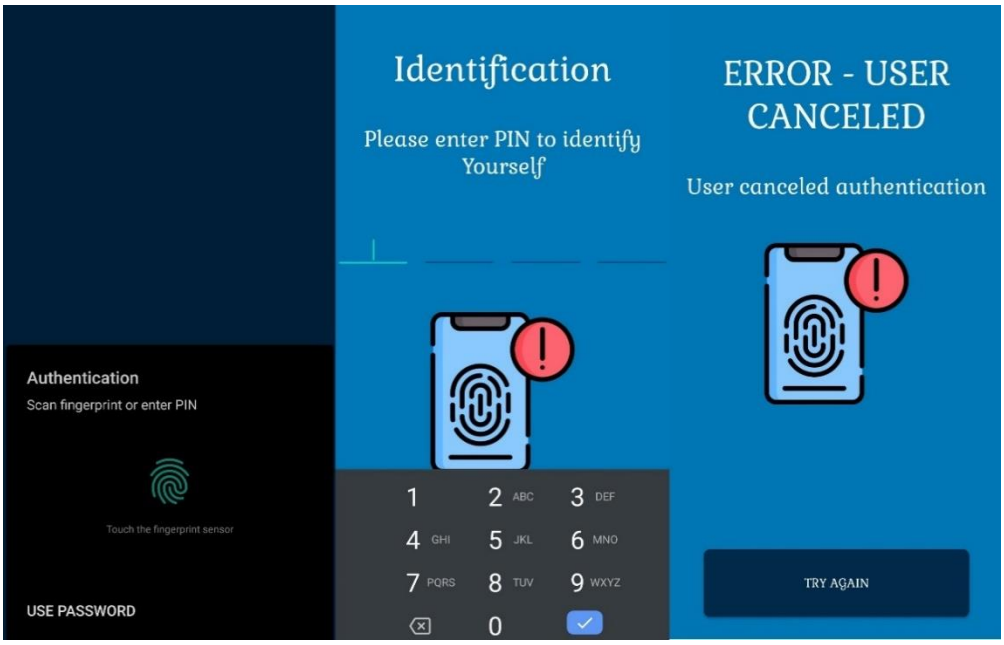
```
fun getHashMap(key: String?, activity: Activity): HashMap<String,
ByteArray>? {
    val prefs = PreferenceManager.getDefaultSharedPreferences(activity)
    val gson = Gson()
    val json = prefs.getString(key, "")
    val type: Type = object : TypeToken<HashMap<String?, ByteArray>?>()
{}.type
    return gson.fromJson(json, type)
}
```

Ispis 8: Metoda za dohvaćanje kriptirane mape

Programski kôd u ispisu 6 prikazuje način na koji se poziva metoda s kojom će se provjeriti ispravnost PIN koda. Uz provjeru PIN koda, za inicijalizaciju novčanika će se provjeriti postoji li ključna rečenica, šifra, i mreža. Programski kôd u ispisu 7 prikazuje način pozivanja metode koja će vratiti kriptiranu *hash* mapu u kojoj se nalazi PIN. Metodi je, uz *activity*, potrebno poslati naziv pod kojim je mapa spremljena u zajedničke postavke. U ovom trenutku dekriptiranje PIN koda nije potrebno jer se zna da PIN postoji ako postoji i mapa. Dohvaćanje mape, kao što je prikazano na programskom kodu u ispisu 8, se radi preko JSON formata. Kriptirana *hash* mapa kao takva se nije mogla spremiti u zajedničke postavke gdje se spremaju samo primitivni tipovi podataka poput teksta i broja.

3.2. Autentikacija korisnika

Autentikaciju korisnika se provodi prilikom prvog pokretanja aplikacije, a ako je aplikacija pokrenuta u pozadini autentikacija neće biti potrebna.



Slika 7: Prikaz zaslona za autentikaciju korisnika

Na slici 7 je prikazan zaslon za autentikaciju korisnika. Postoje dvije vrste autentikacije, a to su otiskom prsta i unosom PIN koda. U slučaju autentikacije otiskom prsta provjerava se dostupnost hardvera i postoji li u sustavu registrirani otisak prsta. Kod autentikacije putem korisničkog PIN koda se omogućuju samo brojevi na virtualnoj tipkovnici. Ako dođe do greške ispisuje se poruka koja će je pobliže objasniti. Nakon uspješne autentikacije provjerava se je li aplikacija inicijalizirana. Ova provjera se vrši zbog mogućnost plaćanja putem dubokog linka kako se, nakon uspješne autentikacije, aplikacija ne bi prebacila na zaslon za plaćanje iako nije inicijalizirana. U slučaju da je aplikacija inicijalizirana i otvorila se preko QR koda onda se, nakon uspješne autentikacije, dohvaća URL koji se rastavlja kako bi se dobili potrebni podaci za plaćanje.

```
if (intentData != null && intentData.data != null) {
    val data = intentData.data.toString()
    if (data.contains(":")) {
        var addressList = data.split(":")
        address = addressList[1]
        addressList = address.split("?")
        address = addressList[0]

    }
    if (data.contains("=")) {
        val amountList = data.split("=")
        amount = amountList[1]
        amount = (amount.toDouble() * 100000000).toString()
        amount = (amount.toDouble().toInt()).toString()
    }
}
```

Ispis 8: Metoda za dobivanje podataka za plaćanje iz URL adrese

Programski kôd u ispisu 8 prikazuje način na koji se iz URL adrese dobivene skeniranjem QR koda izvlače potrebni podaci. Ako aplikacija nije pokrenuta putem skeniranja QR koda otvara se početni zaslon bez potrebe za izvlačenjem podataka iz URL adrese.

3.3. Provjera inicijalizacije i postavljanje PIN koda

Slika 8: Početni zaslon postupka inicijalizacije

Slika 8 prikazuje početni zaslon postupka inicijalizacije koji prikazuje sljedeće potrebne korake. Primjer provjere se može vidjeti kroz programski kôd na ispisu 6. Nakon

toga će se u određenom trenutku morati dekriptirati podaci. Kreiranje PIN koda se radi u dva koraka, a to su upis PIN koda i ponavljanje upisa čiji uneseni brojevi moraju biti identični. Provjera parametara potrebnih za rad aplikacije se obavlja kako bi se slučaju prisilnog gašenja aplikacije moglo nastaviti od ispravnoga zaslona. Unos i provjera PIN koda se obavljaju na istome zaslonu preko *onActivityResult* metode.

```
when (checkSteps()) {
    1 -> {
        next.setOnClickListener {
            hideKeyboard()
            val intent = Intent(context, PinActivity::class.java)
            intent.putExtra(
                "description",
                getString(R.string.pin_identification_description)
            )
            intent.putExtra("keySecret", "pin")
            intent.putExtra("returnActivityResult", "1")
            startActivityForResult(intent, PIN_RETURNED)
        }
    }
```

Ispis 9: Poziv zaslona za kreiranje PIN koda sa specifičnim parametrima

Programskim kodom u ispisu 9 prikazuje se način na koji se kreira PIN putem `onActivityResult` metode. Parametri potrebni prije poziva metode za kreiranje PIN koda su naslov zaslona, mjesto pohranjivanja PIN koda u zajedničkim postavkama, i broj koji će reći upisuje li se PIN prvi ili drugi put. Izgled zaslona za unos PIN koda je prikazan na slici 7. Prilikom učitavanja zaslona postavljaju se određena ograničenja a to su da PIN mora biti 4 znaka dug i da se mogu unositi samo brojevi a ne i slova. Unos PIN koda se prati putem *observera* kako bi se znao sljedeći korak koji je potrebno izvršiti.

```
(0 until editTextArray.size)
    .forEach { i ->
        if (s === editTextArray[i].editableText) {
            if (s!!.isBlank()) {
                return
            }
            if (s.length >= 2) {//if more than 1 char
                val newTemp = s.toString().substring(s.length - 1,
s.length)
                if (newTemp != numTemp) {
                    editTextArray[i].setText(newTemp)
                } else {
                    editTextArray[i].setText(s.toString().substring(0,
s.length - 1))
                }
            } else if (i != editTextArray.size - 1) { //not last char
                editTextArray[i + 1].requestFocus()
                editTextArray[i + 1].setSelection(editTextArray[i +
1].length())
                return
            } else {
                verifyCode(testCodeValidity())
```

Ispis 10: Dio metode za praćenje unosa PIN koda

Postoje četiri mjesta za unos znamenki od kojih će se kasnije konstruirati PIN. Programski kôd u ispisu 10 prikazuje način na koji se prilikom popunjavanja jednog mjesta automatski prebacujemo na drugo dok ne dođemo do kraja. Prije vraćanja na prethodnu metodu konstruira se PIN koji se kriptira i pohranjuje u zajedničke postavke.

```
val pin =
    digOne.text.toString() + digTwo.text.toString() +
digThree.text.toString() + digFour.text.toString()
val secretKey = Cryptography().generateSecretKey(secretKeyLocation)
val encryptedTempPin = Cryptography().encryptMsg(pin, secretKey)
Operations().saveHashMap(
    secretKeyLocation,
    encryptedTempPin,
    this@PinActivity
)
```

Ispis 11: Pozivanje metode za kriptiranje i pohranjivanje PIN koda

Programskim kodom u ispisu 11 prikazuje se način na koji se PIN sprema u zajedničke postavke. Generira se tajni ključ (engl. *Secret key*) s kojim se kriptira PIN u *hash* mapu koja se pretvara u JSON i pohranjuje u zajedničke postavke na zadanu lokaciju.

```
fun generateSecretKey(secretKeyAlias: String): SecretKey {
    val keyGenerator = KeyGenerator.getInstance(
        KeyProperties.KEY_ALGORITHM_AES,
        "AndroidKeyStore"
    )
    val spec = KeyGenParameterSpec
        .Builder(secretKeyAlias, KeyProperties.PURPOSE_ENCRYPT or
KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
        .build()

    keyGenerator.init(spec)
    return keyGenerator.generateKey()
}
```

Ispis 12: Metoda za generiranje tajnog ključa

Programskim kodom u ispisu 12 prikazuje se način na koji se generira tajni ključ. Metoda prima parametar koji služi kao naziv pod kojim će se spremiti i kasnije dohvatiti. Nakon toga je potrebno inicijalizirati Android *keystore* i postaviti specifikacije za kriptiranje. Metoda vraća nasumično generirani tajni ključ.

```
fun encryptMsg(message: String, secret: SecretKey?): HashMap<String,
ByteArray>? {
    val cipher = Cipher.getInstance("AES/GCM/NoPadding")
    cipher.init(Cipher.ENCRYPT_MODE, secret)
    val ivBytes = cipher.iv
    val encryptedBytes = cipher.doFinal(message.toByteArray())
    val map = HashMap<String, ByteArray>()
    map["iv"] = ivBytes
    map["encrypted"] = encryptedBytes
    return map
}
```

Ispis 13: Metoda za kriptiranje PIN koda

Programskim kodom u ispisu 13 prikazuje se način kriptiranja i spremanja primljenoga teksta u *hash* mapu putem tajnim ključem. Postavljaju se načini enkripcije, a to su GCM algoritam s kojim se provjerava integritet i autentičnost podataka i AES algoritma za kriptiranje. Dodaje se nasumično generirani IV (engl. *Initialization vector*) zbog kojega će svaki tekst nakon enkripcije izgledati drugačije. Sada slijedi kriptiranje poruke koja se pretvara u niz bajtova te se dodaje u *hash* mapu uključujući i generiranim IV.

```
fun saveHashMap(key: String?, obj: Any?, activity: Activity) {
    val prefs = PreferenceManager.getDefaultSharedPreferences(activity)
    val editor = prefs.edit()
    val gson = Gson()
    val json: String = gson.toJson(obj)
    editor.putString(key, json)
    editor.apply()
}
```

Ispis 14: Metoda za pohranjivanje mape u zajedničke postavke

Programskim kodom u ispisu 14 prikazuje se način na koji se mapa pohranjuje u zajedničke postavke tako da se kao parametri primaju naziv pod kojom će se spremiti, objekt koji pohranjujemo, te aktivnost. Nakon toga se inicijalizira metoda za pristup, objekt se pretvara u JSON, te se pohranjuje u zajedničke postavke.

```
if (pinCorrect!!) {
    Operations().deleteFromSharedPreferences("tempPin", requireContext())
    parentFragmentManager.beginTransaction()
        .replace(R.id.welcome_container, MailFragment())
        .addToBackStack(null)
        .commit()
} else {
    Operations().deleteFromSharedPreferences("pin", requireContext())
    Operations().deleteFromSharedPreferences("tempPin", requireContext())
    description.text = getString(R.string.incorrect_pin_check)
    description.setTextColor(resources.getColor(R.color.red))
}
```

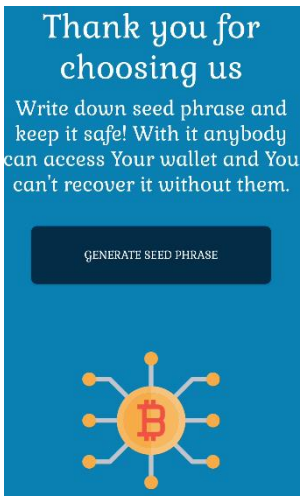
Ispis 15: Metoda za provjera kreiranog PIN koda

Programskim kodom u ispisu 15 provjerava se jesu li uneseni PIN kodovi identični, a u slučaju da jesu ponovljeni PIN se briše iz zajedničkih postavki te se prebacuje na idući zaslon. U suprotnome aplikacija briše oba PIN koda iz zajedničkih postavki te ispisuje grešku.

3.4. Odabir novčanika, dvostruka autentikacija, i odabir mreže

Slika 10: Zaslon za odabir vrste novčanika

Slika 10 prikazuje zaslon za odabir vrste novčanika prema kojemu se aplikacija preusmjerava na idući korak. Omogućuje se odabir novog ili oporavak postojećeg novčanika.



