FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Aplikace pro správu skupinových výdajů a příjmů
Student:	David Nechuta
Vedoucí:	Ing. Jiří Daněček
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Seznamte se s jazykem Kotlin a popište jeho hlavní přednosti oproti jazyku Java.
Prostudujte a popište podporu jazyka Kotlin ve frameworku Spring a na platformě Android.
Vytvořte serverovou aplikaci a odpovídajícího mobilního klienta pro správu skupinových příjmů a výdajů. Uživatelé budou moci vytvářet virtuální peněženky (účty), do kterých bude možné zadávat skupinové výdaje a příjmy. Peněženky bude možné sdílet mezi více uživateli. Aplikace umožní sdružování plateb do kategorií podle jejich typu.
Data mobilního klienta budou automaticky synchronizována se serverem.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 4. prosince 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Aplikace pro správu skupinových výdajů a příjmů

David Nechuta

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Daněček

15. května 2019

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu práce, Ing. Jiřímu Daněčkovi za odborné vedení práce. Dále bych rád poděkoval své rodině a přátelům za podporu během celých mých studií.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 15. května 2019 .....

Odkaz na tuto práci

Nechuta, David. *Aplikace pro správu skupinových výdajů a příjmů.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato práce se zabývá popisem možností použití jazyka Kotlin při vývoji mobilních aplikací na platformě Android a serverových aplikacích používajících framework Spring.

Druhá část práce se zabývá návrhem, implementací a testováním aplikace pro správu společných výdajů. Projekt se skládá z mobilní aplikace pro platformu Android a serverové aplikace používající primárně framework Spring. Uživatelská data z mobilní aplikace jsou v reálném čase synchronizována se serverem.

Klíčová slova správa výdajů, mobilní aplikace, Kotlin, Spring Framework, REST, Android

Abstract

This thesis deals with the description of the use of Kotlin language in the development of mobile applications on Android platform and in server applications using the Spring framework.

The second part of this thesis deals with the design, implementation and testing of the application for managing common expenses. The project consists of an Android mobile application and server application using primarily Spring framework. User data from the mobile application is synchronized in real time with the server application.

Keywords expenses management, mobile application, Kotlin, Spring Framework, REST, Android

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Téměř každý se v dnešní době setkal s tím, že měl nějaké skupinové výdaje. V mé práci se zabývám možností, jak tyto výdaje ukládat na jednom místě v mobilní aplikaci, a díky tomu budou mít uživatelé o společných výdajích přehled a bude pro ně snazší pak vědět kolik dluží, případně kolik mají obdržet od ostatních uživatelů.

V teoretické části práce jsou popsány možnosti použití jazyka Kotlin při vývoji aplikací pro Android a při vývoji serverových aplikací založených na frameworku Spring. Pro operační systém Android se stal Kotlin primárně podporovaným jazykem a postupně nahrazuje dříve používaný jazyk Java. Pro framework Spring existuje podpora jazyka Kotlin. Poznatky z teoretické části jsou pak využity při vývoji aplikací v praktické části práce.

Výstupem práce je mobilní aplikace, která usnadňuje uživatelům práci s počítáním jejich společných výdajů s ostatními uživateli. Aplikace po zadání výdajů od jednotlivých uživatelů spočítá, kolik si uživatelé navzájem dluží. Součástí práce je i popis návrhu, analýzy a implementace řešení aplikace. Práce zároveň slouží jako zobrazení možností použití jazyka Kotlin.

Kapitola 1

Cíl práce

Cílem teoretické části práce je nastudování jazyka Kotlin a analýza hlavních předností a nových funkcionalit oproti jazyku Java, ze kterého jazyk Kotlin vychází. Dalším cílem je provést analýzu podpory jazyka Kotlin ve frameworku Spring a na platformě Android.

Cílem praktické části práce je vytvoření serverové aplikace a mobilního klienta pro správu skupinových příjmů a výdajů. Uživatelé si budou moci v aplikaci vytvářet virtuální peněženky (účty), ve kterých bude možné spravovat skupinové výdaje a příjmy. Peněženky bude moci mezi sebou sdílet více uživatelů a uživatelé si navzájem mohou kontrolovat aktuální stav peněženek a historii transakcí. Aplikace umožní sdružování plateb do kategorií podle jejich typu.

Serverová část aplikace je založená primárně na frameworku Spring a mobilní klient je vytvořen pro mobilní zařízení se systémem Android. Mobilní aplikace bude komunikovat se serverem a v reálném čase synchronizovat data z mobilní aplikace.

Kapitola 2

Využití jazyka Kotlin

Jazyk Kotlin je jeden z novějších programovacích jazyků (jazyk začal být zmiňován v roce 2011). Kotlin je objektově orientovaný jazyk zaměřený na JVM platformu. Důvodem ke vzniku Kotlinu byl podle autorů chybějící JVM jazyk, s dostatkem funkcí a možností kompilovat kód stejně rychle jako v Javě[2].

V této kapitole jsou popsané hlavní přednosti jazyka Kotlin oproti jazyku Java a možnost jejich uplatnění při psaní aplikací používajících framework Spring a na platformě Android.

Syntaxe Kotlinu se od Javy poměrně liší, ale Kotlin je interoperabilní s Javou. To znamená, že lze psát v rámci jednoho projektu (v různých souborech) kód v Javě i Kotlinu a v souborech s Kotlinem se odkazovat na Java soubory a naopak. Díky tomu lze projekty, které jsou napsané v Javě převádět postupně do jazyka Kotlin. V této práci není syntaxe jazyka Kotlin popsána, nicméně je volně dostupná na stránkách oficiální dokumentace jazyka[3].

2.1 Porovnání jazyka Kotlin a jazyka Java

V této sekci jsou popsány nejdůležitější rozdíly v jazycích Java a Kotlin. Je zde tedy popsáno, jaké funkcionality a možnosti poskytuje jazyk Kotlin a jazyk Java ne, a naopak, co je součástí jazyka Java a není v jazyce Kotlin.

2.1.1 Prvky jazyka Java, které neobsahuje jazyk Kotlin

2.1.1.1 Kontrolované výjimky (checked exceptions)

Součástí jazyka Java jsou dva typy výjimek – kontrolované a nekontrolované výjimky (checked a unchecked exceptions). Kontrolované výjimky jsou takové, u kterých je během kompilace zdrojového kódu kontrolováno, zda jsou zachytávány v `try-catch` bloku, nebo zda jsou výjimky deklarovány v metodě, která výjimky vyhazuje.

Kontrolované výjimky je tedy v Javě buď potřeba vložit do `try-catch` bloku, nebo do metody, která výjimku vyhazuje dopsat klíčové slovo `throws` a název výjimky, která je vyhazována.

Nekontrolované výjimky jsou takové výjimky, které nejsou během kompilace kontrolovány, zda jsou ošetřeny některou z výše zmíněných metod. V jazyce Java jsou všechny výjimky, které dědí ze třídy `RuntimeException` nebo `Error` nekontrolované[4], všechny ostatní výjimky jsou kontrolované.

V jazyce Kotlin dědí všechny třídy z `Throwable` a všechny výjimky jsou nekontrolované[5]. Není tedy třeba zachytávat výjimky v `try-catch` bloku, použití tohoto bloku je volitelné. Rozdílem oproti jazyku Java je, že v jazyce Kotlin je možnost používat `try-catch` blok jako výraz, lze z něj tedy vracet hodnotu. Příklad použití je zobrazen v následujícím bloku kódu 2.1, kde se v případě, že je `inputString` validní číslo, tak se přiřadí do proměnné `parse`, jinak je do proměnné přiřazena nula.

```
val parse: Int = try {
    Integer.parseInt(inputString)
} catch (exception: NumberFormatException) {
    0
}
```

Ukázka kódu 2.1: Ukázka použití try-catch bloku jako výrazu

2.1.1.2 Statické metody a proměnné

Jazyk Kotlin neobsahuje statické metody a proměnné. Jako náhrada slouží v jazyce Kotlinu tzv. `companion object`. Proměnné a metody umístěné v tomto objektu se pak dají používat podobně jako by se jednalo v Javě o statické metody. Příklad použití `companion object` je v následujícím bloku kódu 2.2.

```
class Foo {
    companion object {
        val x: Int = 8
    }
}
println(Foo.x) // zobrazi "8"
```

Ukázka kódu 2.2: Ukázka použití companion object

Pokud je vytvářen projekt, kde se používá jazyk Kotlin i Java zároveň, kde je potřeba používat v Java kódu proměnné ze souborů napsaných v Kotlinu jako statické, je potřeba je kvůli absenci statických proměnných a metod v jazyce Kotlin správně anotovat. Pro tyto účely jsou součástí Kotlinu anotace `@JvmField` a `@JvmStatic`. Anotace `@JvmField` udělá z proměnné, nad kterou je anotace napsaná ekvivalent veřejné statické (`public static`) proměnné v jazyce Java. U metod je potřeba použít anotaci `@JvmStatic`, která se používá nad metodami, které mají být z pohledu jazyka Java statické[6].

Takto anotované proměnné a metody lze pak používat z Java kódu stejně, jako by byly statické.

2.1.1.3 Ternární operátor

V jazyce Java existují ternární operátory pro podmíněné výrazy. Tyto operátory v Kotlinu nejsou a jsou v podstatě nahrazeny podmínkou `if`. V Kotlinu je `if` výraz vracející hodnotu, proto není ternární operátor potřeba a lze ho nahradit podmínkou. V následujícím bloku kódu 2.3 je zobrazen příklad s takovýmto použitím podmínky.

```
val num1 = 2
val num2 = 3
val min = if (num1 < num2) {
    num1
} else {
    num2
}
```

Ukázka kódu 2.3: Příklad použití podmínky jako výrazu

2.1.2 Prvky jazyka Kotlin, které neobsahuje jazyk Java

V této části jsou popsány některé důležité přednosti v jazyce Kotlin v porovnání s jazykem Java. Některé funkcionality byly popsány v předchozí části v rámci porovnání Kotlinu s Javou (`companion object`, použití podmínek a zachycování výjimek jako výrazu).

2.1.2.1 Typový systém null safety

Typový systém Kotlinu je zaměřen na eliminaci problémů spojených s hodnotami `null` v proměnných. V jazyce Java a mnohých dalších programovacích jazycích by přistupování k proměnné s hodnotou `null` vyvolalo výjimku, v jazyce Java konkrétně `NullPointerException`. Snahou jazyka Kotlin je co nejvíce tyto výjimky eliminovat[7].