Slika 11: Početni zaslon za kreiranje novog novčanika

Slika 11 prikazuje zaslon na kojem se nalazi upozorenje koje korisnik treba pročitati prije generiranja ključne rečenice. Pomoću ključne rečenice se generiraju privatni i javni ključevi, a privatni ključ se koristi za stvaranje potpisa koji je potreban za trošenje bitcoina. Za generiranje ključne rečenice prvo kreiramo entropy a to je ništa drugo nego nasumični

niz znakova koji nemaju nikakav uzorak iz kojega je moguće zaključiti kako su se dobili te se često koristi kod kriptiranja ili generiranja privatnih ključeva.

```
private fun entropyGenerator(): ByteArray {
    val entropyLen = 16
    val entropy = ByteArray(entropyLen)
    val random = SecureRandom()
    random.nextBytes(entropy)
    return entropy
}
```

Ispis 22: Metoda za generiranje *entropyja*

Programski kôd u ispisu 22 prikazuje generiranje *entropyja* koji se sastoji od 128 nasumično generirani bita. Od 128 bita se kreira niz bajtova koji se pohranjuju u varijablu, a iz te varijable se preko klase `SecureRandom` izvlače nasumični podaci.

```
fun mnemonicGenerator(entropy: ByteArray = entropyGenerator()):
List<String> {
    lateinit var words: List<String>
    try {
        words = MnemonicCode.INSTANCE.toMnemonic(entropy)
    } catch (e: Exception) {
        Log.e("Error", "Couldn't generate mnemonic")
    }
    return words
}
```

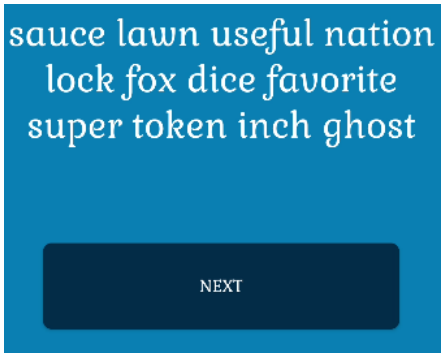
Ispis 23: Metoda za pozivanje generiranja ključne rečenice

Programski kôd u ispisu 23 prikazuje metodu koja poziva generiranje ključne rečenice, a kao parametar prima kreirani *entropy*. Dobivene nasumične riječi se pohranjuju u listu, a ako se ne mogu generirati prikazuje se poruka s greškom.

```
ArrayList<String> words = new ArrayList<>();
int nwords = concatBits.length / 11;
for (int i = 0; i < nwords; ++i) {
    int index = 0;
    for (int j = 0; j < 11; ++j) {
        index <<= 1;
        if (concatBits[(i * 11) + j])
            index |= 0x1;
    }
    words.add(this.wordList.get(index));
}

return words;
```

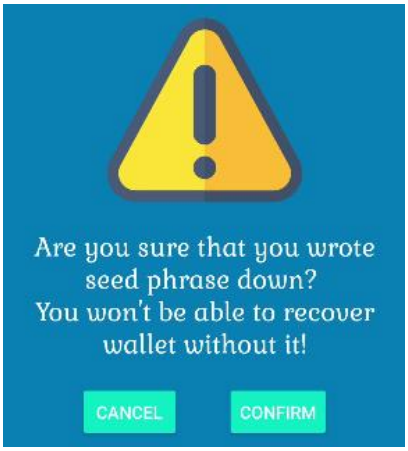
Ispis 24: Metoda za generiranje ključne rečenice

Programski kôd u ispisu 24 prikazuje način generiranja ključne rečenice prema BIP39 standardu dobivanjem nasumičnih riječi iz rječnika koji se sastoji od 2048 engleske riječi. Riječi se generiraju tako da se uzme *entropy* te izračuna njegov *hash*. Bajtovi *hasha* se pretvore u bitove koji se spajaju s originalnim bitovima *entropyja* te ih se dijeli u grupe od 11 bitova. Svaka grupa će predstavljati neku znamenku od 0 do 2047. Znamenka koji se uzme nasumičnim odabirom se pretvori u riječ te se spremi u listu dok se ne napuni s 12 do 24 riječi.

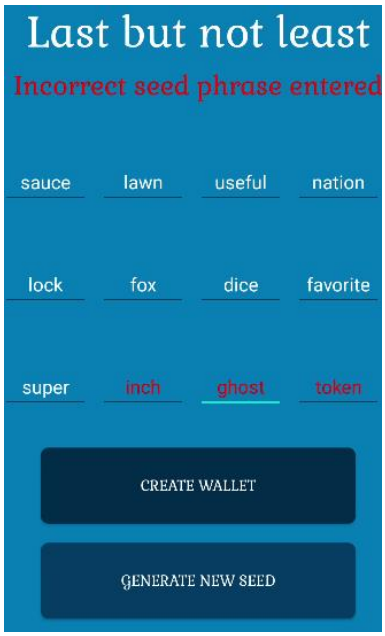


Slika 12: Zaslon za prikaz ključne rečenice

Slika 12 prikazuje način na koji se prikazuje ključna rečenica korisniku. Prije sljedećega koraka će se otvoriti dijalog s obavijesti koji će tražiti potvrdu prije mogućeg nastavka na sljedeći zaslon. Izgled dijaloga i obavijest su prikazani na slici 13.



Slika 13: Dijalog upozorenja prije generiranja ključne rečenice

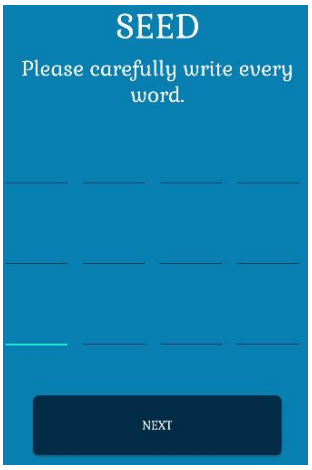


Slika 14: Zaslona za unos ključne rečenice

Prilikom upisa ključne rečenice postoji dvanaest odjeljaka, a u svaki odjeljak ide po jedna ključna riječ. Riječi se moraju upisati pravilnim redoslijedom, a ako se neka riječ upiše pogrešnim redoslijedom ili gramatički neispravno onda će se kratko vrijeme obojiti crvenom bojom kao upozorenje korisniku. Izgled zaslona za unos ključne rečenice je prikazan na slici 14.

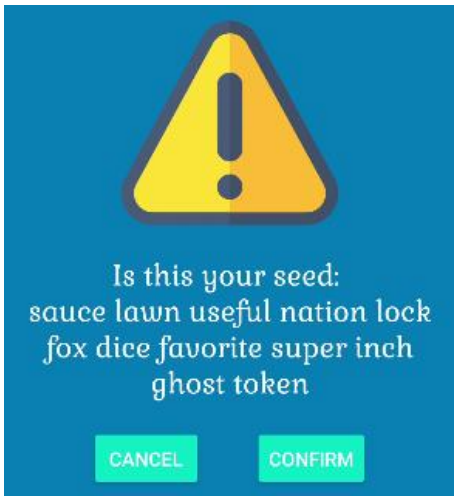
Slika 15: Predlaganje ključnih riječi

Slika 15 prikazuje izgled predlaganja ključnih riječi nakon dva upisana znaka. Riječi su nakon kreiranja spremljene u listu a polja su postavljena tako da se predlažu riječi iz te liste. Kada se upiše ispravna ključna rečenica, prije nastavka se kriptira u mapu te se pohranjuje u zajedničke postavke, a potrebna je kasnije zbog inicijalizacije novčanika.



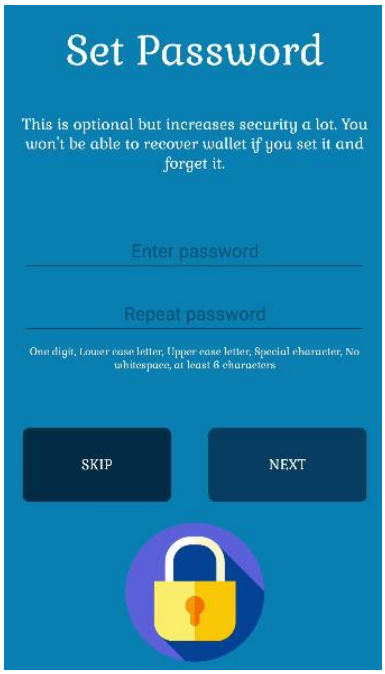
Slika 16: Zaslon za upis ključnih riječi kod oporavka novčanika

Slika 16 prikazuje izgled zaslona za upis ključnih riječi kod oporavka novčanika. Kod oporavka nema generiranja ključne rečenice kao kod kreiranja novog novčanika, a jedina provjera koja se vrši je da neki odjeljak nije ostao bez ključne riječi.



Slika 17: Zaslon za prikaz ključne rečenice kod oporavka novčanika

Slika 17 prikazuje dijalog koji služi za potvrdu ispravnosti unesene ključne rečenice, a prikazuje se prije nastavka na idući zaslon.



Slika 18: Prikaz zaslona za unos lozinke

Slika 18 prikazuje zaslon za unos lozinke, a to je ništa drugo nego dodatna ključna riječ koju je osmislio korisnik te se nadodaje na kraj ključne rečenice. Unos lozinke nije obavezan te ga je moguće preskočiti, a tako aplikacija zna da ne postoji. U slučaju preskakanja se kriptira prazna vrijednost u *hash* mapu koja se upisuje u zajedničke postavke.

```
when (checkPassword()) {
    1 -> Toast.makeText(context, "Both passwords are empty, why?",
Toast.LENGTH_SHORT)
         .show()
    2 -> Toast.makeText(context, "Please enter password first!",
Toast.LENGTH_SHORT)
         .show()
    3 -> Toast.makeText(context, "You didn't repeat password.",
Toast.LENGTH_SHORT)
         .show()
    4 -> Toast.makeText(context, "Password doesn't match.",
Toast.LENGTH_SHORT).show()
    5 -> Toast.makeText(context, "Password is too weak.",
Toast.LENGTH_SHORT).show()
    6 ->
saveEncryptedPassword(binding?.enterPassword?.text?.toString()?.trim())
}
```

Ispis 26: Provjera lozinke kod kreiranja novog novčanika

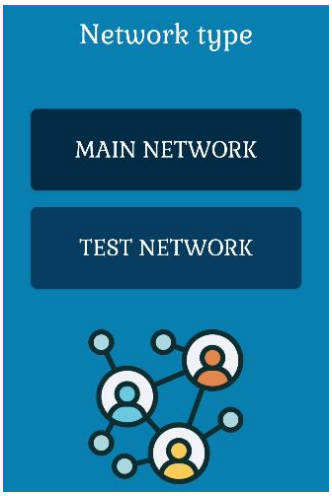
Prije nastavka na sljedeći zaslon provjerava se je li unesena lozinka odgovara uvjetima koji se vide na programskom kodu u ispisu 26, a provodi se samo kod kreiranja

novog novčanika. Kod oporavka se provjerava da polje za unos lozinke nije ostalo prazno jer postoji mogućnost da se korisnik prebacuje iz druge aplikacije koja nije imala iste zahtjeve kod generiranja lozinke.

```
fun isValidPassword(password: String?): Boolean {
    password?.let {
        val passwordPattern =
            "^(?=.*[0-9])(?=.*[a-z])(?=.*[A-
Z])(?=.*[@#$%^&+=*])(?=\\S+$).{6,}$"
        val passwordMatcher = Regex(passwordPattern)

        return passwordMatcher.find(password) != null
    } ?: return false
}
```

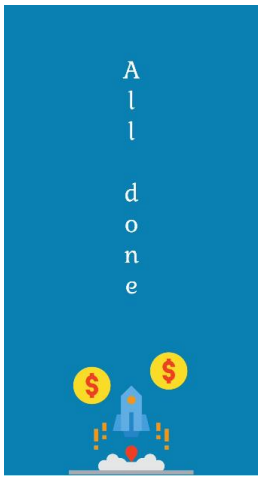
Ispis 27: Metoda za provjeru jačine šifre

Programski kôd u ispisu 27 prikazuje način na koji vrši provjerava jačine upisane šifre. Lozinka treba imati jedno veliko i malo slovo, najmanje šest znakova, znamenku, te jedan specijalni znak.



Slika 19: Prikaz zaslona za odabir mreže

Na kraju, prije kreiranja novčanika se odabire mreža s kojom će novčanik raditi, a na slici 19 je prikazan izgled zaslona u kojem se vrši odabir. Postoje dvije vrste mreže koje novčanik podržava, a to su testna i glavna. Nakon odabira, u varijablu koja se nalazi u zajedničkim postavkama se upisuje „*Main*“ za glavnu ili „*Test*“ za testnu mrežu.



Slika 20: Prikaz zaslona za kraj inicijalizacije

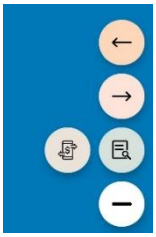
Na slici 20 je prikazan kraj inicijalizacije novčanika, a aplikacija se prebacuje na početni zaslon nakon dvije sekunde.

4. Rad s novčanikom

4.1. Početni zaslon

Slika 21: Prikaz početnoga zaslona

Slika 21 prikazuje izgled početnoga zaslona na kojemu se nalazi gadget zadnje transakcije, gadget detalja sinkronizacije s lancem blokova, iznos novčanika, i prečaci.



Slika 22: Prečaci na početnom zaslonu

Slika 22 prikazuje prečace na početnom zaslonu. Glavna ikona prečaca mijenja ikonu u ovisnosti o tome jesu li su prečaci minimizirani ili maksimizirani. Prečaci koji postoje su minimiziranje i maksimiziranje gadgeta, prelazak na zaslon za plaćanje, i prelazak na zaslon za primanje.

Prije prikaza početnog zaslona prvo se mora odraditi inicijalizacija novčanika koja odvija u pozadini. Metoda za inicijalizaciju novčanika prima potrebne parametre koji su prethodno spremljeni u zajedničke postavke. Novčanik se inicijalizira samo prilikom prvog pokretanja aplikacije i ostaje inicijaliziran sve dok se aplikacija ne zatvori, a tako aplikacija stalno osluškuje šta se događa na mreži. Prvi parametar potreban za inicijalizaciju je ključna rečenica koja se dohvaća iz kriptirane mape, a uz ključnu rečenicu treba još dohvatiti i šifru koja će se dodati na kraj ključne rečenice.

```
fun getMnemonic(activity: Activity): List<String> {
    val encryptedSeed = activity.let { Operations().getHashMap(SEED_LOC,
it) }
    val secretSeedKey = Cryptography().getSecretKey(SEED_LOC)
    val decryptSeed =
        Cryptography().decryptMsg(encryptedSeed,
secretSeedKey)?.decodeToString()!!
    return decryptSeed.split(" ")
}
```

Ispis 28: Metoda za dohvaćanje ključne rečenice

Programski kôd u ispisu 28 prikazuje način na koji se dohvaća ključna rečenica iz zajedničkih postavki, a na isti način se dohvaća i lozinka samo s drugim parametrima. Prvo se dohvati *hash* mapa iz zajedničkih postavki te se pohranjuje u varijablu pod nazivom *encryptedSeed*.

```
fun getHashMap(key: String?, activity: Activity): HashMap<String,
ByteArray>? {
    val prefs = PreferenceManager.getDefaultSharedPreferences(activity)
    val gson = Gson()
    val json = prefs.getString(key, "")
    val type: Type = object : TypeToken<HashMap<String?, ByteArray>?>()
{}.type
    return gson.fromJson(json, type)
}
```

Ispis 29: Metoda za dohvaćanje kriptirane mape

Programski kôd u ispisu 29 prikazuje metodu s kojom se dohvaća kriptirana mapa. Pošto se prilikom enkripcije kriptirani objekt pretvorio u JSON sada se radi obrnuti postupak gdje se dohvaća JSON te se pretvara u objekt. Nakon dohvaćanja kriptirane mape potrebno je dohvatiti tajni ključ s kojim se mapa kriptirala iz Android *keystorea*.

```
fun getSecretKey(secretKeyAlias: String): SecretKey {
    val keyStore = KeyStore.getInstance("AndroidKeyStore").apply {
load(null) }
    val secretKeyEntry = keyStore.getEntry(secretKeyAlias, null) as
KeyStore.SecretKeyEntry
    return secretKeyEntry.secretKey
}
```

Ispis 29: Metoda za dohvaćanje tajnog ključa

Programski kôd u ispisu 29 prikazuje metodu za dohvaćanja tajnog ključa. Na početku metode se inicijalizira *keystore* s kojim se dohvaća ključ prema imenu po kojem je spremljen. Android *keystore* dopušta pohranjivanje kriptiranih ključeva u kontejner operativnog sustava, a samo aplikacija koja ih je spremila može dobiti pravo pristupa.

```
fun decryptMsg(myHashedMap: HashMap<String, ByteArray>?, secret:
SecretKey?): ByteArray? {
    val encryptedBytes = myHashedMap?.get("encrypted")
    val ivBytes = myHashedMap?.get("iv")
    val cipher = Cipher.getInstance("AES/GCM/NoPadding")
    val spec = GCMParameterSpec(128, ivBytes)
    cipher.init(Cipher.DECRYPT_MODE, secret, spec)
    return cipher.doFinal(encryptedBytes)
}
```

Ispis 30: Metoda za dekriptiranje ključne rečenice

Nakon dohvaćanja *hash* mape i tajnog ključa programski kôd u ispisu 30 prikazuje način na koji se dekriptira ključna rečenica. Prvo se dohvaća nasumično generirani IV i *cipher* koji su se koristili kod kriptiranja mape. *Chiper* sadrži algoritme za enkripciju, a to su AES i GCM algoritam. Nakon toga se postavljaju specifikacije za GCM algoritam s kojim se provjerava integritet i autentičnost podataka. Na kraju se dekriptira mapa koja će vratiti niz bajtova koji se kasnije mogu pretvoriti u tekst. Sada kada su ključna rečenica i lozinka dekriptirani potrebno je dohvatiti postavku broja potvrda transakcije (engl. *Transaction confirmations*). Kada rudar uvrsti transakciju u blok to znači da ima jednu potvrdu, a svaki sljedeći blok koji se veže na taj blok je dodatna potvrda transakcije. Preporuča se šest potvrda (pet vezanih blokova) kako bi transakcija bila nepovratna. U postavkama aplikacije je dodana mogućnost mijenjanja broja potvrda transakcije, a šest je zadana. Na kraju prije same inicijalizacije novčanika dohvaćaju se podaci koji govore je li se radi oporavak novčanika i koji je tip mreže s kojom će novčanik raditi.

```
viewModel.init(passphrase, BitcoinAPI().getNetworkType(this) ?: "", words,
    walletRecovery, confirmationsInt)
```

Ispis 31: Poziv inicijalizacije novčanika

Programski kôd u ispisu 31 prikazuje pozivanje metode koja se nalazi u *ViewModelu* s kojom se inicijalizira novčanik uz određene parametre, a to su lozinka, vrsta mreže, ključna rečenica, varijabla koja govori je li se radi oporavak novčanika, i broj potvrda transakcije. Novčanik kojega inicijaliziramo radi na SPV (*Simplified Payment Verification*) principu, a opisan je u BIP 37 standardu. Ovim načinom se omogućuje aplikaciji da provjeri je li se transakcija nalazi u lancu blokova bez da preuzme sve informacije iz blokove krećući od prvoga. To će učiniti tako da će preuzeti samo zaglavlje blokova i koristiti *merkle tree* tehniku što je ništa drugo nego struktura stvorena grupiranjem svih transakcija u parovima i njihovim heširanjem dok se ne dođe do *merkle roota*. Na ovaj način se ne vjeruje čvoru na kojega se novčanik spojio. Naravno i SPV novčanik ima nedostatak, a to je da se sve adrese vezane uz novčanik šalju nepoznatome čvoru te se tako gubi privatnost.

```
fun init(
    passphrase: String,
    networkType: String,
    words: List<String>,
    walletRecovery: Boolean = false,
    confirmations: Int = 6
) {

    val syncMode = if (walletRecovery) {
        BitcoinCore.SyncMode.Full()
    } else {
        BitcoinCore.SyncMode.NewWallet()
    }

    when (networkType) {
        "TEST" -> params = BitcoinKit.NetworkType.TestNet
        "MAIN" -> params = BitcoinKit.NetworkType.MainNet
    }

    bitcoinKit = BitcoinKit(
        appContext,
        words,
        passphrase,
        walletId,
        params,
        syncMode = syncMode,
        bip = bip,
        confirmationsThreshold = confirmations
    )
    bitcoinKit.listener = this

    networkName = bitcoinKit.networkName
    balance.value = bitcoinKit.balance

    bitcoinKit.transactions().subscribe { txList: List<TransactionInfo> ->
        transactions.value = txList
    }.let {
        disposables.add(it)
    }

    lastBlock.value = bitcoinKit.lastBlockInfo
    state.value = bitcoinKit.syncState

    started = false
}
```

Ispis 32: Metoda za inicijalizacija novčanika

U *init* metodi se kao parametri primaju lozinka, tip mreže, ključna rečenica, varijabla za oporavak novčanika, te broj potvrda transakcija. Pomoću ove metode se inicijalizira novčanik, a ona se poziva prilikom prvog pokretanja aplikacije. Metoda prvo radi provjeru tipa novčanika kako bi novčanik znao od kojega će bloka krenuti sa sinkronizacijom, a nakon toga se spaja na odabranu vrstu mreže. Inicijalizira se `bitcoinKit` iz biblioteke bitcoin-kit-android uz sve te parametre te se postavljaju *listeneri* koji će osluškivati rezultate transakcija. Proces je vidljiv na programskom kodu u ispisu 32.

4.2. Oporavak novčanika

Kod oporavka novčanika postoje određeni problemi koji se nameću: Kako znati da je transakcija unutar bloka ako spremamo samo zaglavlja jer se koristi SPV tip novčanika? Na koje će se čvorove aplikacija spojiti? Kako generirati adrese po dubini i širini? Za koliko adresa proširiti pretragu ako se nađe transakcija vezana uz novčanik u bloku? Kod oporavka novčanika prvo se dohvaća korijenski ključ (engl. *Root key*).

```
public static HDKey createRootKey(byte[] seed) throws HDDerivationException
{
    if (seed.length < 16)
        throw new IllegalArgumentException("Seed must be at least 128
bits");
    byte[] i = Utils.hmacSha512("Bitcoin seed".getBytes(), seed);
    byte[] il = Arrays.copyOfRange(i, 0, 32);
    byte[] ir = Arrays.copyOfRange(i, 32, 64);
    BigInteger privKey = new BigInteger(1, il);
    if (privKey.signum() == 0)
        throw new HDDerivationException("Generated master private key is
zero");
    if (privKey.compareTo(ECKey.ecParams.getN()) >= 0)
        throw new HDDerivationException("Generated master private key is
not less than N");
    return new HDKey(privKey, ir, null, 0, false);
}
```

Ispis 33: Metoda za kreiranje korijenskog ključa iz ključne rečenice

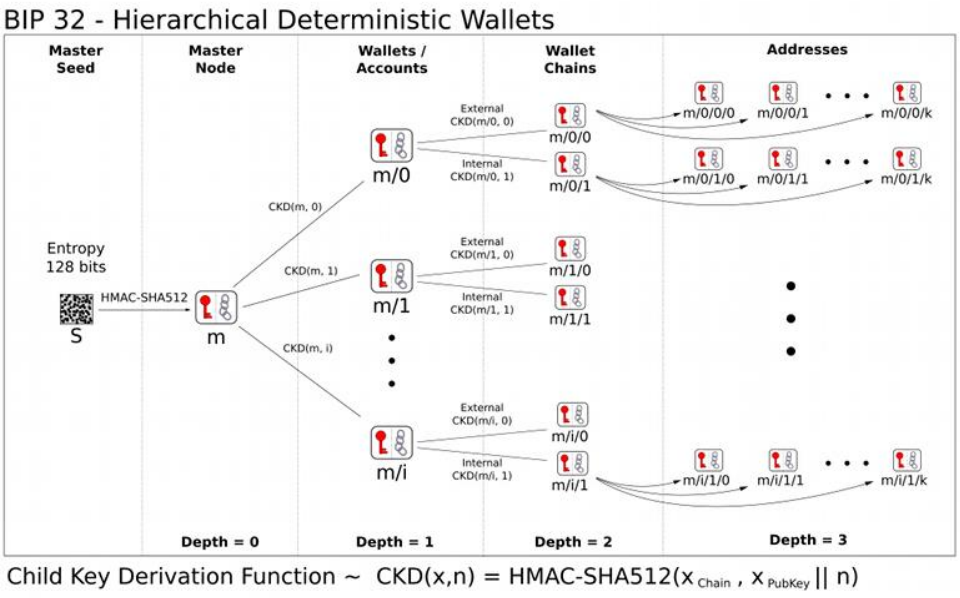
Programski kôd u ispisu 33 prikazuje način na koji se kreira korijenski ključ opisan u BIP32 standardu. Ključna rečenica mora biti između 128 i 512 bajtova kako bi se mogao dobiti korijenski ključ te je to prva provjera koja se radi. Nakon toga se računa HMAC-SHA512 iz ključne rečenice i fiksne rečenice pretvorene u bajtove. HMAC-SHA512 se temelji na SHA512 *hash* funkciji i kodu za provjeru autentičnosti poruka (HMAC). Prvo se promiješa ključna i fiksna rečenica, napravi *hash* rezultata, i nakon toga se opet promiješa rezultat i tajna rečenica te se tako dobije dvostruki *hash*. Sada se podijeli rezultat u dvije sekvence po 32 bajta i tako se dobije lijeva i desna sekvenca. Lijeva sekvenca se koristi kao *master* ključ dok se desna koristi kao *master chain* kôd koji je ništa drugo nego deterministično kreirani korijen ključa koji dobije od roditelja. Ako je dužina *master* ključa 0 ili veća od 32 bajta mora se vratiti greška. Sada kada je korijenski ključ generiran mogu se generirati i adrese za plaćanje, ostatak, te privatne adrese prema BIP44 standardu. Izgled putanje za kreiranje adresa: „*m* / *purpose'* / *coin_type'* / *account'* / *change* / *address_index*„

a putanja koju koristi novčanik je m/44'/0'/0'. Za *purpose* se uvijek koristi 44 jer je to fiksna oznaka za Bitcoin prema BIP43 standardu.

```
fun receiveHDPublicKey(account: Int, index: Int): HDPublicKey {
    return HDPublicKey(index = index, external = true, key =
privateKey(account = account, index = index, chain = 0))
}

fun changeHDPublicKey(account: Int, index: Int): HDPublicKey {
    return HDPublicKey(index = index, external = false, key =
privateKey(account = account, index = index, chain = 1))
}

fun privateKey(account: Int, index: Int, chain: Int): HDKey {
    return privateKey(path =
"m/$purpose'/$coinType'/$account'/$chain/$index")
}

fun privateKey(account: Int, index: Int, external: Boolean): HDKey {
    return privateKey(account, index, if (external) Chain.EXTERNAL.ordinal
else Chain.INTERNAL.ordinal)
}

private fun privateKey(path: String): HDKey {
    return hdKeychain.getKeyByPath(path)
}
```

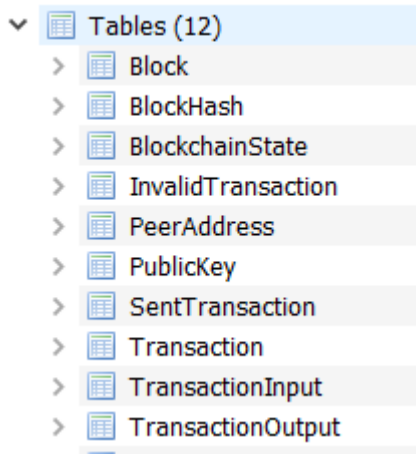
Ispis 34: Metoda za generiranje ključeva prema određenim parametrima [10]

U ovisnosti o vrsti ključa programski kôd u ispisu 34 prikazuje način dohvaćanja ključa.



Slika 23: Primjer generiranja ključeva za hijerarhijsko deterministički (engl. *Hierarchical Deterministic)* novčanik [12]

Slika 23 prikazuje način na koji se generiraju ključevi u hijerarhijsko deterministčkom novčaniku prema BIP32 standardu. Novčanik neće podržavati više računa pa će „*Wallets* / *Accounts*“ parametar uvijek biti 0. „*Wallet chains“* će ovisiti o tipu ključa koji će se generirati, za plaćanje ili ostatak. Sljedeći parametar su adrese koje se generiraju u širinu na trećoj dubini. Novčanik će prilikom inicijalizacije generirati 100 adresa za plaćanje te ih po potrebi koristiti redom, a adresa se mijenja samo kada se prethodna potroši. Na ovaj način se lakše prate transakcija pomoću *bloom* filtera i vrši oporavak novčanika.



Slika 24: Primjer tablica SQLite baze podataka

Slika 24 prikazuje SQLite bazu podataka s odgovarajućim tablicama koje su kreirane prilikom inicijalizacije, a potrebne su za funkcioniranje SPV novčanika. Sve vrijednosti unutar polja su kriptirane kako bi se osigurala sigurnost podataka.

```
fun getKeyByPath(path: String): HDKey {
    var key = privateKey

    var derivePath = path
    if (derivePath == "m" || derivePath == "/" || derivePath == "") {
        return key
    }
    if (derivePath.contains("m/")) {
        derivePath = derivePath.drop(2)
    }
    for (chunk in derivePath.split("/")) {
        var hardened = false
        var indexText: String = chunk
        if (chunk.contains("'")) {
            hardened = true
            indexText = indexText.dropLast(1)
        }
        val index = indexText.toInt()
        key = HDKeyDerivation.deriveChildKey(key, index, hardened)
    }

    return key
}
```

Ispis 35: Metoda za generiranja adresa u dubinu i širinu

Programski kôd u ispisu 35 prikazuje način deriviranja adresa prema zadanome putu u određenu dubinu i širinu.

Novčanik za dobivanje IP adresa poslužitelja koristi *dnsSeed* koji je ništa drugo nego DNS lista koju predstavlja zajednica. Prilikom poziva određenog *dnsSeeda* dohvate se IP adrese svih poslužitelja koji na sebi imaju Bitcoin pune čvorove. Novčanik se prilikom svake inicijalizacije spaja na 5 različitih čvorova kako se ne bi narušila privatnost.

```
dnsSeeds = new String[] {
        "testnet-seed.bitcoin.jonasschnelli.ch", // Jonas Schnelli
        "seed.tbtc.petertodd.org",               // Peter Todd
        "seed.testnet.bitcoin.sprovoost.nl",     // Sjors Provoost
        "testnet-seed.bluematt.me",              // Matt Corallo
};
```

Ispis 36: *dnsSeed* liste koje predstavlja zajednica

Na programskom kodu u ispisu 36 su prikazane DNS liste koje predstavlja zajednica za testnu Bitcoin mrežu koje kada se pozovu vraćaju IP adrese punih čvorova. Jedan upit iz

novčanika se šalje na više čvorova dobivenih putem *dnsSeeda* nakon čega se odgovori izmiješaju te se na kraju vrati jedan nasumičan odgovor novčaniku. Tako se dobiva na privatnosti jer se transakcije ne mogu pratiti do određenog čvora i s određenog čvora do novčanika. Način za provjeru ispravnost DNS servera je da se kreira puni čvor na vlastitom računalu čija bi se IP adresa nakon određenog vremena trebala naći na vraćenoj listi *dnsSeeda*. Port koji se koristi je 18333 koji je standardiziran za testnu Bitcoin mrežu.