V jazyce Kotlin se rozlišuje, zda může být v proměnné uložena hodnota `null` (proměnná je nullable), či nikoliv. Pokud je do proměnné možné uložit hodnotu `null`, je za datový typ proměnné při deklaraci umístěn otazník. Tedy například proměnná s datovým typem `String?` může nabývat hodnoty `null`, kdežto do proměnné s datovým typem `String` by při nastavení hodnoty `null` zobrazil kompilátor chybu.

Pokud je proměnná nullable, je třeba před tím, než je přistupováno k její hodnotě zajistit, že hodnota v proměnné není `null`. Kotlin nabízí několik možností jak toho docílit.

- Pomocí podmínek – pokud je zkontrolováno v podmínce, že proměnná neobsahuje hodnotu `null`, lze s proměnnou dále pracovat, protože kompilátor zaregistruje, že v proměnné nemůže být `null` a povolí pracovat s proměnnou.

```
val a: Int? = -52

if (a != null) {
 println(a.absoluteValue) // je jistota, ze promenna neni null
}
```

Ukázka kódu 2.4: Příklad kontroly null pomocí podmínky

Bez takovéto podmínky by se kód nezkompiloval a kompilátor by zobrazil chybu.

- Safe Calls – další možností jak zkontrolovat nulovatelnou proměnnou je pomocí tzv. safe call operátoru `?.`. Pokud je použita proměnná `a` z ukázky kódu 2.4. Tak pokud je zavoláno `println(a?.absoluteValue)`, tak to znamená, že pokud nemá proměnná `a` hodnotu `null`, je vrácena hodnota `a.absoluteValue` a pokud je hodnota `a` `null`, tak je vráceno `null`.

  Safe call operátor má uplatnění při delších řetězcích. Například při zavolání `foo?.bar?.test`, tak pokud by `foo` nebo `bar` mělo hodnotu `null`, tak pak celý takový řetězec vrací `null`.

- Elvis operátor – elvis operátor slouží jako náhrada podmínky, která by v případě, že je hodnota nějaké proměnné `null`, dosadila do proměnné nenulovou hodnotu nacházející se na pravé straně operátoru. Elvis operátor je psán jako `?:`, kdy na pravé straně operátoru je nenulová hodnota, která by se dosadila do proměnné a na levé straně nulovatelná reference.

```
val a: Int? = -52
val b: Int = a ?: 0
println(b.absoluteValue) // zobrazi 52
```

Ukázka kódu 2.5: Příklad elvis operátoru

V tomto případě, pokud by proměnná `a` byla `null`, tak se do proměnné `b` dosadí hodnota `0`, jinak se dosadí hodnota `a`.

- Operátor `!!` – operátor `!!` konvertuje libovolnou proměnnou na nenulovou a pokud daná proměnná obsahuje `null`, tak vyhodí výjimku. Dalo by se tedy říct, že při použití operátoru `!!` se bude jazyk Kotlin chovat stejně jako jazyk Java, tedy jakoby se vůbec nepoužívala kontrola null safety.

```
val a: Int? = -52
println(a!!.absoluteValue) // zobrazi 52
```

Ukázka kódu 2.6: Příklad !! operátoru

Pokud by v předchozí ukázce kódu byla v proměnné `a` `null`, byla by vyhozena výjimka.

2.1.2.2 Properties

V jazyce Kotlin existují tzv. properties. Jedná se o proměnné, které jsou umístěné ve třídách, ale ne v metodách a funkcích. Properties se označují klíčovém slovem `var`, pokud se hodnota v proměnné může měnit. Pokud se hodnota nemůže měnit, označuje se klíčovým slovem `val` (tedy jako by proměnná byla označena klíčovým slovem `final` v jazyce Java).

Ke každé property je automaticky vygenerován tzv. getter a setter, tedy funkce nastavující a vracející hodnotu proměnné. Pokud je potřeba tyto funkce upravit, například když se hodnota proměnné vypočítává na základě nějaké další proměnné, lze tyto funkce přepsat a nastavit vlastní getter nebo setter. Ukázka takovéhoto nastavení je v následujícím bloku kódu.

```
class TestList {
    private val size : Int = 0

    var isEmpty: Boolean = false
        private set // nelze nastavit hodnotu, setter je skryty
        get () = size == 0 // getter vraci hodnotu podle promenne size
    // ...
}
```

Ukázka kódu 2.7: Příklad Kotlin properties

2.1.2.3 Extension funkce a extension properties

V Kotlinu je umožněno rozšiřovat existující třídy novými funkcemi, bez toho, aby bylo nutné z této třídy dědit, nebo jiným způsobem do třídy přímo zasahovat. Je možné toho dosáhnout pomocí rozšiřujících funkcí a properties[8]. Takovéto funkce lze použít například pro přidání nových funkcí k některým třídám z knihoven, které jsou použity v projektu a nelze měnit jejich zdrojový kód. Přidání takovýchto funkcí pak může zpřehlednit kód.

Na následujícím příkladu je zobrazena extension funkce. Funkce rozšiřuje třídu `String` o funkci `isInt()`, která určuje, zda lze danou proměnnou typu `String` konvertovat na typ `Int`.

```
fun String.isInt(): Boolean = try {
    Integer.parseInt(this) //this je String, nad kterym je volana funkce
    true
} catch (e: NumberFormatException) {
    false
}

"52".isInt()  // true
"abc".isInt() // false
```

Ukázka kódu 2.8: Příklad použití extension funkcí

2.1.2.4 Smart cast

V jazyce Kotlin existuje tzv. chytré typování (nazývané Smart cast). V případech, kdy je v kódu zkontrolováno, zda je proměnná určitého typu, není pak potřeba

danou proměnnou explicitně přetypovávat, ale proměnná je automaticky přetypována kompilátorem [9]. Použití Smart Cast v kódu pak zlepší čitelnost kódu, kdy není třeba každou proměnnou přetypovávat.

```
val x: Any
val y: Any
// ...
if (x is Int) {
 println(x.absoluteValue) // neni treba pretypovat x na Int
}

if (y !is Int) return // pokud y neni Int, je zavolan return
println(y.absoluteValue) // y je automaticky pretypovano na Int
```

Ukázka kódu 2.9: Příklad použití Smart Cast

Smart Cast nemusí fungovat automaticky na všech proměnných, lze jej použít pouze pokud je kompilátor schopen zjistit, že se nemůže změnit proměnná mezi kontrolou typu v podmínce a mezi použitím proměnné. Pokud by se proměnná mohla měnit, znamenalo by to, že se může měnit i typ proměnné.

2.1.2.5 Odvození typu proměnné

V Kotlinu během psaní proměnných není vždy nutné deklarovat typ. Pokud je kompilátor schopen zjistit o jaký datový typ se jedná z hodnoty, která je do proměnné přiřazovaná, tak není vyžadováno, aby se v kódu k proměnné explicitně deklaroval datový typ. V případě, že kompilátor není schopen rozpoznat o jaký typ proměnné se jedná, je zobrazena během kompilace chyba.

2.1.2.6 Přetěžování operátorů

Jazyk Kotlin umožňuje přetěžování některých operátorů. Lze přetěžovat operátory jak pro unární, tak i pro binární operace. Přetěžování operátorů lze zavést pomocí funkcí v třídě, která má daný operátor přetěžovat, nebo pomocí rozšiřujících funkcí popsaných v předchozí části 2.1.2.3.

```
class Counter(val cnt: Int = 0) {
        // pretizeni operatoru +
        operator fun plus(num: Int): Counter {
            return Counter(cnt + num)
        }

        // pretizeni operatoru -
        operator fun minus(num: Int): Counter {
            return Counter(cnt - num)
        }
    }
```

Ukázka kódu 2.10: Příklad přetěžování operátorů

2.1.2.7 Datové třídy

V jazyce Kotlin byly přidány datové třídy tzv. `data class`. Ty mají primární úkol mít v sobě uložená nějaká data. V podstatě jsou zjednodušením POJO tříd. Datová třída má v Kotlinu oproti POJO poměrně krátkou deklaraci. Ukázka datové třídy je v následující ukázce kódu.

```
data class Dog(val name: String, val color: String, val age: Int)
```

Ukázka kódu 2.11: Ukázka data class

K vytvoření datové třídy tedy stačí deklarovat atributy třídy v konstruktoru. Kompilátor pak automaticky doplní metody `equals()`, `hashCode()`, `toString()` a `copy()`. V případě, že je potřeba měnit výchozí nastavení některé z těchto metod (například měnit výpis `toString()`), lze jednoduše tuto metodu přetížit a dopsat vlastní kód metody.

Datové třídy tedy v porovnání s POJO třídami snižují množství boilerplate kódu a zvyšují tak přehlednost kódu.

2.1.2.8 Destrukturalizace objektů

V Kotlinu je umožněna tzv. destrukturalizace objektů. To znamená, že lze rozdělit jeden objekt do několika proměnných v rámci jednoho příkazu. S těmito proměnnými pak lze dále pracovat. Ukázka destrukturalizace je v následující ukázce kódu

```
data class Animal(val name: String, val age: Int)
// ...
val animal = Animal("Bob", 2)
val (name, age) = animal // destrukturalizace na "name" a "age"
// ...
```

Ukázka kódu 2.12: Příklad destrukturalizace objektů

Aby mohla nějaká třída být destrukturalizována, musí implementovat funkce `componentN()`, kde `N` je pořadí proměnné v destrukturalizovaných proměnných. U datových tříd jsou tyto funkce implementovány automaticky a pořadí proměnných je dáno pořadím parametrů v konstruktoru.

2.2 Podpora jazyka Kotlin ve frameworku Spring

Od verze frameworku Spring 5.0 je zavedena do frameworku Spring podpora jazyka Kotlin[10]. Je tedy možné psát téměř plnohodnotné aplikace v jazyce Kotlin stejně jako v jazyce Java. V této části je popsána podpora jazyka Kotlin ve frameworku Spring a některé případy využití funkcionalit jazyka Kotlin.