Transakcije se prate pomoću *bloom* filtera koji su važni za funkcioniranje SPV tipa novčanika te su definirani u BIP37 standardu. *Bloom* filteri omogućuju SPV novčaniku da ograniči količinu podataka o transakcijama koje primaju s punih čvorova na samo one transakcije koje utječu na novčanik. Filter se kreira samo za adrese koje se generiraju putem ključne rečenice unesene u novčanik, a šalje se čvoru koristeći P2P protokol definiran u BIP37 standardu koji zatraži poseban oblik blokova (*merkle blocks*). Čvor će vratiti samo one transakcije koje su poslane filterom, zaglavlje bloka, i djelomičnu *merkle* granu (*branch*) koja povezuje svaku odgovarajuću transakciju s *merkle rootom* u zaglavlju bloka. Uz sve ostalo se primaju i najave nepotvrđenih transakcija, a zadnja se pokazuje u gadgetu na početnome zaslonu.

```
val watchedTransactionManager = WatchedTransactionManager()
bloomFilterManager.addBloomFilterProvider(watchedTransactionManager)
bloomFilterManager.addBloomFilterProvider(publicKeyManager)
bloomFilterManager.addBloomFilterProvider(pendingOutpointsProvider)
bloomFilterManager.addBloomFilterProvider(irregularOutputFinder)
```

Ispis 37: Primjer koda za registriranje *bloom* filtera [10]

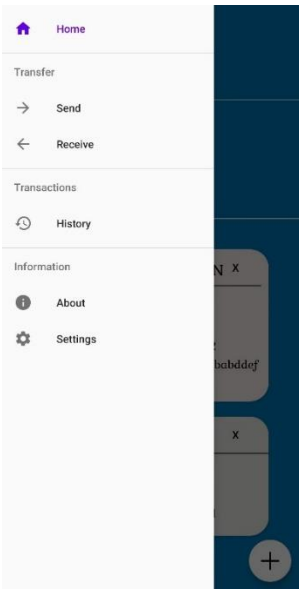
Programski kôd u ispisu 37 prikazuje način na koji se registriraju *bloom* filteri za određenu vrstu adresa.

```
val localDownloadedBestBlockHeight: Int
    get() = storage.lastBlock()?.height ?: 0

val localKnownBestBlockHeight: Int
    get() {
        val blockHashes = storage.getBlockchainBlockHashes()
        val headerHashes = blockHashes.map { it.headerHash }
        val existingBlocksCount =
headerHashes.chunked(sqliteMaxVariableNumber).map {
            storage.blocksCount(it)
        }.sum()

        return localDownloadedBestBlockHeight.plus(blockHashes.size -
existingBlocksCount)
    }
```

Ispis 38: Metoda za postavljanje početnoga bloka [10]

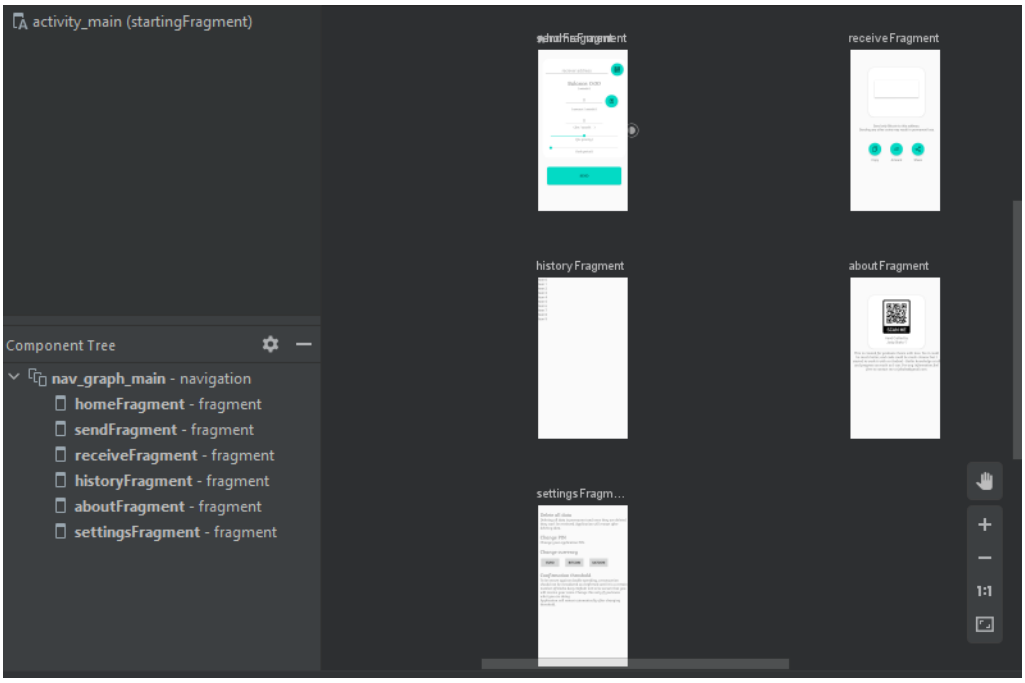
Programski kôd u ispisu 38 prikazuje način postavljanja početnoga bloka prilikom pokretanja aplikacije za sinkronizaciju tako da dohvaća zadnji blok iz SQLite baze podataka koja se nalazi na uređaju. Koristeći *seed* i podešenu putanju generira se prvih 20 adresa koje se dodaju u *bloom* filter. Oporavak kreće od zadnjeg bloka dohvaćenog programskim kodom u ispisu 38. Skenira se svaki idući blok u lanac blokova sve do zadnjeg. Pomoću *bloom* filtera se pitaju čvorovi je li postoji transakcija u bloku koja uključuje te adrese, a u slučaju pozitivnog odgovora poveća se brojač transakcija za jedan, spremi se visina bloka i transakcija u SQLite bazu podataka, te se generira još 20 idućih adresa koje se opet dodaju u *bloom* filter kako bi se proširila pretragu. Vrijeme trajanja ove operacije ovisi o tome koliko je dug lanac blokova te broju iskorištenih adresa.

4.3. Navigacijski izbornik, prečaci, i gadgeti



Slika 25: Prikaz navigacijskog izbornika

Slika 25 prikazuje izgled navigacijskog izbornika koji će sadržavati prečace za početni zaslon, slanje, primanje, povijest, informacije o aplikaciji, i postavke. Navigacijski izbornik je podijeljen po sekcijama, a to su transferi, transakcije, i informacije. Klikom na određeni prečac u navigacijskom izborniku se mijenja stari fragment za novi u ovisnosti o tome na što se klikne.



Slika 26: Kreiranje navigacijskoga grafa

Navigacijski izbornik je napravljen preko navigacijskoga grafa, a on je ništa drugo nego XML datoteka koja sadrži sve informacije vezane za navigaciju te se prikazuje na slici 26. Navigacijski graf se registrira na početnoj aktivnosti kako bi se prikazao na svim fragmentima. Navigacijski graf novčanika sadrži informacije o fragmentima za slanje, plaćanje, povijest, informacije o aplikaciji, i postavke.

Slika 27: Dio početnog zaslona za prikaz iznosa

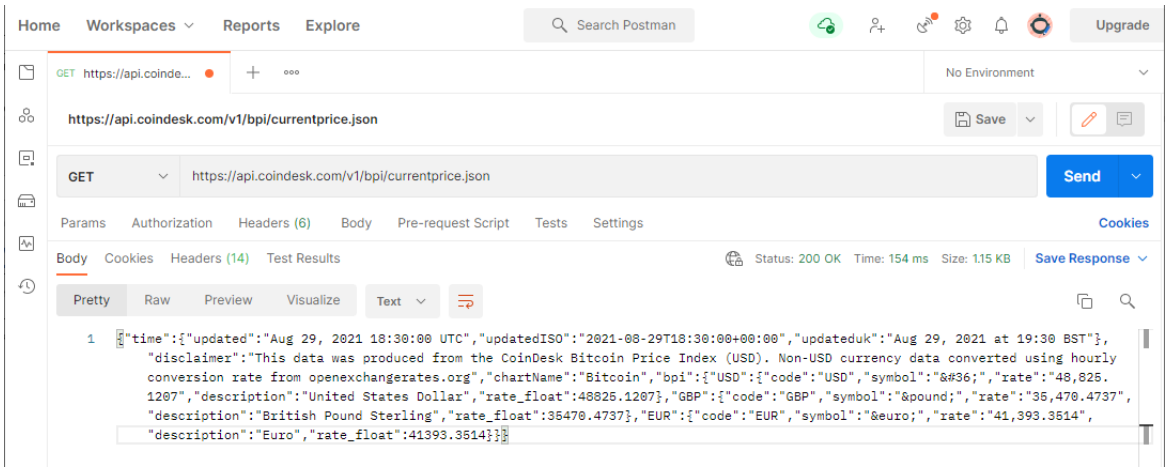
Prilikom prvog pokretanja aplikacije iznos se na početnom zaslonu prikazuje u eurima te je prikazan na slici 27, a iznos govori da postoje adrese u novčaniku za koje postoji privatni ključ kojim se dokazuje vlasništvo. U postavkama postoji način da se promjeni zadana valuta te je moguće odabrati euro, bitcoin ili *satoshi*.

```
val currentCurrency =
Operations().readFromSharedPreferences(requireActivity(), "currency")

viewModel.balance.observe(viewLifecycleOwner, { balance ->
    when (balance) {
        null -> {
            when(currentCurrency) {
                "0" -> {
                    binding?.balanceTextView?.text = "0 €"
                    binding?.unspendableBalance?.text = "0 €"
                }
                "1" -> {
                    binding?.balanceTextView?.text = "0 ₿"
                    binding?.unspendableBalance?.text = "0 ₿"
                    binding?.unspendableBalance?.text = "0 ₿"
                }
                "2" -> {
                    binding?.balanceTextView?.text = "0 ș"
                    binding?.unspendableBalance?.text = "0 ș"
                }
            }
        }
        else -> {
            when(currentCurrency) {
                "0" -> {

loadBitcoinPrice(NumberFormatHelper.cryptoAmountFormat.format(balance.spend
able / 100_000_000.0))

loadBitcoinPrice(NumberFormatHelper.cryptoAmountFormat.format(balance.unspe
ndable / 100_000_000.0), false)
                }
                "1" -> {
                    val bitcoinValue =
NumberFormatHelper.cryptoAmountFormat.format(balance.spendable /
100_000_000.0)
                    val unspendableBitcoinValue =
NumberFormatHelper.cryptoAmountFormat.format(balance.unspendable /
100_000_000.0)
                    binding?.balanceTextView?.text = "$bitcoinValue ₿"
                    binding?.unspendableBalance?.text =
"$unspendableBitcoinValue ₿"
                }
                "2" -> {
                    val satoshiValue = balance.spendable
                    val unspendableSatoshiValue = balance.unspendable
                    binding?.balanceTextView?.text = "$satoshiValue ș"
                    binding?.unspendableBalance?.text =
"$unspendableSatoshiValue ș"
                }
            }
        }
    }
})
```

Ispis 40: Metoda za prikaz valute u odnosu na zadanu postavku

Programski kôd u ispisu 40 prikazuje način na koji će se dohvatiti i postaviti odabrana valute na početni zaslon. Prvo se dohvaća trenutna valuta u privremenu varijablu iz zajedničkih postavki te se prema tome postavlja početni zaslon. Za prikaz stanja novčanika u valuti *satosh* ne treba raditi nikakvu konverziju jer je to valuta s kojom biblioteka koju koristimo radi. Za izračun broja bitcoina se dohvati broj *satoshija* te podjeli s 100,000,000 nakon čega se iznos postavlja u željeni format. Za prikaz stanja novčanika u eurima broj

satoshija se dijeli s 100,000,000 kako bi se dobio broj bitcoina. Nakon toga se poziva metoda koja će vratiti trenutnu tržišnu bitcoina u eurima.



Slika 28: Test dohvaćanja podataka preko API sučelja koristeći Postman

Na slici 28 je prikazan test dohvaćanja cijene u eurima koristeći Postman i CoinDesk API. CoinDesk API je besplatan servis koji vraća trenutnu cijenu bitcoina u više valuta.

```
private fun loadBitcoinPrice(currentValue: String = "", spendable: Boolean
= true) {
    val request: Request = Request.Builder().url(url).build()
    var eurRate: String
    okHttpClient.newCall(request).enqueue(object : Callback {
        override fun onFailure(call: Call, e: IOException) {

        }

        @SuppressLint("SetTextI18n")
        override fun onResponse(call: Call, response: Response) {
            val json = response.body?.string()
            eurRate = (JSONObject(json!!).getJSONObject("bpi")
                .getJSONObject("EUR")["rate"] as String).split(".")[0]
            eurRate = eurRate.replace(",", ".")
            val euroValue: Double
            if (eurRate.isNotEmpty() && currentValue.isNotEmpty()) {
                euroValue = (eurRate.toDouble() * currentValue.toDouble())
                if(spendable) {
                    viewModel.updateEuro(euroValue)
                } else {
                    viewModel.updateUnspendableEuro(euroValue)
                }

            }
        }
    })
}
```

Ispis 41: Metoda za dohvaćanje trenutne tržišne cijene bitcoina u eurima

Programskim kodom u ispisu 41 je prikazan način na koji se dohvaća trenutna tržišna cijena bitcoina u eurima preko API sučelja prikazanog na slici 28. Prvo se pošalje upit poslužitelju koji će vratiti cijenu bitcoina u svim valutama, a nakon što se dobije odgovor od poslužitelja izvlače se potrebne informacije.

```
viewModel.euroValue.observe(viewLifecycleOwner, {
    if(currentCurrency == "0") {
        val euroValue = "%.2f".format(it)
        binding?.balanceTextView?.text = "$euroValue €"
    }
})

viewModel.euroUnspendableValue.observe(viewLifecycleOwner, {
    if(currentCurrency == "0") {
        val euroValue = "%.2f".format(it)
        binding?.unspendableBalance?.text = "$euroValue €"
    }
})
```

Ispis 42: Metoda za praćenje varijable s iznosom u eurima

Varijabla koja sadrži iznos u euru se prati te kada dođe do bilo kakve promjene automatski se ažurira iznos na početnom ekranu a to je vidljivo na programskom kodu u ispisu 42.

Slika 29: Prikaz gadgeta na početnom ekranu

Gadgeti su dizajnirani u XML-u koristeći *CardView* putem razvojnog okruženja Android Studio. Prvi gadget prikazuje posljednju transakciju s pripadajućim detaljima, a to

su status, visina bloka, i UID. Ako u novčaniku ne postoji transakcija bit će minimiziran, a dopušteno ga je maksimizirati korištenjem prečaca. U drugome gadgetu se nalaze informacije vezane uz sinkronizaciju novčanika sa zadnjim blokom u lancu blokova. Gadgeti su prikazani na slici 29.

```
viewModel.transactions.observe(viewLifecycleOwner, {
    it?.let { transactions ->
        if (it.isNotEmpty()) {
            try {
                viewModel.showLastTransaction()
                binding?.statusTV?.text = "Status: " +
transactions.first().status.name
                binding?.blockHeightTV?.text =
                    "Block height: " +
transactions.first().blockHeight.toString()
                binding?.timeStampTV?.text =
                    "Timestamp: " +
formatDate(transactions.first().timestamp)
                binding?.uidTV?.text = "UID: " + transactions.first().uid
            } catch (exception: Exception) {

            }
        } else {
            viewModel.hideLastTransaction()
        }
    }
})
```

Ispis 43: Metoda za praćenje gadgeta za posljednju transakciju

Programski kôd u ispisu 43 prikazuje način na koji se postavlja observer na varijablu koja prati transakcije iz ViewModela, a ako dođe do promjene gadget za transakcije na početnom zaslonu se automatski ažurira dobivenim informacijama.

```
viewModel.openMenu.observe(viewLifecycleOwner, {
    when (it) {
        true -> {
            binding?.sendButton?.isVisible = true
            binding?.receiveButton?.isVisible = true
            binding?.showDetail?.isVisible = true
            binding?.showLastTransaction?.isVisible = true
            binding?.openMenu?.setImageResource(R.drawable.minus)
        }
        false -> {
            binding?.sendButton?.isVisible = false
            binding?.receiveButton?.isVisible = false
            binding?.showDetail?.isVisible = false
            binding?.showLastTransaction?.isVisible = false
            binding?.openMenu?.setImageResource(R.drawable.plus)
        }
    }
})
```

Ispis 44: Metoda za prikaz prečaca na početnom ekranu

Programski kôd u ispisu 44 prikazuje način na koji se minimiziraju i maksimiziraju prečaci na početnom ekranu uz promjenu ikone glavnoga prečaca.

```
binding?.sendButton?.setOnClickListener {
    parentFragmentManager.beginTransaction()
        .replace(R.id.startingFragment, SendFragment())
        .addToBackStack(SendFragment().tag)
        .commit()
}

binding?.receiveButton?.setOnClickListener {
    parentFragmentManager.beginTransaction()
        .replace(R.id.startingFragment, ReceiveFragment())
        .addToBackStack(null)
        .commit()
}
```

Ispis 45: Metoda za transakciju fragmenta klikom na prečac

Programski kôd u ispisu 45 prikazuje način transakcije fragmenta klikom na odabrani prečac.