Popis frameworku Spring a vysvětlení použití jednotlivých částí je pak v dalších kapitolách zabývajících se popisem použitých technologií v projektu. Tato část se zabývá pouze možnostmi použití jazyka Kotlin společně s frameworkem Spring.

Struktura projektu napsaného v jazyce Kotlin se v podstatě neliší od projektu napsaného v jazyce Java. Převod projektu napsaného v jazyce Java do jazyka Kotlin tak není příliš komplikovaný.

2.2.1 Použití datových tříd

Při návrhu entit v databázovém modelu není třeba využívat tzv. POJO tříd. Na návrh entit se v Kotlinu dají použít datové třídy. Vysvětlení co jsou entity a jak jsou pak použity v projektu je vysvětleno v subsekci s realizací aplikace 4.1.1.

Při použití datových tříd není třeba generovat tzv. gettery a settery, tedy metody mající funkci získání a nastavení hodnoty proměnné, kompilátor vygeneruje tyto metody automaticky. Dojde tedy k redukci boiler-plate kódu. Anotace pro určení jednotlivých atributů entity zůstávají stejné, akorát jsou umístěny nad proměnné, které jsou umístěny v konstruktoru. V následující ukázce kódu je zobrazena jednoduchá entita se třemi atributy `id`, `username` a `wallets`.

```
@Entity
@Table(name = "app_user")
data class User (
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "id")
        var id: Long? = null,

        @Column(name = "username")
        var username: String? = null,

        @ManyToMany(mappedBy = "wallets")
        var wallets: MutableList<Wallet>? = null,
)
```

Ukázka kódu 2.13: Příklad entity

2.2.2 Použití nullable typů

Spring framework zavedl v některých případech použití informaci o tom, zda je do proměnných možné ukládat hodnotu `null` k doplnění informací za vývojáře.

První možností je použití v kontrolerech, kde je pro získání parametru z HTTP požadavku použita anotace `@RequestParam`. Tato anotace může mít parametr `required`, který určuje, zda daný parametr musí být součástí HTTP požadavku. Tento parametr nemusí vývojář nastavovat, protože se parametr nastaví automaticky podle toho, zda je datový typ proměnné nullable. Pokud je tedy typ nastaven jako nullable, není parametr vyžadován, v opačném případě je.

Podobně je této informace využito u injektáže Spring bean. Pokud je tedy použita anotace `@Autowired`, nebo `@Inject` a datový typ proměnné není nullable, značí to, že musí daná Spring bean být nastavena. Pokud je datový typ

nullable, nemusí v aplikačním kontextu Spring bean existovat a v případě, že neexistuje, nebude vyvolána žádná výjimka a do proměnné se nastaví hodnota `null`.

2.2.3 Registrace bean

Ve frameworku Spring jsou tří možnosti registrace Spring bean. Registrace může proběhnout pomocí XML konfiguračního souboru, pomocí konfiguračních anotací, nebo přímo v kódu.

Pokud je zvolena metoda registrace přímo v kódu, je možnost použít pro registraci doménově specifický jazyk (DSL). Registrace pak probíhá v lambda výrazu. Použití takovéto metody registrace oproti jazyku Java zkracuje a zpřehledňuje kód. V následující ukázce jsou ukázky ekvivalentní registrace v jazyce Java a registrace v jazyce Kotlin.

```
GenericApplicationContext context = new GenericApplicationContext();
context.registerBean(BalanceRepository.class);
context.registerBean(BalanceService.class, () -> new
    BalanceService(context.getBean(BalanceRepository.class))
);
```

Ukázka kódu 2.14: Příklad registrace bean v jazyce Java

```
beans {
    bean<BalanceRepository>()
    bean { BalanceServiceImpl(ref()) }
}
```

Ukázka kódu 2.15: Příklad registrace bean v jazyce Kotlin

V ukázce kódu v jazyce Kotlin je použita funkce `ref()` tato funkce je součástí Spring knihovny pro jazyk Kotlin. Tato funkce vrací referenci na Spring bean. V případě, že kompilátor je schopen zjistit, jaký typ bean se má do funkce vracet, je možné použít funkci bez parametrů, jinak je potřeba do funkce doplnit i typ, případně jméno Spring bean, která se má použít.

Funkce `ref()` i ostatní funkce použité v ukázce jsou součástí třídy `BeanDefinitionDsl`, která obsahuje další pomocné metody pro práci v jazyce Kotlin při vytváření a registraci Spring bean.

2.2.4 Finální třídy

V jazyce Kotlin jsou automaticky všechny třídy nastavené jako finální, což ve frameworku Spring u některých typů tříd nelze. Před zavedením podpory jazyka Kotlin bylo potřeba všechny třídy označovat klíčovým slovem `open`, tedy označit, že daná třída není finální, aby s nimi mohl dále pracovat (pomocí proxy).

Po vydání oficiální podpory jazyka Kotlin jsou automaticky všechny třídy, které jsou anotované některou z anotací `@Component`, `@Async`, `@Transactional`, nebo `@Cacheable` jsou automaticky nastaveny pluginem `kotlin-spring` během

kompilace jako `open`. Toto platí i pro meta-anotace, které obsahují některou ze zmíněných anotací (například `@Repository`). Meta-anotace jsou anotace, které jsou anotovány dalšími anotacemi.

2.2.5 Testování s JUnit 5

V této části jsou popsány některé možnosti testování aplikace s pomocí knihovny JUnit 5. Testování probíhá téměř totožně jako při testování Java aplikací.

2.2.5.1 Jména funkcí

Výhodou je, že do jména funkce se v jazyce Kotlin dá do zpětných uvozovek vložit libovolný řetězec. Funkce, které provádí testování se tak dají lépe popsat a výpis testů je lépe čitelný, než kdyby se používala některá ze standardních notací.

```
@Test
fun `test create balance with new user`() {
    // ...
    val foundBalance =
        balanceService.findBalance(wallet.id!!, testUser!!.id!!)
    assertTrue(foundBalance.isPresent)
    // ...
}
```

Ukázka kódu 2.16: Příklad testovací funkce

2.2.5.2 Injektáž parametrů konstruktoru

V testech napsaných s pomocí JUnit 5 (v předchozích verzích nelze tuto funkcionalitu použít) lze provádět injektáž parametrů konstruktoru. V předchozích verzích se musel používat delší zápis v těle třídy. Oba dva způsoby zápisu jsou v následující ukázce kódu.

```
@SpringBootTest
@Transactional
class BalanceOperationServiceTest(
        @Autowired
        val balanceOperationService: BalanceOperationService
) {
    // ...

    // starsi zpusob zapisu
    @Autowired
    private lateinit var balanceOperationService: BalanceOperationService
}
```

Ukázka kódu 2.17: Příklad injektáže parametrů

2.3 Možnosti použití jazyka Kotlin při vývoji na platformu Android

V této části je popsáno, jak je podporován jazyk Kotlin při vývoji na platformu Android. Vzhledem k tomu, že jazyk Kotlin je nyní oficiálně podporovaným jazykem pro vývoj Android aplikací, byla vydána i knihovna Android KTX rozšiřující Android knihovnu o funkcionality, které zjednodušují a zpřehledňují kód. Některé z těchto funkcionalit jsou popsány v následujících sekcích.

Před jazykem Kotlin byl hlavním vývojovým jazykem pro vývoj aplikací Java. Proto jsou součástí oficiálního IDE Android Studio i nástroje pro převod Java kódu na Kotlin kód. Tím, že je Kotlin interoperabilní s jazykem Java, může zároveň docházet k postupnému převodu starších aplikací na jazyk Kotlin a není třeba konvertovat celý projekt najednou.

2.3.1 Vyhledání view

Uživatelské rozhraní se na platformě Android skládá z prvků dědících ze třídy `View`. Například TextView je view zobrazující text, ImageView zobrazuje obrázky. Uspořádání a konfigurace těchto prvků zobrazených na jednotlivých obrazovkách probíhá v XML souborech. Z těchto souborů se pak uživatelské rozhraní načítá do tříd, kde jsou dále použity (především do aktivit a fragmentů).

Pokud je potřeba používat některou z komponent definovaných v XML souboru ve fragmentu nebo aktivitě aplikace, tak k tomu je nejčastěji používaná funkce `findViewById()`, kde se daná komponenta vyhledávala podle `id` definovaného v XML souboru s rozložením.

Nyní tato funkce není potřeba, ale je možnost do souboru přímo importovat soubor s XML souborem. V kódu fragmentu nebo aktivity je pak možné používat dané `View` přímo jako proměnnou.

```
<!-- ... -->
<TextView
   android:id="exampleText"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
/>
<!-- ... -->
```

Ukázka kódu 2.18: Příklad XML souboru s rozložením

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.example_activity)

        // nastaveni textu, neni treba inicializovat exampleText
        exampleText.text = "Setted text"

        // zpusob zapisu bez pouziti knihovny
        val text = findViewById(R.id.exampleText)
}
```

Ukázka kódu 2.19: Příklad XML souboru s rozložením

Při používání takovýchto proměnných je stylistická nevýhoda, že v XML souborech je potřeba používat tzv. velbloudí notaci, aby v kódu byla správná notace. Což vede k tomu, že id `View` jsou napsaná velbloudí notací (camel case) a ostatní parametry používají tzv. snake case.

2.3.2 Android KTX

Android KTX je knihovna pro jazyk Kotlin. Jedná se o oficiální Android knihovnu, která je vydávána jako součást softwarových komponent označovaných jako Android Jetpack. Android Jetpack se skládá z dalších knihoven usnadňující práci při vývoji Android aplikací.

Android KTX obsahuje rozšíření k Android API a obsahuje rozšiřující funkce, které využívají nových funkcionalit jazyka Kotlin a pomocí nich je možné zpřehlednit a zlepšit čitelnost kódu.

V následujících částech je popsáno jak jsou použity funkcionality jazyka Kotlin v této knihovně. Vzhledem k tomu, že knihovna je velmi obsáhlá, jsou popsány jen některé části knihovny. Ostatní rozšiřující funkce jsou pak popsány v dokumentaci ke knihovně [11].

2.3.2.1 Lambda výrazy

Mnoho částí kódu se dá zjednodušit díky rozšířením používající lambda výrazy. Ukázkou může být nastavení uživatelských nastavení pomocí `SharedPreferences`. V první ukázce je zobrazena původní možnost zapsání nastavení a v druhé ukázce je zobrazen kód využívající Android KTX.