Postoji još jedna nevidljiva značajka početnog zaslona a to je duboki link. Skeniranjem QR koda koji sadrži tekst „bitcoin“ na početku URL adrese Android operativni sustav automatski ponudi sve aplikacije koje će moći obraditi tu informaciju.

```
<activity
    android:name=".BiometricActivity"
    android:screenOrientation="sensorPortrait">
    <intent-filter>
        <data android:scheme="bitcoin" />
        <action android:name="android.intent.action.VIEW" />

        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
    </intent-filter>
</activity>
```

Ispis 46: Postavljanje dubokoga linka za određenu aktivnost

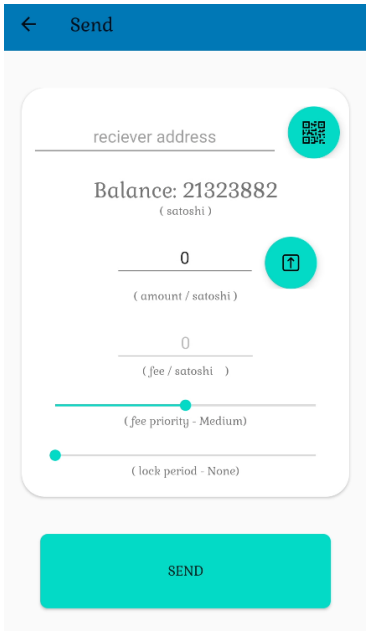
Programski kôd u ispisu 46 prikazuje način registriranja dubokoga linka za određenu aktivnost. U biometrijskoj aktivnosti novčanika se provjerava je li postoji URL te ako postoji iz njega se uzima adresa i iznos te ih se šalje kao parametre u glavnu aktivnosti.

```
private fun getExtra() {
    val extras = intent.extras
    if (extras != null) {
        addressIntent = extras.getString("address") ?: ""
        amountIntent = extras.getString("amount") ?: ""
        sendViewModel.setAddressIntent(addressIntent)
        sendViewModel.setAmountIntent(amountIntent)

        if (addressIntent.isNotEmpty()) {
            navController.navigate(R.id.sendFragment)
        }
    }
}
```

Ispis 47: Metoda za čitanje parametara za duboki link u glavnoj aktivnosti

Programski kôd u ispisu 47 prikazuje način na koji se u glavnoj aktivnosti čitaju parametri za duboki link. Prvo se provjerava je li parametri postoje, a ako da onda ih se postavlja u određena polja u fragmentu za plaćanje.

4.4. Podsustav za plaćanje



Slika 30: Prikaz fragmenta za plaćanje

Slika 30 prikazuje fragment iz kojega se vrši plaćanje. Prvo polje za unos je adresa koja ima svoj standardni format te se provjerava je li unesena adresa odgovara tome formatu prije nego što se može izvršiti plaćanje.

```
binding?.addressEditText?.setOnFocusChangeListener { _, hasFocus ->
    if (!hasFocus) {
        if
(!validateBitcoinAddress(binding?.addressEditText?.text.toString())) {
            binding?.addressEditText?.setText("")
            Toast.makeText(context, "Address is not valid.",
Toast.LENGTH_SHORT).show()
        }
    }
}
```

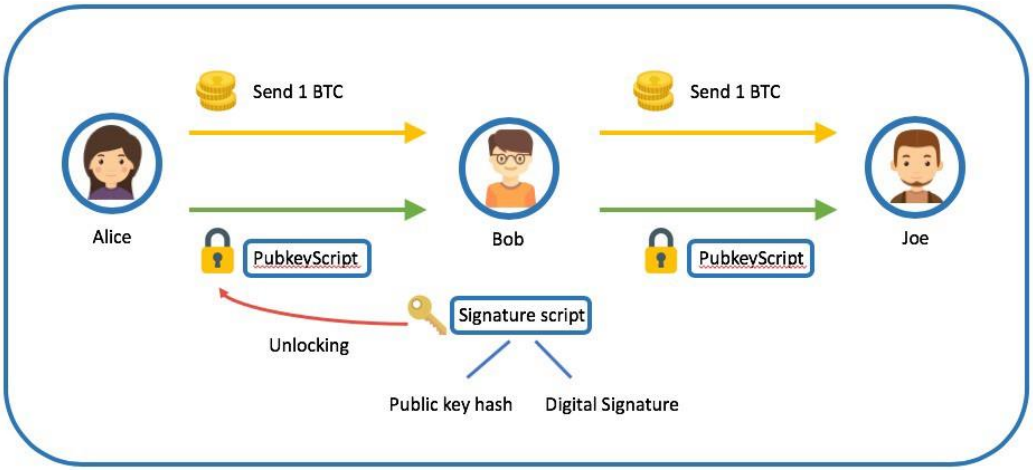
Ispis 48: Pozivanje metode za provjeru adrese

Programski kôd u ispisu 48 prikazuje način pozivanja metode za provjeru adrese nakon što polje za adresu izgubi fokus. U slučaju negativnog rezultata se ispisuje poruka korisniku te se brišu svi karakteri uneseni u to polje.

```
object BitcoinAddressValidator {
    private const val ALPHABET =
"123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz"
    fun validateBitcoinAddress(addr: String): Boolean {
        if (addr.length < 26 || addr.length > 35) return false
        val decoded = decodeBase58To25Bytes(addr) ?: return false
        val hash1 = sha256(decoded.copyOfRange(0, 21))
        val hash2 = sha256(hash1)
        return hash2.copyOfRange(0,
4).contentEquals(decoded.copyOfRange(21, 25))
    }

    private fun decodeBase58To25Bytes(input: String): ByteArray? {
        var num: BigInteger = BigInteger.ZERO
        for (t in input.toCharArray()) {
            val p = ALPHABET.indexOf(t)
            if (p == -1) return null
            num =
num.multiply(BigInteger.valueOf(58)).add(BigInteger.valueOf(p.toLong()))
        }
        val result = ByteArray(25)
        val numBytes: ByteArray = num.toByteArray()
        System.arraycopy(numBytes, 0, result, result.size - numBytes.size,
numBytes.size)
        return result
    }

    private fun sha256(data: ByteArray): ByteArray {
        return try {
            val md = MessageDigest.getInstance("SHA-256")
            md.update(data)
            md.digest()
        } catch (e: NoSuchAlgorithmException) {
            throw IllegalStateException(e)
        }
    }
}
```

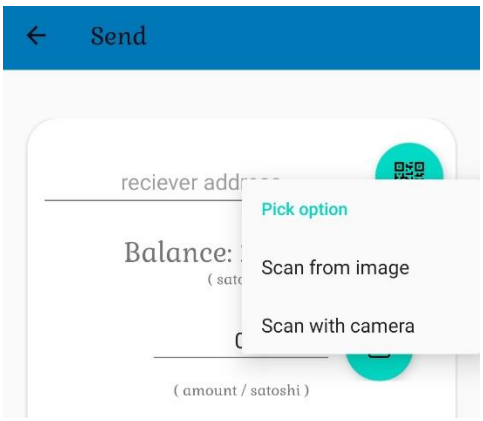
Ispis 49: Metoda za provjeru ispravnosti adrese

Programski kôd u ispisu 49 prikazuje način na koji se radi provjera ispravnosti upisane adrese. Prvo se definiraju dopušteni znakovi i izostavljaju oni koji se ne smiju koristi (nula, veliko slovo o, veliko slovo i, i malo slovo l). Nakon toga se provjerava je li adresa duga između 27 i 34 znaka. Adresa koristi base58 kodiranje, a s ovim kodiranjem adresa bi trebala imati 25 bajtova. Prvi bajt je broj verzije, sljedeći 20 bajtova su podaci, i zadnja četiri bajta su *checksum* provjera. Kako bi se provjerilo da je adresa ispravna čita se prvih 21 bajt, izračuna se *checksum*, i provjeri je li jednak kao zadnja četiri bajta adrese. Novčanik podržava samo P2PKH (engl. *Pay to Public Key Hash*) adrese za obavljanje transakcija.



Slika 31: Prikaz P2PKH način rada [13]

Slika 31 prikazuje način na koji funkcionira P2PKH. Pošiljatelj mora potpisati transakciju valjanim privatnim ključem. Skripta za izlaznu transakciju će koristiti taj potpis i javni ključ te će kroz kriptografske funkcije provjeriti valjanost transakcije. Problem P2PKH adresa za transakciju je veća i naknada od novijih formata adresa.



Slika 32: Prikaz izbornika za dobivanje adrese putem QR koda

Na slici 32 je prikazan izbornik koji sadrži opcije koje omogućuju dobivanje adrese putem skeniranja QR koda. Postoje dvije opcije a to su skeniranje QR koda učitavajući sliku i skeniranje QR koda putem kamere.

```
private fun scanQRCode() {
    val intentIntegrator = IntentIntegrator.forSupportFragment(this)
    intentIntegrator.setBeepEnabled(false)
    intentIntegrator.setCameraId(0)
    intentIntegrator.setPrompt("SCAN")
    intentIntegrator.setBarcodeImageEnabled(false)
    intentIntegrator.initiateScan()
}

private fun selectFromGallery() {
    val pickIntent = Intent(Intent.ACTION_GET_CONTENT)
    pickIntent.setDataAndType(MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
"image/*")
    startActivityForResult(Intent.createChooser(pickIntent, "Pick QR i-
mage"), 111)
}
```

Ispis 50: Metoda za integraciju opcija iz izbornika QR koda

Programski kôd u ispisu 50 prikazuje integraciju opcija za dobivanje adrese putem skeniranja QR koda. Prva metoda *scanQRCode* služi za učitavanje QR koda putem kamere na uređaju. Prvo se postavljaju određeni parametri kao što su: hoće li se čuti zvuk potvrde nakon skeniranja, naziv, ID kamere,… Nakon postavljanja svih potrebnih parametara pokreće se skeniranje te se osluškuje rezultat. Kod *selectFromGallery* metode postavlja se putanja na kojoj će se tražiti datoteke te se postavljaju dozvoljeni tipovi datoteka.

```
val result = IntentIntegrator.parseActivityResult(requestCode, resultCode,
data)
when (requestCode) {
    49374 -> {
        if (result.contents.isNotEmpty()) {
            binding?.addressEditText?.setText(result.contents)
        }
    }
    111 -> {
        if (data == null || data.data == null) {
            Log.e(
                "Error",
                "User cancelled the image selection process using the back
button."
            )
            return
        }
        readQRimage(data.data!!)
    }
}
```

Ispis 51: Metoda za osluškivanje rezultata dobivenog skeniranjem QR koda

Programski kôd u ispisu 51 prikazuje način osluškivanja rezultat za obje opcije iz izbornika. Ako se uspješno učita QR kôd putem kamere rezultat se šalje u `onActivityResult` metodu gdje se automatski postavlja adresu u polje za adresu te se vrši provjeru je li ispravna. Kod učitavanja slike metoda `onActivityResult` kao

parametar dobije putanju (URI) do datoteke koja se mora proslijediti dodatnoj metodi kako bi se pročitala adresu ako postoji.

```
private fun readQRimage(uri: Uri) {
    try {
        val inputStream: InputStream =

requireActivity().applicationContext.contentResolver.openInputStream(uri)!!
        val bitmap = BitmapFactory.decodeStream(inputStream)
        if (bitmap == null) {
            Log.e("Error", "uri is not a bitmap, $uri")
            return
        }
        val width = bitmap.width
        val height = bitmap.height
        val pixels = IntArray(width * height)
        bitmap.getPixels(pixels, 0, width, 0, 0, width, height)
        bitmap.recycle()
        val source = RGBLuminanceSource(width, height, pixels)
        val bBitmap = BinaryBitmap(HybridBinarizer(source))
        val reader = MultiFormatReader()
        try {
            val result = reader.decode(bBitmap)
            binding?.addressEditText?.setText(result.text)
        } catch (e: NotFoundException) {
            Log.e("Error", "decode exception", e)
        }
    } catch (e: FileNotFoundException) {
        Log.e("Error", "can not open file $uri", e)
    }
}
```

Ispis 52: Metoda za čitanje podataka iz datoteke s dobivene URL adrese

Programski kôd u ispisu 52 prikazuje način na koji se čita adresa iz datoteke s dobivene URI putanje. Prvo se otvora *stream* prema dobivenoj putanji, a nakon toga se datoteka dekodira te se pohranjuje u *bitmap* format te izvršava provjeru da novčanik bude siguran da se radi o slici. Nakon toga *bitmap* se prebacuje u binarni *bitmap* čitajući piksele te se nakon toga dekodira. Dobiveni rezultat se postavlja u polje za adresu gdje će se opet izvršiti provjera je li adresa ispravnoga formata.