```
sharedPreferences.edit()
    .putString("name", name)
    .apply()
```

Ukázka kódu 2.20: Ukládání do SharedPreferences

```
sharedPreferences.edit {
    putString("name", name)
}
```

Ukázka kódu 2.21: Ukládání do SharedPreferences s Android KTX

Lambda výrazy se pak dají použít i při dalších operacích. Například při transakcích s fragmenty, nebo při databázových transakcích.

2.3.2.2 Destrukturalizace objektů

Další funkcí, která je v knihovně využívána je destrukturalizace objektů. Příkladem může být rozložení instance `Color`, tedy objektu obsahující informace o barvě, na proměnné obsahující informace o jednotlivých složkách barvy (tedy obsah červené, modré a zelené). Dalšími příklady jsou destrukturalizace tříd obsahujících časové informace, na proměnné s časovými údaji náležící danému objektu (například sekundy, minuty, hodiny).

```
val (r, g, b) = color // instance tridy Color
```

Ukázka kódu 2.22: Destrukturalizace instance tridy Color

2.3.2.3 Přetěžování operátorů

V knihovně je většinou použito přetěžování operátorů na operace s geometrickými objekty. Je tedy možné například použít matematické operace sčítání a odčítání nad instancemi tříd `Point`, `Rect`, tedy objekty reprezentujícími bod a obdélník. Například operace odčítání dvou objektů typu `Rect` vrátí objekt s oblastí určující rozdíl těchto obdélníků.

Existuje mnoho dalších možností, kde je využito přetěžování operátorů, jako například sčítání dvou barev vrátí barvu, která vznikne složením těchto barev dohromady.

Kapitola 3

Analýza a návrh

V této části jsou popsány postupy použité při analýze a návrhu aplikace. Cílem analýzy je specifikace požadavků, návrh postupu při vývoji a návrh používaných technologií. Součástí analýzy jsou i diagramy znázorňující některé struktury a postupy v aplikaci.

3.1 Popis aplikace

Cílem této práce je vytvořit mobilní aplikaci, ve které si uživatelé budou moci ukládat své společné výdaje.

Pro přístup do aplikace se uživatelé budou muset registrovat a přihlásit. Po úspěšném přihlášení do aplikace, budou uživatelům zobrazeny jejich peněženky.

Peněženka slouží v aplikaci k ukládání společných výdajů uživatelů. Do peněženky je možné přidávat další registrované uživatele. Uživatelé, kteří jsou členové peněženky, pak mohou do peněženky přidávat své transakce, tedy kolik obdrželi případně zaplatili za ostatní uživatele. Uživatelé budou moci zároveň sledovat všechny ostatní výdaje v peněžence.

V peněženkách pak mají uživatelé navzájem přehled o tom, kolik dluží ostatním uživatelům a podle toho pak mohou zaplatit ostatním uživatelům a vyrovnat účty.

3.1.1 Serverová a mobilní aplikace

Aplikace bude rozdělena na serverovou a mobilní část. Mobilní aplikace slouží jako klientská aplikace, ke které mají přístup uživatelé. Data z klientské aplikace jsou pak posílána na server, kde je veškerá logika aplikace a jsou zde ukládána data od klientů.

Protože jsou data od uživatele synchronizovaná se serverem, může se uživatel přihlásit na dalším libovolném mobilním zařízení a budou mu vždy zobrazena jeho data, která jsou uložena na serveru.

3.2 Analýza požadavků

V této kapitole jsou popsány jednotlivé požadavky na serverovou a mobilní aplikaci. Na základě těchto požadavků byla navržena architektura aplikací a použití technologií. Požadavky jsou rozděleny do dvou kategorií, funkční a nefunkční.

3.2.1 Aplikace pro zpracování dat

Funkční požadavky

F 1 Autentizace

- **F 1.1** Nepřihlášený uživatel bude mít možnost se přihlásit
- **F 1.2** Nepřihlášený uživatel bude mít možnost se registrovat
- **F 1.3** Přihlášený uživatel bude mít možnost se odhlásit

F 2 Operace s peněženkami

- **F 2.1** Přihlášený uživatel bude mít možnost vytvářet peněženky
- **F 2.2** Přihlášený uživatel bude mít možnost zobrazovat vlastní peněženky
- **F 2.3** Přihlášený uživatel bude mít možnost mazat vlastní peněženky
- **F 2.4** Přihlášený uživatel vlastnící peněženku bude mít možnost přidávat a odebírat z dané peněženky další uživatele

F 3 Operace s transakcemi

- **F 3.1** Přihlášený uživatel bude mít možnost přidávat transakce do peněženek, ve kterých je členem
- **F 3.2** Přihlášený uživatel bude mít možnost mazat transakce z peněženek, ve kterých je členem
- **F 3.3** Přihlášený uživatel bude mít možnost zobrazovat detaily transakcí z peněženek, ve kterých je členem
- **F 4.3** Po přidání transakce se budou automaticky počítat transakce nutné k vyrovnání stavů kont v peněžence

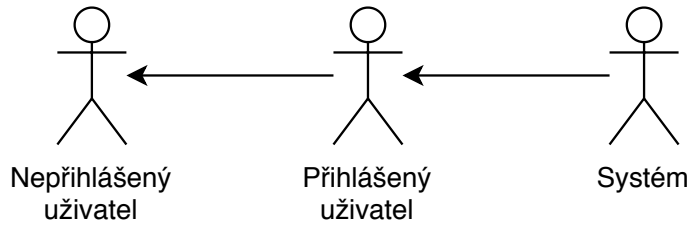
F 4 Přihlášený uživatel si bude moci zobrazit transakce nutné k vyrovnání stavů kont v peněžence

Nefunkční požadavky

- Serverová aplikace bude pracovat s daty v téměř reálném čase
- Mobilní i serverová aplikace bude napsaná v jazyce Kotlin
- Serverová aplikace bude primárně používat framework Spring
- Mobilní aplikace bude určená pro platformu Android

3.3 Aktéři

V systému jsou tři druhy aktérů. Prvním druhem aktéra je systém reprezentující aplikaci, druhým druhem aktéra je přihlášený uživatel a třetím druhem aktéra je nepřihlášený uživatel. Nepřihlášený uživatel má jen dvě dostupné operace – registrace a přihlášení. Na ostatní akce je potřeba být přihlášen.



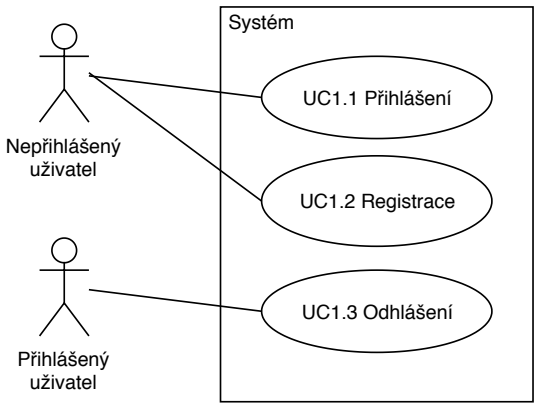
Obrázek 3.1: Aktéři

3.4 Případy užití

V této sekci jsou popsané jednotlivé případy užití. Případ užití zobrazuje vztah mezi aktéry a funkčními požadavky. Graficky jsou případy znázorněny pomocí diagramu případu užití.

3.4.1 Autentizace

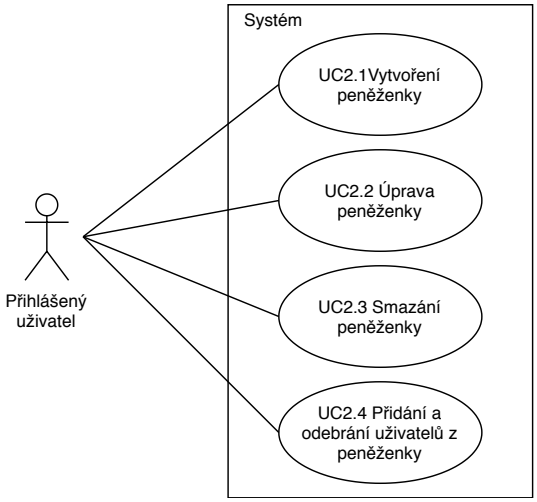
Tato část pokrývá funkční požadavky F1. Nepřihlášený uživatel se může registrovat nebo přihlásit, čímž se stane přihlášeným uživatelem. Přihlášený uživatel se může odhlásit, čímž se z něj stane nepřihlášený uživatel.



Obrázek 3.2: Use Case diagram Autentizace

3.4.2 Operace s peněženkami

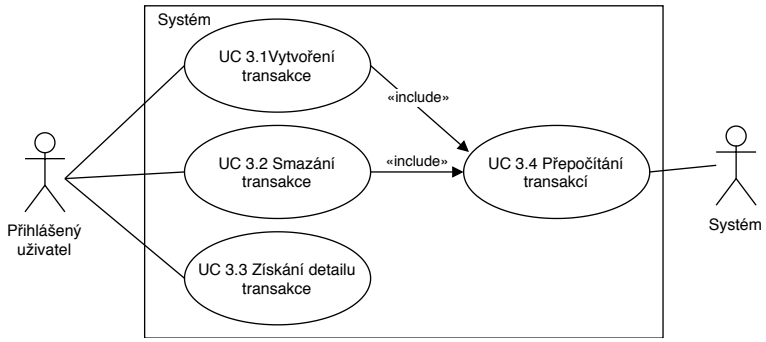
Tato část pokrývá funkční požadavky F2. Přihlášený uživatel může vytvářet, upravovat a mazat peněženky. Dále může přidávat a odebírat další uživatele jako členy peněženky, ve kterých je členem.



Obrázek 3.3: Use Case diagram Operace s peněženkami

3.4.3 Operace s transakcemi

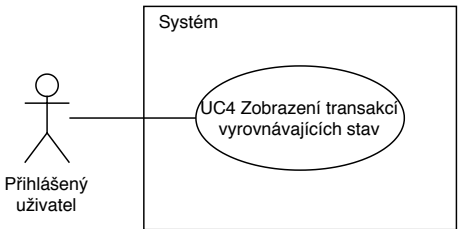
Tato část pokrývá funkční požadavky F3. Přihlášený uživatel může v peněženkách, které vlastní, nebo ve kterých je členem, přidávat, odebírat a zobrazovat detaily transakcí. Po přidání nebo odebrání transakce jsou automaticky přepočítány transakce nutné k vyrovnání stavů kont mezi uživateli.



Obrázek 3.4: Use Case diagram Operace s transakcemi

3.4.4 Vyrovnání stavů v peněženkách

Tato část pokrývá funkční požadavky F4. Přihlášený uživatel si bude moci zobrazit transakce nutné k vyrovnání stavů kont v peněžence.



Obrázek 3.5: Use Case diagram Vyrovnání stavů v peněženkách

3.5 Diagram aktivit vybraných procesů

Ke znázornění některých složitějších procesů jsou pro lepší pochopení přiloženy diagramy aktivit. U jednotlivých procesů jsou sepsány i kroky, které popisují průchod diagramem.

3.5.1 Vytvoření transakce

V této části jsou popsány jednotlivé kroky pro vytvoření transakce v jedné peněžence.

1. Přihlášený uživatel si vybere peněženku, do které chce přidat novou transakci.

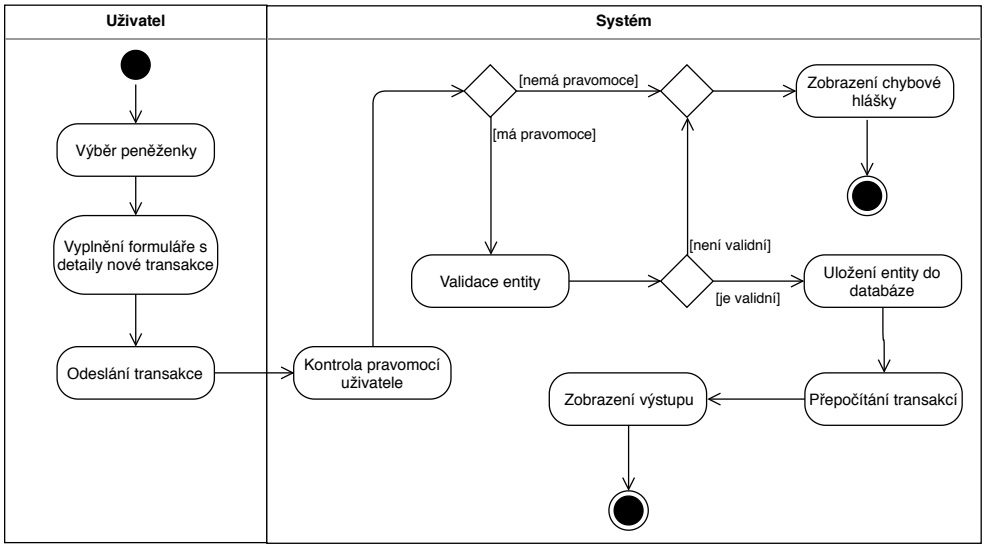
2. Přihlášený uživatel vyplní formulář pro vytvoření nové transakce a odešle transakci do systému.

3. Systém zkontroluje, zda má uživatel dostatečná oprávnění pro přidání nové transakce. Pokud ne, je mu zobrazena chybová hláška.

4. Systém zkontroluje, zda je entita, která mu přišla od uživatele validní. Pokud ne, je uživateli zobrazena chybová hláška.

5. Systém uloží entitu do databáze a přepočítá transakce nutné k vyrovnání účtů mezi uživateli v peněžence.

6. Uživateli je zobrazen detail peněženky s novou transakcí.



Obrázek 3.6: Diagram aktivit pro přidání nové transakce

3.5.2 Vytvoření nové peněženky

V této části jsou popsány jednotlivé kroky pro vytvoření nové peněženky.

1. Přihlášený uživatel klikne na tlačítko pro vytvoření nové peněženky.

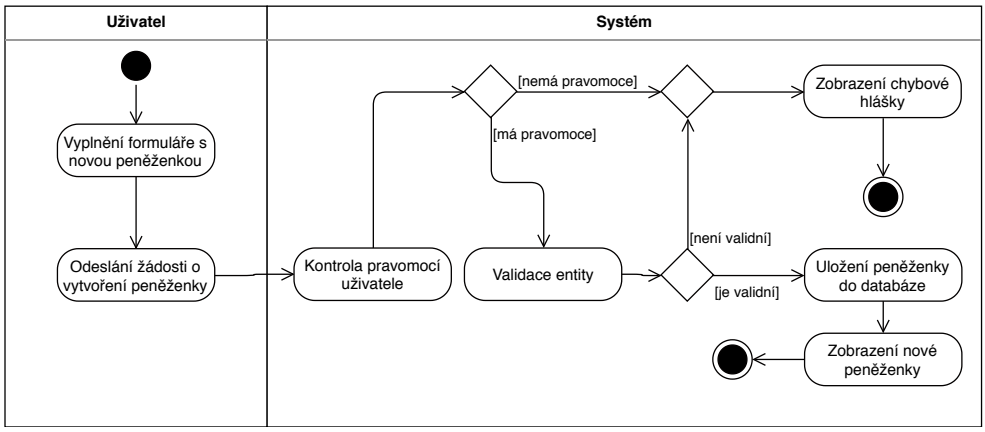
2. Uživatel vyplní formulář (se jménem peněženky a členy peněženky) a odešle požadavek na vytvoření peněženky do systému.

3. Systém zkontroluje, zda má uživatel dostatečná oprávnění pro vytvoření peněženky. Pokud ne, je uživateli zobrazena chybová hláška.

4. Systém provede validaci jednotlivých atributů entity. Pokud entita nebude validní, je uživateli zobrazena chybová hláška.

5. Systém uloží peněženku do databáze.

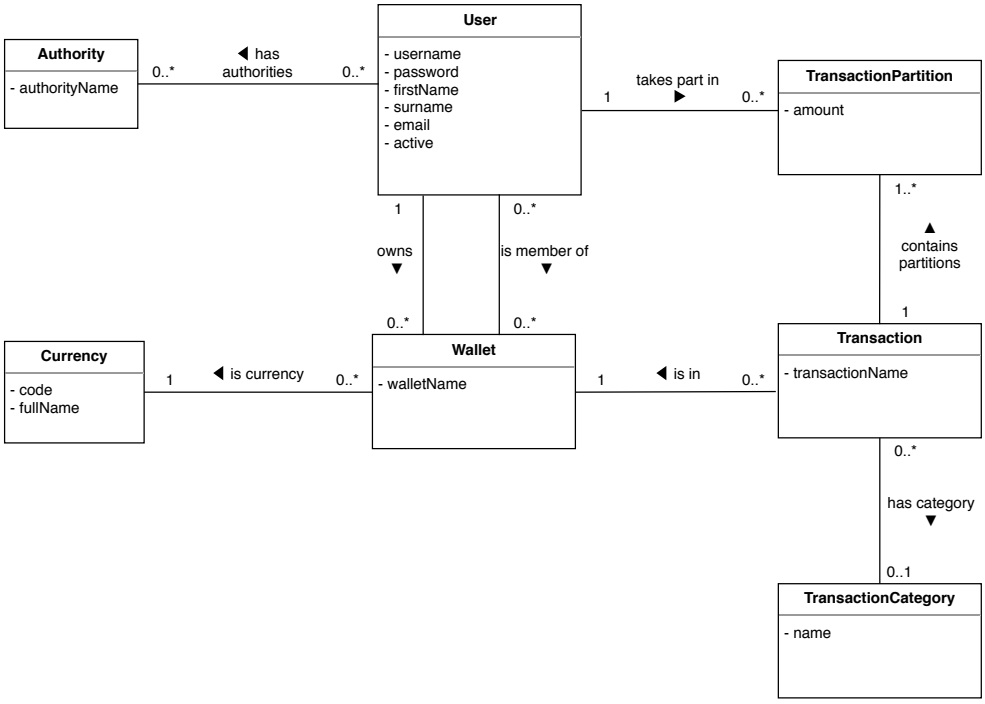
6. Uživateli je zobrazena nová peněženka.



Obrázek 3.7: Diagram aktivit pro vytvoření nové peněženky

3.6 Doménový model

V této sekci se nachází doménový model 3.8 a popis entit. Na diagramu jsou vidět vztahy a násobnosti těchto vztahů mezi entitami. U jednotlivých entit jsou pro přehlednost rozepsány názvy atributů a jejich popis.



Obrázek 3.8: Doménový model

3.6.1 Entita Wallet

Entita reprezentující peněženku uživatele. Peněženka je objekt, kam si uživatelé mohou ukládat své transakce – kolik peněz, a komu zaplatili, případně obdrželi. V jedné peněžence může být členy více uživatelů. U každé peněženky si pak uživatelé mohou zjistit, kolik dluží resp. kolik půjčili ostatním členům peněženky.

Název atributu	Popis atributu
walletName	Jméno peněženky

Tabulka 3.1: Atributy entity Wallet

3.6.2 Entita User

Entita reprezentující uživatele. Uživatel je osoba, která se může přihlašovat do aplikace a provádět dostupné operace. Součástí této entity jsou informace o uživateli.

Název atributu	Popis atributu
username	Přihlašovací jméno
password	Heslo
firstName	Křestní jméno
surname	Příjmení
email	Email
active	Je tento účet aktivní

Tabulka 3.2: Atributy entity User

3.6.3 Entita Currency

Entita reprezentující měnu. U každé měny je uvedena její třípísmenná zkratka a celý název. Měna je pak přiřazena k peněžence. Každá peněženka může mít transakce jen v jedné měně.

Název atributu	Popis atributu
code	Zkratka měny
fullName	Celé jméno měny

Tabulka 3.3: Atributy entity Currency

3.6.4 Entita Transaction

Entita reprezentující transakci. Každá transakce je složena pouze ze jména transakce a entit TransactionPartition. Každá transakce je rozdělena do částí (Transaction Partition), ve kterých jsou zaznamenány změny stavů kont jednotlivých uživatelů.

Název atributu	Popis atributu
transactionName	Jméno transakce

Tabulka 3.4: Atributy entity Transaction

3.6.5 Entita TransactionPartition

Entita reprezentující část platby v transakci. Každá část je složena z uživatele a částky, která mu byla přičtena (resp. odečtena) v dané transakci. Například pokud je transakce, kde uživatel A zaplatil za uživatele B a C částku 20, bude pak existovat transakce složená ze tří `TransactionPartition`. První `TransactionPartition` bude obsahovat částku +40 pro uživatele A a dvě `TransactionPartition` s částkou -20 pro uživatele B a C.

Součet všech částek `TransactionPartition` pro jednu transakci se vždy musí rovnat nule.

Název atributu	Popis atributu
amount	Částka

Tabulka 3.5: Atributy entity TransactionPartition

3.6.6 Entita TransactionCategory

Entita reprezentující kategorii transakce. Kategorie transakce slouží jen pro uživatele pro zpřehlednění jeho transakcí. Každá kategorie má pouze jméno, kteér jí může být přiřazeno.

Název atributu	Popis atributu
name	Jméno kategorie

Tabulka 3.6: Atributy entity TransactionCategory

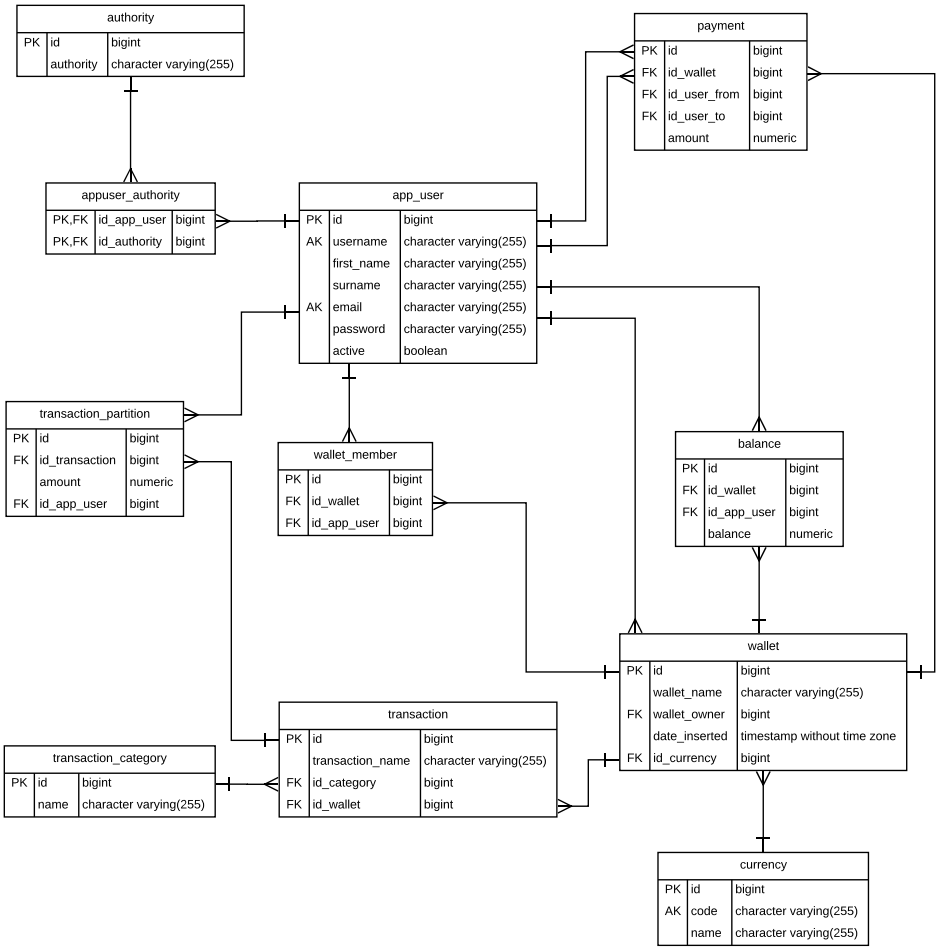
3.6.7 Entita Authority

Entita reprezentující autoritu uživatele. Podle jednotlivých autorit má pak uživatel přístup k různým operacím v aplikaci. Uživatel může mít více autorit.

Název atributu	Popis atributu
authorityName	Jméno autority

Tabulka 3.7: Atributy entity Authority

3.7 Databázový model



Obrázek 3.9: Databázový model

Databázový relační model slouží k zobrazení rozložení jednotlivých tabulek v databázi. U každé tabulky jsou sepsány sloupce (atributy), ze kterých se tabulka skládá. Databázový model je univerzální pro různé typy relačních databází. Díky tomu lze pak model použít v různých typech databází.

Databázový model aplikace vychází z doménového modelu. K jednotlivým atributům jsou v tabulkách přiřazeny jejich datové typy a integritní omezení. Oproti doménovému modelu zde přibyly tabulky pro dekompozici M:N vazeb a tabulky `payment` a `balance`, jejichž význam je vysvětlen v následujících sekcích. U jednotlivých tabulek jsou uvedeny primární klíče (PK) a cizí klíče (FK).

3.7.1 Tabulka Balance

V tabulce `balance` jsou uloženy aktuální stavy účtů uživatelů v jednotlivých peněženkách. Stav konta se počítá z transakcí uživatelů. Po každé transakci se od aktuálního stavu konta uživatele buď přičte nebo odečte částka, podle toho, zda uživatel v transakci figuruje jako plátce, nebo příjemce.

Data v této tabulce slouží pouze k rychlejšímu výpočtu vyrovnávání dluhů mezi uživateli v peněženkách. Data v této tabulce tedy není nutné ukládat, ale bylo by při každém výpočtu algoritmu potřeba vypočítat stavy kont z transakcí, což by bylo časově náročné, a proto se ukládají stavy kont průběžně.

3.7.2 Tabulka Payment

V tabulce `payment` jsou uloženy platby pro jednotlivé peněženky, které je potřeba vykonat, aby se vyrovnal stav účtů mezi všemi uživateli. Data z tabulky se stejně jako u tabulky `balance` dají vypočítat, ale z důvodu časové náročnosti výpočtu je zvolena varianta, kdy se průběžně aktualizuje tato tabulka. Tabulka se aktualizuje při každé transakci.

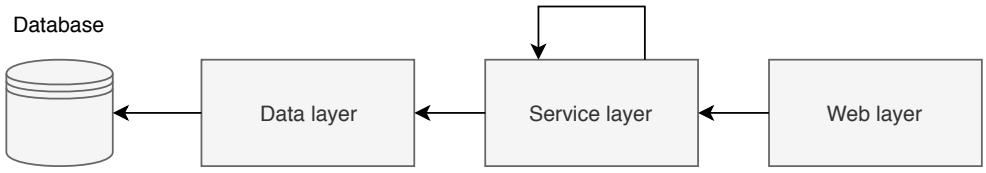
3.8 Architektura

3.8.1 Rozdělení serverové aplikace

Serverová aplikace je rozdělená do tří vrstev. První vrstva je vrstva komunikující s databází. V této vrstvě jsou namapované entity na databázový model. Úkolem této vrstvy je zpracovávat databázové úkony z vyšších vrstev.

Druhou vrstvou je servisní vrstva. Ta má za úkol pomocí dostupných operací zpracovávat požadavky klientů. V této vrstvě je umístěna veškerá logika aplikace. Třídy ze servisní vrstvy mohou, pokud je to zapotřebí, volat další třídy ze servisní vrstvy.

Třetí vrstvou je webová vrstva. V této vrstvě se nacházejí třídy s kontrolery. Tyto kontrolery obsluhují endpointy, které může klient volat. Z této vrstvy jsou volány třídy ze servisní vrstvy.



Obrázek 3.10: Architektura systému

3.8.2 Komunikace mezi serverem a klientem

V této části je popsáno, jak probíhá komunikace klienta se serverem. Serverová aplikace vystavuje endpointy, které klientská aplikace volá. Před voláním endpointů se musí klient autorizovat pomocí OAuth 2. Z tohoto volání získá klient token, kterým následně autorizuje svá volání. Před každým zpracováním volání je automaticky validován autorizační token klienta pomocí Spring Security.