Sljedeće polje je ukupan iznos koji se dobije tako da se kreira *observer* u *ViewModelu* koji će ažurirati polje sa svakom promjenom koja se dogodi što je prikazano programskim kodom u ispisu 54.

```
override fun onBalanceUpdate(balance: BalanceInfo) {
    this.balance.postValue(balance)
}
```

Ispis 53: Metoda za postavljanje ukupnog iznosa u ViewModelu

Programski kôd u ispisu 53 prikazuje način na koji se ažurira ukupan iznos novčanika.

```
viewModel.balance.observe(viewLifecycleOwner, { balance ->
    when (balance) {
        null -> {
            binding?.balance?.text = "Balance: 0.00 Satoshi"
        }
        else -> {
            binding?.balance?.text = "Balance: " + balance.spendable
        }
    }
})
```

Ispis 54: *Observer* za ukupan iznos novčanika

Sada se dolazi do polja gdje se upisuje iznos koji se želi poslati na određenu javnu adresu. Svaki put kada dođe do promijene iznosa za slanje, prioriteta naknade, adrese, ili zaključavanja transakcije pozvat će se automatsko ažuriranje, a ako se upiše iznos veći od ukupnog iznosa novčanika prikazat će se poruka s greškom te će se postaviti najveći mogući iznos uključujući naknadu za transakciju.

```
private fun callFeeCalculator() {
    try {
        if (binding?.sendAmount?.text.toString() != "" &&
binding?.addressEditText?.text.toString() != "") {
            //value: Long, address: String? = null, feePriority:
FeePriority
            val amount = binding?.sendAmount?.text.toString().toLong()
            val address = binding?.addressEditText?.text.toString()

            when (binding?.feePriority?.progress) {
                0 -> viewModel.fee(
                    amount,
                    address,
                    FeePriority.Low,
                    binding?.timeLockSeekBar?.progress ?: 0
                )
                1 -> viewModel.fee(
                    amount,
                    address,
                    FeePriority.Medium,
                    binding?.timeLockSeekBar?.progress ?: 0
                )
                2 -> viewModel.fee(
                    amount,
                    address,
                    FeePriority.High,
                    binding?.timeLockSeekBar?.progress ?: 0
                )
                else -> viewModel.fee(
                    amount,
                    address,
                    FeePriority.Medium,
                    binding?.timeLockSeekBar?.progress ?: 0
                )
            }
        }
        checkForMax()
    } catch (e: Exception) {
        Log.e(e.javaClass.simpleName, e.message ?: "")
        viewModel.feeLiveData.value = 0L
    }
}
```

Ispis 55: Metoda za pozivanje automatskog izračuna naknade

Programski kôd u ispisu 55 prikazuje način na koji se poziva metoda za automatski izračun naknade. Prvo se provjerava jesu li su iznos za slanje i adresa uneseni te ako jesu onda se pohranjuju u privremene varijable. Uz te varijable se provjerava je li se prioritet slanja mijenjao te je li se koristilo zaključavanje transakcije.

```
fun fee(value: Long, address: String? = null, feePriority: FeePriority,
timeLockProgress: Int) {
    timeLockSeekBar = timeLockProgress
    var tempFee = 0L
    if (value != 0L) {
        tempFee = if (timeLockSeekBar == 0) {
            bitcoinKit.fee(
                value,
                address,
                feeRate = feePriority.feeRate
            )
        } else {
            bitcoinKit.fee(
                value,
                address,
                feeRate = feePriority.feeRate,
                pluginData = getPluginData()
            )
        }
    }
    feeLiveData.value = tempFee
}
```

Ispis 56: Metoda za automatski izračun naknade

Programski kôd u ispisu 56 prikazuje način na koji se vrši automatski izračun naknade. Metoda prima potrebne parametre te poziva metodu iz kita kojega smo postavili prilikom inicijalizacije.

```
private fun checkForMax() {
    val amount = binding?.sendAmount?.text?.toString()?.toLong()
    val fee = binding?.sendFeeAmount?.text?.toString()?.toLong()
    val address = binding?.addressEditText?.text.toString()

    if ((fee!! + amount!!) > viewModel.balance.value?.spendable!!) {
        when (binding?.feePriority?.progress) {
            0 -> viewModel.maxValue(
                address,
                FeePriority.Low,
                binding?.timeLockSeekBar?.progress ?: 0
            )
```

Ispis 57: Metoda za provjera automatskog izračuna naknade

Kada se naknada dobila povratno iz metode na ispisu 56 potrebno je provjeriti je li, kada se zbroji s iznosom koji se šalje, veća od ukupnog iznosa novčanika. Ako iznos i naknada za transakciju zbrojeni prelaze ukupni iznos novčanika zove se metoda koja će vratiti i postaviti maksimalan mogući iznos uključujući naknadu za transakciju. Način na koji se to radi se može vidjeti na programskom kodu u ispisu 57.

Slika 33: Botun za postavljanje maksimalnog iznosa

Na slici 33 je prikazano polja za unos iznosa do kojega se nalazi botun koji će klikom na njega postaviti maksimalni mogući iznos za slanje uz odgovarajuću naknadu. Klikom na njega se poziva metoda koji izgleda kao metoda `checkForMax` na ispisu 57, a jedina razlika je ta da se sada ne vrši provjera unesenog iznosa zbrojenog s naknadom već se automatski izračuna i vrati najveći mogući iznos uz izračunatu naknadu.

Slika 34: Prikaz postavljanja prioriteta za transakcije i zaključavanje

Značajke prikazane na slici 34 su određivanje prioriteta slanja i zaključavanje transakcije. Transakcija koje se zaključa na odabrano vrijeme neće biti dodana u lanac blokova za vrijeme trajanja toga perioda, a prioritet će odrediti brzinu dodavanja transakcije u blok.

```
sendViewModel.feePriority.observe(viewLifecycleOwner, {
    when (it) {
        0 -> {
            binding?.seekPriorityDesc?.text = "( fee priority - Low)"
        }
        1 -> {
            binding?.seekPriorityDesc?.text = "( fee priority - Medium)"
        }
        2 -> {
            binding?.seekPriorityDesc?.text = "( fee priority - High)"
        }
    }
    callFeeCalculator()
})
```

Ispis 58: Metoda za ispisivanje prioriteta

Programski kôd u ispisu 58 prikazuje način na koji se ispisuje vrsta prioriteta u ovisnosti o tome koji je odabran, a nakon ispisa se zove metoda za izračun naknade. Prioritet je ništa drugo nego povećavanje ili smanjivanje naknade za transakciju što na kraju rezultira time da se transakcija za kraće ili duže vrijeme uvrsti u blok. Ako brzina uključivanja transakcije u blok nije bitna odabrat će se najmanji prioritet i obrnuto, ako je potrebno da transakcija bude što prije u bloku odabrat će se najveći prioritet.

Sada kada je odabran prioritet može se odabrati i vrijeme zaključavanja transakcije, drugim riječima kada se neka transakcija zaključa na određeni period primatelj neće moći koristiti valutu taj period jer neće biti dodana u lanac blokova. Postoje dvije vrste zaključavanja transakcija, prva je da se zaključa na određeno vrijeme, a druga je da se zaključa dok lanac blokova ne dosegne određeni blok. Omogućena su ova vremena za zaključavanje transakcije: sat, mjesec, šest mjeseci, godina. Dizajn zaključavanja transakcija je isti kao i prioritet slanja, a nakon ispisane poruke se zove automatsko izračunavanje naknade no sada je naknadna fiksna za sva ponuđena vremena.

```
private fun getPluginData(): MutableMap<Byte, IPluginData> {
    val pluginData = mutableMapOf<Byte, IPluginData>()
    val timeLockInterval: LockTimeInterval = when (timeLockSeekBar) {
        1 -> LockTimeInterval.hour
        2 -> LockTimeInterval.month
        3 -> LockTimeInterval.halfYear
        4 -> LockTimeInterval.year
        else -> LockTimeInterval.hour
    }
    timeLockInterval.let {
        pluginData[HodlerPlugin.id] = HodlerData(it)
    }
    return pluginData
}
```

Ispis 59: Metoda za postavljanje zaključavanja transakcije

Programski kôd u ispisu 59 prikazuje način na koji se postavljanja zaključavanje transakcije. Varijabla se puni odabranim vremenom te se vraća u određenom formatu koji se šalje putem inicijaliziranog kita prilikom plaćanja.

```
binding?.sendButton?.setOnClickListener {
    if (binding?.sendAmount?.text.toString() != "" &&
binding?.addressEditText?.text.toString() != ""
        && binding?.sendFeeAmount?.text.toString() != "") {
        //value: Long, address: String? = null, feePriority: FeePriority
        val amount = binding?.sendAmount?.text.toString().toLong()
        val address = binding?.addressEditText?.text.toString()
        var success = false
        when (binding?.feePriority?.progress) {
            0 -> success = viewModel.send(
                amount,
                address,
                FeePriority.Low,
                binding?.timeLockSeekBar?.progress ?: 0
            )
              }
        if (success) {
            binding?.addressEditText?.setText("")
            binding?.sendAmount?.setText("")
            binding?.sendFeeAmount?.setText("")
            parentFragmentManager.beginTransaction()
                .replace(R.id.startingFragment, HomeFragment())
                .addToBackStack(null)
                .commit()
            Toast.makeText(context, "Transaction sent successfully.",
Toast.LENGTH_SHORT)
                .show()
        } else {
            Toast.makeText(
                context,
                "Something went wrong with transaction.",
                Toast.LENGTH_SHORT
            )
                .show()
        }
    } else {
        if (binding?.sendAmount?.text.toString() == "") {
            Toast.makeText(context, "Amount cannot be empty.",
Toast.LENGTH_SHORT)
                .show()
        }
        if (binding?.addressEditText?.text.toString() == "") {
            Toast.makeText(context, "Address cannot be empty.",
Toast.LENGTH_SHORT)
                .show()
        }
        if (binding?.sendFeeAmount?.text.toString() == "") {
            Toast.makeText(
                context,
                "Fee cannot be zero, your amount might be too low to
send.",
                Toast.LENGTH_SHORT
            )
                .show()
        }
    }
}
```

Ispis 60: Pozivanje metode za slanje

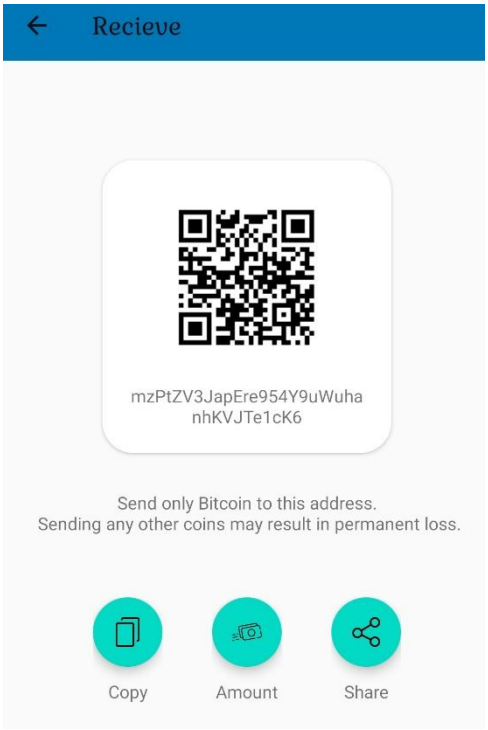
Na programskom kodu u ispisu 60 je prikazano da se prilikom slanja prvo provjere polja iznosa, adrese, i naknade koja ne smiju biti prazna. Nakon toga se dohvaćaju sve potrebne varijable te se poziva metoda za slanje iz *ViewModela*, a u slučaju greške ili ako je transakcija uspješna ispisuje se poruka na zaslon.

```
fun send(
    amount: Long,
    address: String? = null,
    feePriority: FeePriority,
    timeLockProgress: Int
): Boolean {
    timeLockSeekBar = timeLockProgress
    try {
        if (timeLockSeekBar == 0) {
            bitcoinKit.send(
                address!!,
                amount,
                feeRate = feePriority.feeRate,
                sortType = TransactionDataSortType.Shuffle
            )
        } else {
            bitcoinKit.send(
                address!!,
                amount,
                feeRate = feePriority.feeRate,
                sortType = TransactionDataSortType.Shuffle,
                pluginData = getPluginData()
            )
        }
        amountLiveData.value = 0
        feeLiveData.value = 0
        return true
    } catch (e: Exception) {
        when (e) {
            is SendValueErrors.InsufficientUnspentOutputs,
            is SendValueErrors.EmptyOutputs -> "Insufficient balance"
            is AddressFormatException -> "Could not Format Address"
            else -> e.message ?: "Failed to send transaction
(${e.javaClass.name})"
        }
        return false
    }
}
```

Ispis 61: Slanje transakcije

Programski kôd u ispisu 61 prikazuje metodu za slanje transakcija koja se nalazi u *ViewModelu*. Nakon primanja svih potrebnih parametra poziva se bitcoin kit koji se inicijalizirao prilikom pokretanja novčanika. Nakon obavljene transakcije iznos i naknada se resetiraju na nulu, a ako dođe do greška s transakcijom prikazuje se na zaslonu. Zbog korištenja *bloom* filtera transakcija je odmah vidljiva na početnom zaslonu.

4.5. Podsustav za primanje



Slika 35: Prikaz zaslona za primanje

Na slici 35 je prikazan zaslon za primanje, a na njemu se nalazi QR kôd koji skeniran daje adresu koja se nalazi ispod njega. Na zaslonu postoje 3 mogućnosti, a to su kopiranje adrese, postavljanje iznosa na QR kôd, i dijeljenje pomoću drugih aplikacija putem android operativnog sustava.

```
viewModel.receiveAddressLiveData.observe(viewLifecycleOwner, {
    address = it
    binding?.receiveAddress?.text = it

binding?.imageView?.setImageBitmap(QRCodeGenerator().generateQRCode("bitcoi
n:$it"))
})
```

Ispis 62: Metoda za dohvaćanje i postavljanje adrese za primanje

Spomenuto je da se prilikom inicijalizacije prethodno generira 100 adresa koje se dodaju u *bloom* filter. Zbog toga se zna koja adresa još nije korištena te se prikazuje na zaslonu za plaćanje, a pomoću nje se generira QR kôd. Kada se adresa iskoristi stavlja se na

listu iskorištenih te se automatski generira nova. Dohvaćanje adrese u fragmentu se radi preko *observera*, a prikazan je na programskom kodu u ispisu 62 gdje će se, ako dođe do promjene, adresa automatski ažurirati.

```
fun generateQRCode(textOnQRCode: String = "", width: Int = 200, height: Int
= 200): Bitmap {
    val multiFormatWriter = MultiFormatWriter()
    val bitMatrix = multiFormatWriter.encode(textOnQRCode,
BarcodeFormat.QR_CODE, width, height)
    val barcodeEncoder = BarcodeEncoder()
    return barcodeEncoder.createBitmap(bitMatrix)
}
```

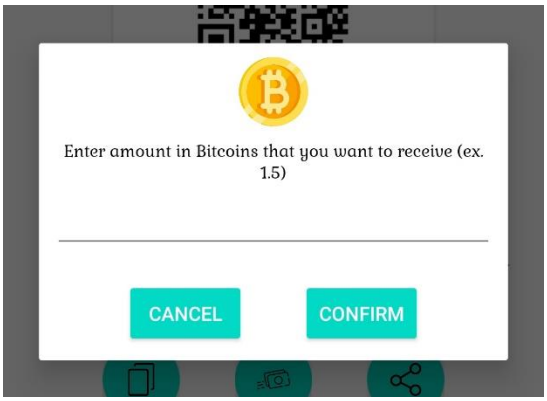
Ispis 63: Metoda za generiranje QR koda

Za generiranje QR koda kao parametre metoda prima dužinu i širinu te tekst koji će sadržavati kada se skenira što je vidljivo na programskom kodu u ispisu 63. Dužina i širina će određivati veličinu slike koja će se generirati te koja će biti prikazana na zaslonu za plaćanje. Nakon što se generira QR kod kreira se *bitmap* datoteka te vraća povratno gdje se postavlja u *ImageView*. Ispod QR koda i adrese koji se nalaze u `CardViewu` se nalazi poruka korisniku koja upozorava da generirana adresa služi samo za primanje bitcoina te da se ne može primiti niti jedna druga valuta na tu adresu.

```
private fun copyToClipboard(txt: String) {
    (requireActivity().getSystemService(CLIPBOARD_SERVICE) as
ClipboardManager).apply {
        setPrimaryClip(ClipData.newPlainText(null, txt))
    }
}
```

Ispis 64: Metoda za kopiranje adrese

U metodu za kopiranje koja je prikazana na programskom kodu u ispisu 64 se prima tekst koji se kasnije klikom na botun pohranjuje u međuspremnik (engl. *Clipboard*) dohvaćanjem i korištenjem potrebnih servisa na androidu. Nakon što se međuspremnik uspješno napuni na zaslonu se ispisuje povratna informacija.