3.9 Zabezpečení aplikace

3.9.1 Přihlášení uživatele

Zabezpečení aplikace je řešeno pomocí Spring Security. Autorizace klientů je řešena pomocí protokolu OAuth 2. Každá klientská aplikace používající API této aplikace, musí mít své přihlašovací údaje (username aplikace a application secret). Uživatel, který se přihlašuje do aplikace pak k přihlášení potřebuje přihlašovací jméno a heslo.

Přihlašovací údaje uživatele jsou uloženy v databázi v tabulce `app_user`. Sloupec `username` obsahuje uživatelské jméno a sloupec `password` obsahuje heslo zahešované algoritmem BCrypt.

Před samotným voláním API si musí klient zažádat o přístupový token do aplikace. Ten získá zavoláním metody pro získání tokenu s přihlašovacím jménem aplikace, client secret a přihlašovacími údaji uživatele. Pokud je požadavek na přihlášení úspěšný je mimo jiné klientovi vrácen přístupový token a token pro obnovu přístupového tokenu.

Přístupový token je určen k tomu, aby se posílal společně s požadavky klienta na API a server díky tomu ověřuje, že uživatel má přístup do aplikace. Tento token, ale má určitou expiraci (zbývající čas do expirace tokenu je součástí odpovědi na HTTP požadavek na získání tokenu). Po expiraci tokenu je potřeba použít token pro obnovu přístupového tokenu tzv. refresh token. S tímto tokenem se zavolá požadavek na server a klientovi v odpovědi přijde nový přístupový token.

3.9.2 Dodatečná bezpečnostní opatření

Knihovna Spring Security automaticky nastavuje některá bezpečnostní opatření bránící proti určitým typům útoků. Například do HTTP odpovědí jsou automaticky přidávány hlavičky bránící proti útokům typu XSS, CSRF.

Při použití aplikace jsou pak jak ve webové vrstvě, tak i v servisní vrstvě ověřovány pravomoce uživatele. Pokud by uživatel upravoval například transakce, ke kterým by neměl přístup, vrátí mu aplikace výjimku, zabezpečení je podrobněji popsáno u jednotlivých vrstev.

3.10 Použité technologie

3.10.1 Spring

Spring Framework je open-source aplikační framework podporující jazyky Java, Kotlin a Groovy. Spring patří mezi nejpoužívanější frameworky na trhu[12]. Samotný Spring je rozdělen do knihoven podle jejich účelu. Použité knihovny v mém projektu jsou Spring Boot, Spring Security, Spring Data JPA a Spring MVC.

Spring Boot je rozšíření základní Spring knihovny, které usnadňuje konfiguraci aplikace. Při použití Spring Boot není třeba konfigurovat jednotlivé knihovny, ale Spring Boot je automaticky nakonfiguruje. Pro jednodušší práci se Spring Boot jsou pro uživatele připraveny tzv. startovací balíčky, kde si lze podle typu aplikace vybrat potřebný balíček. Například balíček `spring-boot-starter-data-jpa` obsahuje potřebné knihovny pro datovou vrstvu aplikace (Spring Data JPA, Hibernate ...).

Spring Security je rozšíření určené pro zabezpečení aplikace. Poskytuje možnost autorizace a autentizace uživatele v aplikaci. Další funkcí je obrana proti některým typům útoků (clickjacking, CSRF). V aplikaci je použita knihovna Spring Security a v ní implementace protokolu OAuth 2.

Spring Data JPA je rozšíření určené pro práci v datové vrstvě aplikace. Toto rozšíření se stará o komunikaci s databází. Knihovna navíc v porovnání s Java Persistence API snižuje množství kódu k vykonávání jednotlivých databázových operací.

Spring MVC je rozšíření poskytující MVC architekturu. Pomocí tohoto rozšíření lze vytvářet jednotlivé endpointy aplikace, které pak bude volat klientská aplikace.

3.10.2 PostgreSQL

PostgreSQL je open-source objektově-relační databázový systém. PostgreSQL je díky tomu, že se jedná o projekt s otevřeným zdrojovým kódem, jeden z nejpoužívanějších databázových systémů. Díky velké uživatelské základně se dá považovat tento systém za spolehlivý.

3.11 Použité nástroje pro vývoj

V této sekci jsou popsány jednotlivé nástroje použité během vývoje této aplikace. Jedná se jak o IDE, tak i o další programy, které byly použity.

3.11.1 IntelliJ IDEA

Pro vývoj backendové části aplikace bylo použito IDE IntelliJ IDEA. Výhodou použití je připravenost IDE na vývoj backendových aplikací v různých frameworcích, mimojiné i framework Spring. Další výhodou je, že IDE obsahuje plugin pro vývoj v jazyce Kotlin (IntelliJ IDEA i Kotlin vznikly ve společnosti JetBrains). Tento program je navíc pro studenty pro vývoj nekomerčních aplikací zadarmo.

3.11.2 Git

Pro verzování zdrojového kódu aplikací je použit nástroj Git. Git je v dnešní době nejpoužívanějším nástrojem pro správu zdrojového kódu[13] a díky jeho nenáročnosti na obsluhu je vhodnou volbou. Podpora nástroje Git je i v IDE používaných při vývoji aplikace.

3.11.3 Android Studio

Pro vývoj frontendové části aplikace je použito Android Studio. Jedná se o oficiální nástroj pro vývoj Android aplikací. Obsahuje mnoho užitečných funkcí, které vývojářům usnadňují práci například i pro práci s Android emulátorem a sestavením aplikací. Nástroj je zdarma ke stáhnutí.

3.11.4 Docker

Docker je nástroj, který umožňuje uživateli spouštět různé aplikace ve virtualizovaných oddělených kontejnerech. Jednotlivé kontejnery obsahují pouze aplikace a soubory, které aplikace vyžaduje, ale neobsahuje operační systém. Tím se odlišuje od standardních virtuálních strojů. Pro běh aplikací je přímo využíván hostující operační systém.

Pro vývojáře má Docker výhodu, že pokud aplikaci spustí na jiném systému, tak kontejnerizovaná aplikace se bude všude chovat stejně. Docker má oproti ostatním nástrojům výhodu, že je poměrně jednoduchý na nastavení.

3.11.5 SonarQube

SonarQube je nástroj pro kontrolu kvality zdrojového kódu. Nástroj SonarQube vyhledává mimo jiné potenciální chyby, bezpečnostní zranitelnosti a duplicity kódu. Aplikace podporuje mnoho jazyků, včetně jazyka Kotlin. Nástroj je s určitými omezeními k použití zdarma.

Do aplikace je možné automaticky nahrávat výsledky testů aplikace. V aplikaci je pak vidět úspěšnost testů a velikost pokrytí aplikace testy.

Kapitola 4

Realizace

V této kapitole je popsáno, jak byla implementovaná serverová a mobilní aplikace.

Serverová aplikace, jak bylo popsáno, se skládá ze tří vrstev. V následujících sekcích jsou popsány postupy použité při implementace jednotlivých vrstev.

4.1 Datová vrstva

Datová vrstva má na starost práci s databází. Z jiných vrstev by se nemělo k databázi přistupovat. Tato vrstva se skládá z entit a repozitářů.

4.1.1 Entita

Entita je jeden objekt, který je ukládán do databáze. V Kotlin kódu je definována pomocí `data class` a příslušných anotací. Každá entita je namapovaná na tabulku v databázi.

Příkladem může být namapování entity `Currency`. Anotací `@Entity` označíme entitu a anotací `@Table` je předáno jméno tabulky, na kterou se tato entita namapuje v databázi[14]. Pokud by nebyla použita anotace `@Table`, je použita jako jméno tabulky název třídy. V tomto případě tedy není anotace nutná, ale pro zpřehlednění jsou anotace u všech entit.

Poté následuje výčet proměnných v entitě v konstruktoru třídy. Proměnné se namapují na databázové sloupce pomocí proměnné `@Column`. U id entity se pak speciálně používají anotace `@Id` a `@GeneratedValue`, která označuje jakým způsobem se generuje id třídy.

```
@Entity
@Table(name = "currency")
data class Currency (

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "id")
        var id: Long?,

        @Column(name = "code")
        var code: String,

        @Column(name = "name")
        var fullName: String
)
```

Ukázka kódu 4.1: Ukázka namapovaní entity Currency

4.1.2 Repozitáře

Repozitáře jsou tzv. DAO, tedy objekty, které přistupují přímo do databáze. Přes tyto objekty se posílají do databáze příkazy pro vykonání databázových operací. V knihovně Spring Data JPA jsou repozitáře reprezentovány rozhraním. Součástí knihovny Spring Data JPA je i několik předpřipravených repozitářů, které lze rozšířit a vývojář nemusí psát některé za něj implementované funkce. Příkladem může být `JpaRepository`, které má v sobě implementované metody na základní operace s entitami – ukládání, načítání, mázání a další (tzv. CRUD operace – create, read, update, delete).

Vývojář si může v případě potřeby dopsat do repozitáře vlastní metody pro přístup k datům. K takovýmto metodám je pak potřeba dopsat i databázové query, které se má použitím metody zavolat. U některých jednodušších metod není toto query potřeba psát, protože Spring dokáže některé metody rozpoznat a vytvořit si k nim příslušné query. Například pokud entita obsahuje atribut `name`, tak při napsání metody `findByName(name: String)` není třeba dopisovat query, ale je automaticky doplněno query vyhledávající podle sloupce `name`.

Tato rozhraní lze pak používat v dalších vrstvách aplikace pro přístup k datům z aplikace.

```
@Repository
interface TransactionPartitionRepository
 : JpaRepository<TransactionPartition, Long> {

    fun findTransactionPartitionByTransactionId(transactionId: Long)
     : List<TransactionPartition>