Slika 36: Metoda za postavljanje iznosa na QR kôd

Slika 36 prikazuje dijalog koji služi za unos iznosa s kojim će se generirati adresa za plaćanje na QR kodu. Prvo se unese iznos u bitcoinu te se potvrdi, a nakon toga se automatski generira QR kôd s adresom koja sadrži uneseni iznos.

```
binding?.receiveSetAmountButton?.setOnClickListener {
    hideKeyboard()
    val mDialogView =
        LayoutInflater.from(context).inflate(R.layout.amount_dialog, null)
    val dialogConfirmBtn =
mDialogView.findViewById<Button>(R.id.dialogConfirmBtn)
    val dialogCancelBtn =
mDialogView.findViewById<Button>(R.id.dialogCancelBtn)
    val amount =
mDialogView.findViewById<EditText>(R.id.amountTextNumberDecimal)
    //AlertDialogBuilder
    val mBuilder = AlertDialog.Builder(context)
        .setView(mDialogView)
    //show dialog
    val mAlertDialog = mBuilder.show()
    //login button click of custom layout
    dialogConfirmBtn.setOnClickListener {
        //bitcoin:bitcoin_address_is_here?amount=5.00
        val enteredAmount = amount.text.toString()
        val addressBuilder = "bitcoin:$address?amount=$enteredAmount"

binding?.imageView?.setImageBitmap(QRCodeGenerator().generateQRCode(address
Builder))
        Toast.makeText(context, "Amount set to QR code.",
Toast.LENGTH_SHORT)
            .show()
        mAlertDialog.dismiss()
    }
    //cancel button click of custom layout
    dialogCancelBtn.setOnClickListener {
        mAlertDialog.dismiss()
    }
}
```

Ispis 65: Metoda za kreiranje dijaloga za unos iznosa kod generiranja QR koda

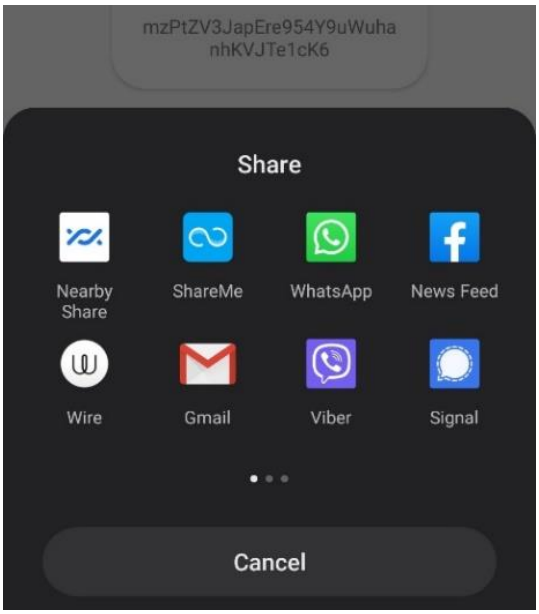
Programski kôd u ispisu 65 prikazuje način implementacije dijaloga koji služi za unosa iznosa kod generiranja QR koda. Nakon što se dijalog prikaže čeka se njegov

završetak nakon čega će metoda dobiti rezultat prema kojemu će se izvršiti sljedeća radnja, a na kraju će se prikazati poruka da je QR kôd generiran i spreman za korištenje. Način generiranja QR koda je prikazan na programskom kodu u ispisu 63. Adresa koja se prikazuje ispod QR koda se ne mijenja.

```
binding?.receiveShareButton?.setOnClickListener {
    val sendIntent: Intent = Intent().apply {
        action = Intent.ACTION_SEND
        putExtra(Intent.EXTRA_TEXT, "Bitcoin address: $address")
        type = "text/plain"
    }
    val shareIntent = Intent.createChooser(sendIntent, null)
    startActivity(shareIntent)
}
```

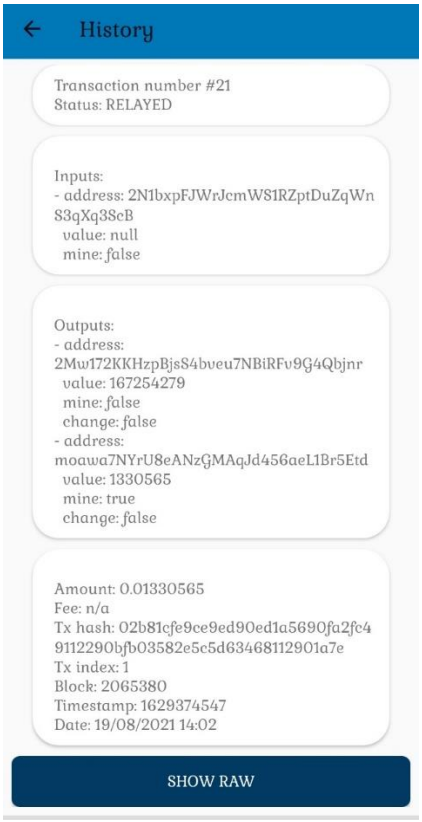
Ispis 66: Metoda za implementacija dijeljenja adrese

Zadnja opcija koja je ostala je dijeljenje adrese putem android operativnog sustava čija je implementacija prikazana na programskom kodu u ispisu 66. Postavlja se tekst za *broadcast* te se nakon toga otvori aktivnost gdje se može odabrati način na koji će se podijeliti koja je vidljiva na slici 37.



Slika 37: Prikaz prozora za dijeljenje adrese

4.6. Povijest transakcija



Slika 38: Prikaz fragmenta za povijest transakcija

Slika 38 prikazuje fragment u kojemu se vide sve transakcije vezane za novčanik. Prva sekcija prikazuje broj transakcije i status. Transakcija je odvojena na tri sekcije a to su ulaz, izlaz, te ostale informacije. U sekciji ulaznih transakcija se vide sve adrese s kojih je transakcija poslana i njihove vrijednosti. U sekciji izlaznih transakcija se vide sve adrese na koje se transakcija šalje, njihove vrijednosti, postoji li ostatak, i je li transakcija zaključana te do kada. Na kraju u zadnjoj sekciji je prikazan iznos, naknada, *hash*, blok u kojemu se transakcija nalazi, vrijeme i datum.

```
override fun onTransactionsUpdate(
    inserted: List<TransactionInfo>,
    updated: List<TransactionInfo>
) {
    bitcoinKit.transactions().subscribe { txList: List<TransactionInfo> ->
        transactions.postValue(txList)
    }.let {
        disposables.add(it)
    }
}
```

Ispis 67: Metoda za praćenje transakcija u *ViewModelu*

U *ViewModelu* se prate sve transakcije koje su vezane uz novčanik te ih se pohranjuje kao lista a taj proces je prikazan na programskom kodu u ispisu 67.

```
constructor(uid: String,
            transactionHash: String,
            transactionIndex: Int,
            inputs: List<TransactionInputInfo>,
            outputs: List<TransactionOutputInfo>,
            fee: Long?,
            blockHeight: Int?,
            timestamp: Long,
            status: TransactionStatus,
            conflictingTxHash: String? = null) {
    this.uid = uid
    this.transactionHash = transactionHash
    this.transactionIndex = transactionIndex
    this.inputs = inputs
    this.outputs = outputs
    this.fee = fee
    this.blockHeight = blockHeight
    this.timestamp = timestamp
    this.status = status
    this.conflictingTxHash = conflictingTxHash
}
```

Ispis 68: Polja koja sadrži transakcija [10]

Programski kôd u ispisu 68 prikazuje polja koja će sadržavati spremljena transakcija, a informacije iz tih polja će biti ispisane u određenoj sekciji fragmenta za povijest.

```
data class TransactionInputInfo(val mine: Boolean, val value: Long? = null,
val address: String? = null)

data class TransactionOutputInfo(val mine: Boolean,
                                 val changeOutput: Boolean,
                                 val value: Long,
                                 val address: String? = null,
                                 val pluginId: Byte? = null,
                                 val pluginData: IPluginOutputData? = null,
                                 internal val pluginDataString: String? =
null)

data class BlockInfo(
        val headerHash: String,
        val height: Int,
        val timestamp: Long
)

data class BalanceInfo(val spendable: Long, val unspendable: Long)
```

Ispis 69: Podatkovne klase za transakcije [10]

Programski kôd u ispisu 69 prikazuje podatkovne klase za ulazne transakcije, izlazne transakcije, podatke o bloku, i informacije o saldu.

```
viewModel.transactions.observe(viewLifecycleOwner, {
    it?.let { transactions ->
        transactionsAdapter.items = transactions
        transactionsAdapter.notifyDataSetChanged()
    }
})
```

Ispis 70: Postavljanje *observera* za praćenje transakcije u fragmentu

Programski kôd u ispisu 70 prikazuje način postavljanja *observera* koji će ako se dogodi neka nova transakcija napuniti listu i obavijestiti *RecyclerView*.

```
viewModel.transactionRaw.observe(viewLifecycleOwner, { transactionHex ->
    activity?.let {
        val dialog = AlertDialog.Builder(it)
            .setMessage(transactionHex)
            .setTitle("Transaction RAW")
            .setPositiveButton(
                "Copy"
            ) { _, _ ->

(requireActivity().getSystemService(Context.CLIPBOARD_SERVICE) as
ClipboardManager)
                    .apply {
                    setPrimaryClip(ClipData.newPlainText(null,
transactionHex))
                }
                Toast.makeText(context, "RAW copied to clipboard.",
Toast.LENGTH_SHORT)
                    .show()
            }
            .setNegativeButton(
                "Cancel"
            ) { dg, _ -> dg.dismiss() }
            .create()
        dialog.show()
    }
})
```

Ispis 71: Postavljanje *observera* za sirovi oblik transakcije

Programski kôd u ispisu 71 prikazuje način na koji se dohvaća sirovi oblik određene transakcije (engl. *Raw*). Nakon dohvaćanja podataka kreira se dijalog koji će prikazati sirovi oblik transakcije te će imati mogućnost kopiranja u međuspremnik. U slučaju kopiranja sirovog oblika transakcije podataka u međuspremnik na zaslonu će se prikazati obavijest o uspješnosti.

Slika 39: Pregled sirovog oblika transakcije

Na slici 39 je prikazan izgled dijaloga za prikaz sirovog oblika transakcije s pripadajućim botunima.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    transactionsRecyclerView = binding?.transactions!!
    transactionsRecyclerView.adapter = transactionsAdapter
    transactionsRecyclerView.layoutManager = LinearLayoutManager(context)
}
```

Ispis 72: Metoda za postavljanje adaptera za *RecyclerView*

Programski kôd u ispisu 72 prikazuje način na koji se postavlja adapter za `RecyclerView` iz dodatne klase koja je kreirana u istome fragmentu.

```
class TransactionsAdapter(private val listener:
ViewHolderTransaction.Listener, cntxt: Context) :
    RecyclerView.Adapter<RecyclerView.ViewHolder>() {
    var items = listOf<TransactionInfo>()
    var context = cntxt

    override fun getItemCount() = items.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
RecyclerView.ViewHolder =
        ViewHolderTransaction(
            LayoutInflater.from(parent.context)
                .inflate(R.layout.view_holder_transaction, parent, false),
listener
        )

    override fun onBindViewHolder(holder: RecyclerView.ViewHolder,
position: Int) {
        when (holder) {
            is ViewHolderTransaction -> holder.bind(items[position],
itemCount - position, context)
        }
    }
}
```

Ispis 73: Adapter za transakcije

Programski kôd u ispisu 73 prikazuje adapter za transakcije. On prima još jednu dodatnu klasu koju je kreirana u istome fragmentu, a s njom se definira dizajn i postavljaju podaci koji se prikazuju. Adapter u `onCreateViewHolder` metodi nasljeđuje kreirani dizajn, a putem `onBindViewHolder` se prikazuju podaci na specifičnoj poziciji.

```
private fun mapOutputs(list: List<TransactionOutputInfo>): String {
    return list.joinToString("") {
        val sb = StringBuilder()
        sb.append("\n- address: ${it.address}")
        sb.append("\n  value: ${it.value}")
        sb.append("\n  confirm: ${it.mine}")

        if (it.pluginId == HodlerPlugin.id && it.pluginData != null) {
            (it.pluginData as? HodlerOutputData)?.let { hodlerData ->
                val lockTimeInterval = hodlerData.lockTimeInterval

                hodlerData.approxUnlockTime?.let { lockedUntilApprox ->
                    sb.append(
                        "\n  * Locked: ${lockTimeInterval.name}, approx
until ${
                            formatDate(
                                lockedUntilApprox
                            )
                        }"
                    )
                }

                sb.append("\n  * Address: ${hodlerData.addressString}")
                sb.append("\n  * Value: ${it.value}")
            }
        }
        sb.toString()
    }
}
```

Ispis 74: Mapiranje izlaza u adapteru

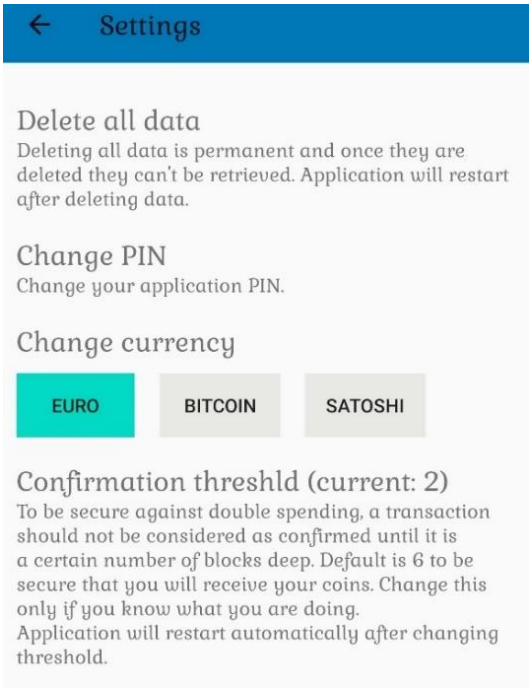
Programski kôd u ispisu 74 prikazuje kreiranje teksta za izlazne transakcije tako da se dohvaćaju sve informacije iz liste koja je dobivena kao parametar. Uz to se dohvaća i informacija o zaključavanju transakcije te se nakon dohvaćanja formatira. Na isti način se dohvaćaju i ulazne transakcije koje se nalaze u drugoj listi.

```
val txAmount = calculateAmount(transactionInfo)
    val amount = NumberFormatHelper.cryptoAmountFormat.format(txAmount /
100_000_000.0)
    val fee = transactionInfo.fee?.let {
        NumberFormatHelper.cryptoAmountFormat.format(it / 100_000_000.0)
    } ?: "n/a"

    var textNumber = "Transaction number #$index"
    textNumber += if(transactionInfo.status.name == "RELAYED"){
        "\nStatus: COMPLETE"
    }else {
        "\nStatus: ${transactionInfo.status.name}"
    }

    val lastBlock = Operations().readFromSharedPreferences(activity,
Operations.LAST_BLOCK)
    val confirmationsThreshold =
Operations().readFromSharedPreferences(activity, "confirm")
    val currentBlockHeight = transactionInfo.blockHeight
    val confirmations = (lastBlock!!.toInt() -
currentBlockHeight!!.toInt()) + 1

    if(confirmations < confirmationsThreshold!!.toInt()) {
        textNumber += "\nConfirmations:
$confirmations/$confirmationsThreshold"
    }
    if(confirmations == confirmationsThreshold.toInt()) {
        textNumber += "\nConfirmations:
$confirmationsThreshold/$confirmationsThreshold"
    }
    if(confirmations > confirmationsThreshold.toInt()) {
        textNumber += "\nConfirmations:
$confirmationsThreshold+/$confirmationsThreshold"
    }

    numberTV.text = textNumber

    val inputsText = "\nInputs: ${mapInputs(transactionInfo.inputs)}"
    inputsTV.text = inputsText

    val outputText = "\nOutputs: ${mapOutputs(transactionInfo.outputs)}"
    outputTV.text = outputText

    val infoText = "\nAmount: $amount" +
            "\nFee: $fee" +
            "\nTx hash: ${transactionInfo.transactionHash}" +
            "\nTx index: ${transactionInfo.transactionIndex}" +
            "\nBlock: ${transactionInfo.blockHeight}" +
            "\nTimestamp: ${transactionInfo.timestamp}" +
            "\nDate: ${formatDate(transactionInfo.timestamp)}"
    infoTV.text = infoText
}
```

Ispis 75: Metoda za postavljane sekcija

Programski kôd u ispisu 75 prikazuje način na koji se ispisuju dohvaćene informacije te ispisuju po sekcijama. Da bi se ispisao broj potvrda transakcije dohvatila se visina bloka transakcije i visina zadnjeg bloka koji je sinkroniziran u novčaniku. Nakon toga se izračuna broj potvrda transakcije tako da se oduzme visina bloka transakcije od visine zadnjeg bloka te se nadoda jedan.

4.7. Postavke i informacije o aplikaciji



Slika 40: Prikaz fragmenta za postavke

Slika 40 prikazuje fragment za postavke sa svim pripadajućim opcijama. Prva postavka je brisanje svih podataka, a razlog brisanja može biti odabir krivog novčanika ili zamjena trenutnoga uređaja na kojemu se nalazi novčanik. Sljedeća postavka je postavka za mijenjanje PIN koda aplikacije, pa postavka za mijenjanje valute, te na kraju postavka koja će definirati broj potvrda transakcije. Ova postavka određuje nakon koliko potvrda će se transakcija prikazati u novčaniku. Preporučeni broj potvrda je 6 jer se smatra da je nakon toga broja transakcija nepovratna.

```
binding?.deleteAllDataTV?.setOnClickListener {
    val mDialogView =

LayoutInflater.from(context).inflate(R.layout.warning_dialog_delete, null)
    val dialogConfirmBtn =
mDialogView.findViewById<Button>(R.id.dialogConfirmBtn)
    val dialogCancelBtn =
mDialogView.findViewById<Button>(R.id.dialogCancelBtn)
    //AlertDialogBuilder
    val mBuilder = AlertDialog.Builder(context)
        .setView(mDialogView)
    //show dialog
    val mAlertDialog = mBuilder.show()
    //login button click of custom layout
    dialogConfirmBtn.setOnClickListener {
        (requireContext().getSystemService(ACTIVITY_SERVICE) as
ActivityManager)
            .clearApplicationUserData()
    }
    //cancel button click of custom layout
    dialogCancelBtn.setOnClickListener {
        mAlertDialog.dismiss()
    }
}
```

Ispis 76: Metoda za brisanje podataka aplikacije

Programski kôd u ispisu 76 prikazuje način na koji se brišu podaci aplikacije s android operativnog sustava. Kreira se dijalog koji će tražiti potvrdu brisanja podataka aplikacije, a nakon potvrde svi će se podaci izbrisati bespovratno. Nakon brisanja podataka aplikacija će se resetirati te će se morati napraviti ponovna inicijalizacija ako se želi koristiti.

```
dialogConfirmBtn.setOnClickListener {
    val secretKey = Cryptography().generateSecretKey(PIN_LOC)
    val encryptedPassword = Cryptography().encryptMsg(
        binding?.newPassword?.text.toString(),
        secretKey
    )
    activity?.let { it1 ->
        Operations().saveHashMap(
            PIN_LOC, encryptedPassword,
            it1
        )
    }
    mAlertDialog.dismiss()
    Toast.makeText(
        context,
        "PIN has been changed successfully.",
        Toast.LENGTH_SHORT
    )
        .show()
    parentFragmentManager.beginTransaction()
        .replace(R.id.startingFragment, SettingsFragment())
        .addToBackStack(null)
        .commit()
}
```

Ispis 77: Metoda za promjenu PIN koda

Klikom na postavku za izmjenu PIN koda kreira se dijalog koji će sadržavati tri polja, a to su polje za unos staroga PIN koda, polje za unos novoga PIN koda, te polje za ponovni unos novoga PIN koda. Nakon potvrde, ako su novi i ponovljeni PIN identični, PIN se kriptira te ga se pohranjuje u zajedničke postavke. Proces kriptiranja PIN koda je prikazan na programskom kodu u ispisu 77, dok je izgled dijaloga za promjenu PIN koda prikazan na slici 41. Tipkovnicu za unos PIN koda je ograničena samo na brojeve. Prije potvrde su napravljene provjere kao kod kreiranja PIN koda prilikom inicijalizacije.

```
val currentCurrency =
Operations().readFromSharedPreferences(requireActivity(), "currency")
if (currentCurrency != null) {
    settingsViewModel.setCurrency(currentCurrency)
}

binding?.euroCurrency?.setOnClickListener {
    Operations().writeToSharedPreferences(requireActivity(), "currency",
"0")
    settingsViewModel.setCurrency("0")
    Toast.makeText(context, "Currency changed to Euro.",
Toast.LENGTH_SHORT).show()
}

binding?.bitcoinCurrency?.setOnClickListener {
    Operations().writeToSharedPreferences(requireActivity(), "currency",
"1")
    settingsViewModel.setCurrency("1")
    Toast.makeText(context, "Currency changed to Bitcoin.",
Toast.LENGTH_SHORT).show()
}

binding?.satoshiCurrency?.setOnClickListener {
    Operations().writeToSharedPreferences(requireActivity(), "currency",
"2")
    settingsViewModel.setCurrency("2")
    Toast.makeText(context, "Currency changed to Satoshi.",
Toast.LENGTH_SHORT).show()
}

settingsViewModel.currency.observe(viewLifecycleOwner, { currency ->
    if (currency == "0") {

binding?.euroCurrency?.setBackgroundColor(requireActivity().getColor(R.colo
r.colorAccent))
    } else {

binding?.euroCurrency?.setBackgroundColor(requireActivity().getColor(R.colo
r.platinum))
    }
    if (currency == "1") {

binding?.bitcoinCurrency?.setBackgroundColor(requireActivity().getColor(R.c
olor.colorAccent))
    } else {

binding?.bitcoinCurrency?.setBackgroundColor(requireActivity().getColor(R.c
olor.platinum))
    }
    if (currency == "2") {

binding?.satoshiCurrency?.setBackgroundColor(requireActivity().getColor(R.c
olor.colorAccent))
    } else {

binding?.satoshiCurrency?.setBackgroundColor(requireActivity().getColor(R.c
olor.platinum))
    }
})
```

Ispis 78: Metoda za mijenjanje prikaza valute

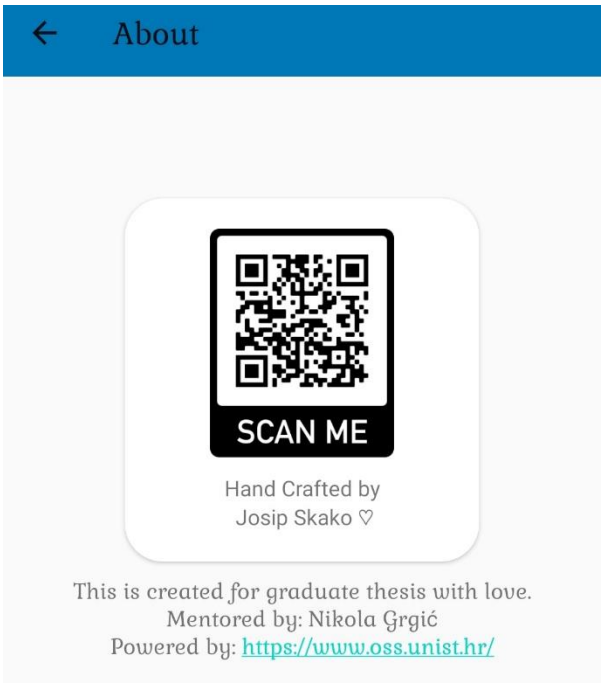
Programski kôd u ispisu 78 prikazuje način na koji se mijenja prikaz valute. Prvo se pročita trenutna valuta kako bi se u postavkama mogla označiti te tako znati koja je trenutno u upotrebi. Nakon toga se postavlja trenutna valuta u *ViewModel* na koju će se postaviti *observer*. Sada kada se valuta nakon odabira druge promjeni to se zapisuje zajedničke postavke, postavlja *ViewModel*, te se ispisuje poruka da je operacija uspješna. Zahvaljujući *observeru*, prilikom izmjene se automatski ažuriraju sva potrebna polja. Botun trenutne valute je obojan različitom bojom od botuna valuta koje se ne koriste.

```
val confirmations =
Operations().readFromSharedPreferences(requireActivity(), "confirm")
var confirmationsInt = 6
if (!confirmations.equals("0")) {
    confirmationsInt = confirmations?.toInt() ?: 6
}
binding?.confirmationsTitle?.text = "Confirmation threshld (current:
$confirmationsInt)"

binding?.confirmationsTitle?.setOnClickListener {
    hideKeyboard()
//Some code between
        desc.text = "Enter number of confirmation threshold"
    amount.inputType = InputType.TYPE_CLASS_NUMBER
    //AlertDialogBuilder
    val mBuilder = AlertDialog.Builder(context)
        .setView(mDialogView)
    //show dialog
    val mAlertDialog = mBuilder.show()

    //login button click of custom layout
    dialogConfirmBtn.setOnClickListener {
        var myAmount = amount.text.toString()
        if(myAmount.trim() == "0" || myAmount.trim().isEmpty()) {
            myAmount = "6"
        }
        Operations().writeToSharedPreferences(requireActivity(), "confirm",
myAmount)
        Toast.makeText(context, "Confirmation threshold changed to
$myAmount, application will restart now.", Toast.LENGTH_SHORT)
            .show()
        mAlertDialog.dismiss()
        val runnable = Runnable {
            Operations().doRestart(requireContext())
        }
```

Ispis 79: Metoda za promjenu broja potvrda transakcije

Programski kôd u ispisu 79 prikazuje način na koji se mijenja zadani broj potvrda transakcije a proces je sličan mijenjanju valute. Prvo se dohvaća trenutni zadani broj kako bi se vizualno prikazao u postavkama. Prilikom izmjene otvara se dijalog gdje se unosi broj

potvrda transakcije koji se nakon potvrde pohranjuje u zajedničke postavke. Ako se unese nula ili ostavi polje prazno broj potvrda će se postaviti na preporučeni, a to je šest.



Slika 42: Fragment za prikaz informacija

Za kraj je ostao fragment informacija koji je prikazan na slici 42. Na njemu se nalazi ručno generirani QR kôd koji daje više informacija o kreatoru aplikacije. Na ovom fragmentu je implementirana metoda koja će se potezom prsta od lijeva prema desno vratiti na početni zaslon.

```
fun setTouchListener(context: Context, layout: FrameLayout, view: View) {
    layout.setOnTouchListener(object : OnSwipeTouchListener(context) {
        override fun onSwipeRight() {
            super.onSwipeRight()
            view.findNavController().navigateUp()
        }
```

Ispis 80: Metoda za registriranje poteza prsta

Slika 80 prikazuje način implementacije poteza prsta za vraćanje na početni zaslon preko navigacijskog kontrolera.

5. Zaključak

U ovom završnom radu opisan je postupak izrade funkcionalnog Bitcoin novčanika za Android operativni sustav kao i korištene tehnologije. Bitcoin je jedna od najbrže rastućih mreža koja iz dana u dan broji porast korisnika. Kako bi se zaštitili, korisnici sve više tragaju za brzim, sigurnim, i pouzdanim digitalnim novčanikom koji omogućuje mobilnost i jednostavnu upotrebu.

U praktičnom dijelu rada izrađen je novčanik koji obavlja pouzdano plaćanje i primanje. Korištenjem *bloom* filtera transakcije su gotovo trenutno vidljive u novčaniku. Zahvaljujući SPV principu, kontaktiranjem punih čvorova se dohvaćaju samo one transakcije koje su vezane uz novčanik. To rezultira brzom sinkronizacijom s lancem blokova i uštedom prostora koji je dosta ograničen na mobilnim uređajima. Kako su digitalni novčanici česta meta napada zbog podataka koje čuvaju osigurana je velika razina sigurnosti tako da se prilikom svakog pokretanja vrši provjera uređaja, a podaci se kriptiraju AES algoritmom. Podaci potrebni za rad novčanika se pohranjuju u SQLite bazu podataka koja se pokazala brza, sigurna, i efikasna. Automatskim izračunom naknade i mogućnosti odabira prioriteta slanja mogućnost neuvrštavanja transakcije u blok se sveo na minimum.

U budućem razvoju novčanika bilo bi korisno uvesti podršku za plaćanje na više tipova adresa poput P2PSH i Bech32. Uvođenjem novih tipova adresa bi se smanjila cijena naknade za transakciju. Uz to bi se trebalo omogućiti korisniku spajanje na čvorove koje on odabere putem IP adrese i porta, a ne one koje mu nameće aplikacija.

Literatura

1. javaTpoint, Android Studio https://www.javatpoint.com/android-studio (posjećeno 25.08.2021)
2. InfoWorld, *What is Kotlin? The Java alternative explained* https://www.infoworld.com/article/3224868/what-is-kotlin-the-java-alternative-explained.html (posjećeno 25.08.2021)
3. PromptBytes, *Java vs. Kotlin – The No-nonsense Comparison of Android Programming Languages* https://www.promptbytes.com/blog/java-vs-kotlin-the-no-nonsense-comparison-of-android-programming-languages (posjećeno 25.08.2021)
4. O'Reilly, *Using SQLite by Jay A. Kreibich* https://www.oreilly.com/library/view/using-sqlite/9781449394592/ch01.html (posjećeno 25.08.2021)
5. kinsta, *What is Github? A Beginner's Introduction to GitHub* https://kinsta.com/knowledgebase/what-is-github/ (posjećeno 25.08.2021)
6. crobitcoin, Što je Bitcoin? https://crobitcoin.com/bitcoin/sto-je-bitcoin/ (posjećeno 25.08.2021)
7. BUG, Što je u stvari *blockchain* i kako radi? https://www.bug.hr/tehnologije/sto-je-u-stvari-blockchain-i-kako-radi-3011 (posjećeno 25.08.2021)
8. Android Developers, *Create an Android library* https://developer.android.com/studio/projects/android-library (posjećeno 26.08.2021)
9. bitcoinJ, *What is bitcoinj?* https://bitcoinj.org/ (posjećeno 26.08.2021)
10. horizontalsystems, *bitcoin-kit-android* https://github.com/horizontalsystems/bitcoin-kit-android (posjećeno 27.08.2021)
11. security.foi.hr, *MitM* napadi https://security.foi.hr/wiki/index.php/MitM_napadi.html (posjećeno 27.08.2021)

12. bitcoin, *bip-0032* https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki (posjećeno 28.08.2021)
13. Genesis Block, *What is P2PKH?* https://genesisblockhk.com/what-is-p2pkh/ (posjećeno 28.08.2021)