}
```

Ukázka kódu 4.2: Ukázka repozitáře TransactionPartitionRepository

4.2 Servisní vrstva

V této sekci je popsáno, jakým způsobem je realizovaná servisní vrstva. V servisní vrstvě se nachází servisní třídy, které vykonávají operace s entitami a logika aplikace. Třídy a metody ze servisní vrstvy jsou pak používány dalšími servisními třídami, případně třídami z webové vrstvy.

4.2.1 Zabezpečení

Na úrovni servisní vrstvy se řeší bezpečnost, zda má klient, který posílá požadavek příslušná oprávnění k tomu vykonávat daný požadavek. V případě, že klient nemá dostatečná oprávnění, je vyhozena výjimka `AccessDeniedException` a server požadavek nezpracuje. Samotné zpracování výjimky má na starosti webová vrstva, která na základě typu výjimky vrací odpověď klientovi.

4.2.2 Výpočet plateb pro vyrovnání účtů

Základní funkcí aplikace má být možnost vyrovnat stav účtů plateb mezi uživateli v jednotlivých peněženkách. Je tedy potřeba vytvořit algoritmus, který bude z jednotlivých transakcí mezi uživateli schopen spočítat, které platby je třeba uskutečnit, aby byl stav účtů mezi uživateli vyrovnán. Pro implementaci algoritmu byl použit již navržený dostupný algoritmus [15].

Při počítání plateb není potřeba znát historii transakcí mezi uživateli, ale je potřeba znát pouze stav, kolik každý uživatel celkem zaplatil, případně kolik za něj bylo zaplaceno, tedy stavy kont, které jsou uložené v tabulce `balance`. Z těchto stavů se pak počítá minimální počet transakcí tak, aby se vyrovnaly dluhy mezi všemi uživateli tedy, aby byl stav všech kont po vykonání těchto transakcí nulový.

4.2.2.1 Příklad transakcí

Na následujícím příkladu lze vidět, jak se aktualizují stavy kont uživatelů po jednotlivých transakcích. Kladný stav konta, znamená, že uživatel ve všech transakcích více platil za ostatní, než obdržel. Záporný stav konta znamená, že za uživatele bylo více placeno, než kolik on sám zaplatil.

Mějme 4 osoby, pojmenované Alice, Bob, Carol, Dave. Počáteční stav konta, každé osoby je 0$.

Alice	Bob	Carol	Dave
0$	0$	0$	0$

Tabulka 4.1: Příklad výpočtu plateb, počáteční stav

Poté zaplatí Alice za Boba 20$. Alici se tedy v tabulce přičte 20$ a Bobovi se v tabulce odečte 20$. Stav kont po transakci bude následující.

Alice	Bob	Carol	Dave
20$	–20$	0$	0$

Tabulka 4.2: Příklad výpočtu plateb, stav po 1. transakci

Dalšími transakcemi bude, že Bob zaplatí za Carol 5$ a Dave zaplatí za Carol 10$. Stavy kont po těchto dvou transakcí jsou zobrazeny v následujících tabulkách. Za povšimnutí stojí, že součet všech stavů kont je vždy roven nule.

Alice	Bob	Carol	Dave
20$	–15$	–5$	0$

Tabulka 4.3: Příklad výpočtu plateb, stav po 2. transakci

Alice	Bob	Carol	Dave
20$	–15$	–15$	10$

Tabulka 4.4: Příklad výpočtu plateb, stav po 3. transakci

Cílem algoritmu je tedy vypočítat nejmenší možný počet transakcí takový, že stav všech kont po uskutečnění těchto transakcí bude pro všechny uživatele nulový.

V tomto případě je více možností jak vyrovnat všechny stavy. Jedním z možných výsledků, který by měl algoritmus vrátit je zobrazen v následující tabulce.

Platba od	Platba komu	Částka
Bob	Alice	15$
Carol	Alice	5$
Carol	Dave	10$

Tabulka 4.5: Příklad výpočtu plateb, transakce pro vyrovnání účtů

K vyrovnání tedy stačí tři transakce. Po uskutečnění těchto transakcí budou stavy všech uživatelů 0$ a všechny platby tedy budou vyrovnány.

4.2.2.2 Algoritmus pro výpočet plateb

Algoritmus pro výpočet plateb funguje na jednoduchém principu. Nalezení transakce probíhá následovně. Na počátku máme seznam uživatelů, jejichž stav konta je nenulový.

Poté je hledána transakce podle těchto pravidel

1. Je hledána dvojice osob, jejichž stav konta se dá jednou transakcí vynulovat. Například jedna osoba má 20$ a druhá má –20$, tak zaplacením 20$ od první osoby druhé se stavy kont vynulují.

2. Pokud se nepodaří najít dvojici osob pomocí prvního pravidla, je hledána taková dvojice transakcí, která by vedla k tomu, že první transakcí se vynuluje konto jednoho uživatele a druhou transakcí se vynulují konta dalších dvou uživatelů.

   Například jsou tři uživatelé jejich stavy kont jsou 15$, –10$ a –5$. Pomocí první transakce, kdy je zaplaceno od třetího uživatele prvnímu 5$, se vynuluje stav konta třetího uživatele. Druhou transakcí, kdy druhý uživatel zaplatí prvnímu 10$, se vynulují stavy kont prvního a druhého uživatele a všichni tři uživatelé budou mít nulový stav konta.

   Tento problém se dá řešit pomocí hledání trojic kont, takových, že součet stavu těchto kont je roven nule. Při implementaci jsem vycházel z dostupných algoritmů pro hledání trojic čísel v seznamu o daném součtu [16].

3. Pokud se nepodaří uplatnit první dvě pravidla, vybere se libovolný uživatel a jednou transakcí se vynuluje stav jeho konta transakcí vůči dalšímu uživateli.

Po každém uplatnění jednoho z pravidel se ze seznamu uživatelů odstraní uživatelé, jejichž stav konta je nulový, protože už mají vyrovnaný stav účtu a není třeba s nimi dále provádět nějaké výpočty.

Pravidla se provádí do doby, dokud existují uživatelé, kteří mají nenulový stav konta. Opakovaným prováděním pravidel se postupně všechna konta vynulují.

4.2.2.3 Implementace algoritmu

Samotný algoritmus je v aplikaci implementován tak, jak je popsán v předchozí sekci. Na vstupu dostane algoritmus seznam s entitami `Balance`. Seznam obsahuje stav kont všech uživatelů z vybrané peněženky. Výstupem je množina s entitami `Payment`, obsahující transakce nutné k vyrovnání stavu kont mezi uživateli.

4.2.3 Transakce

V této části jsou popsány jednotlivé operace s transakcemi a postup, co v aplikaci probíhá během těchto operací.

Požadavek na provedení operace s transakcemi přichází z webové vrstvy. Základními operace s transakcí je přidávání, mazání a získání detailu transakce. Nejsložitější operací je přidávání transakce, její průběh je popsán v následující části.

4.2.3.1 Přidávání transakce

Při přidávání transakce přijde z webové vrstvy DTO složené z údajů pro uložení transakce a seznamu s částmi transakce `TransactionPartitionDto`. Na základě toho se v servisní třídě vytvoří entita s transakcí a entity s částmi transakce (TransactionPartition) a vše se uloží do databáze.

Při každém uložení části transakce je nutné přepočítat stav konta uživatele v dané peněžence, proto se pro příslušného uživatele mění hodnota v tabulce `Balance`. Poté co je celá transakce včetně jednotlivých částí uložena, jsou přepočítány nutné transakce pro vyrovnání účtů.

4.3 Webová vrstva

Ve webové vrstvě se nachází kontrolery. Tyto třídy mají za úkol poskytovat klientům API. V každém kontroleru jsou metody, které zpracovávají HTTP požadavky klientů. Požadavek z webové vrstvy je předán do servisní vrstvy ke zpracování. Webová vrstva je tedy jediná vrstva, ke které mají přístup klientské aplikace.

4.3.1 Kontrolery

Kontrolery jsou třídy, které mají za úkol spravovat API. Každý kontroler má přiřazenou URL cestu, na které vyřizuje požadavky klientů. Kontrolery jsou v kódu označené anotací `@RestController` a URL, které obsluhují se určuje pomocí anotace `@RequestMapping(path)`, kde za `path` se dosadí URL cesta daného kontroleru.

```
@RestController
@RequestMapping("/api/user/")
class UserController {

    @Autowired
    private lateinit var userService: AppUserService

    @GetMapping("/exists/{username}",
                produces = [MediaType.APPLICATION_JSON_VALUE])
    fun userExists(@PathVariable("username") username: String)
        : ResponseEntity<Boolean> {
        return ResponseEntity(userService.userExists(username),
                              HttpStatus.OK)
    }
}
```

Ukázka kódu 4.3: Ukázka kontroleru

V serverové aplikaci jsou vytvořeny 4 kontrolery

- `WalletController` Vyřizuje požadavky klienta na CRUD operace s peněženkami.
- `UserController` Vyřizuje požadavky klienta na CRUD operace s uživateli aplikace.
- `TransactionController` Vyřizuje požadavky klienta na CRUD operace s transakcemi.
- `PaymentController` Vyřizuje požadavky na získání nutných plateb k vyrovnání účtu ve zvolené peněžence

4.3.2 Http stavové kódy

Po zpracování požadavku je uživateli vráceno buď příslušné DTO, nebo v aplikaci byla někdy v průběhu vyhozena výjimka. Na základě těchto informací, které se zpracují v kontroleru se uživateli vrací odpověď s příslušným HTTP stavovým kódem. Seznam použitých kódu s vysvětlivkami je v následující tabulce.

Kód	Název	Popis
200	OK	Požadovaná operace proběhla v pořádku
201	Created	Požadovaná operace na vytvoření nového objektu proběhla v pořádku
204	No Content	Požadavek byl úspěšně zpracován, ale odpověď nevrací žádný obsah
404	Not Found	Požadavek nemohl být zpracován, protože nelze nalézt požadovanou entitu.
422	Unprocessable Entity	Požadavek je zaslán správně, ale kvůli chybě v přijaté entitě ho nelze zpracovat.
500	Internal Server Error	Došlo k chybě na straně serveru

Tabulka 4.6: Používané HTTP stavové kódy

4.3.3 Zabezpečení webové vrstvy

Zabezpečení webové vrstvy probíhá pomocí Spring Security. V konfiguraci se nastavuje, které koncové body jsou zabezpečeny a jaké pravomoce jsou potřeba, aby k nim mohl klient a uživatel přistupovat.

Nastavuje se, zda musí být uživatel pro přístup ke koncovým bodům přihlášen, případně jaké musí mít práva. Například u koncového bodu pro registraci uživatele není vyžadováno přihlášení, u všech ostatních koncových bodů je vyžadováno, aby byl uživatel autorizován.

Dodatečná kontrola zabezpečení probíhá i v servisní vrstvě, ze které může přijít výjimka `AccessDeniedException`, která se vrací v případě, že uživatel nemá dostatečné pravomoce pro vykonání operace. Tato výjimka je pak vrácena uživateli s příslušnou chybovou hláškou.

4.3.4 Zpracování výjimek

Během zpracování požadavku klient může dojít k tomu, že aplikace v některé části vyhodí výjimku. Výjimky, které nejsou zpracovány v nižších vrstvách a vyskytnou se ve webové vrstvě jsou zpracovány pomocí tříd zachytávající tyto výjimky. Tyto třídy anotované `@ControllerAdvice` v sobě obsahují metody anotované `@ExceptionHandler`, u kterých se určuje typ zpracované výjimky. Poté pokud se vyskytne v kontroleru výjimka, je volána tato metoda, která výjimku zpracuje a vrátí klientovi příslušnou odpověď s chybovou hláškou.

Po zpracování některých výjimek je klientovi vrácena v odpovědi na jeho požadavek vytvořená datová třída `ApiError`. Tato třída obsahuje typ chyby a zprávu pro klienta. Klientovi se tak nevrací celý výpis výjimky, ke kterému by klient neměl mít přístup, ale jen zprávu pro uživatele a data, která určí vývojář.

```
@ControllerAdvice
class RestExceptionHandler : ResponseEntityExceptionHandler() {

    @ExceptionHandler(InvalidEntityException::class)
    fun handleInvalidEntityException(exception: InvalidEntityException)
    : ResponseEntity<ApiError> {
        val apiError = ApiError(HttpStatus.UNPROCESSABLE_ENTITY,
                exception.message,
                exception.errors)
        return ResponseEntity(apiError, apiError.status)
    }
}
```

Ukázka kódu 4.4: Ukázka zpracování výjimky

4.4 Realizace Android aplikace

V této části jsou popsány postupy při realizaci Android aplikace. Aplikace slouží jako klient k serverové aplikaci, veškerá data jsou tedy odesílány a přijímány ze serverové aplikace. Mobilní aplikace slouží k zobrazovaní těchto dat.

4.4.1 Aktivity a fragmenty

Základem aplikace jsou třídy, které dědí z tříd typu `Activity` a `Fragment`. Třída dědící ze třídy `Activity`, tzv. aktivita má za úkol zobrazovat uživatelské rozhraní, zpracovávat uživatelský vstup a vyřizovat požadavky uživatele. Dalo by se říct, že aktivita reprezentuje jednu obrazovku s uživatelským rozhraním.

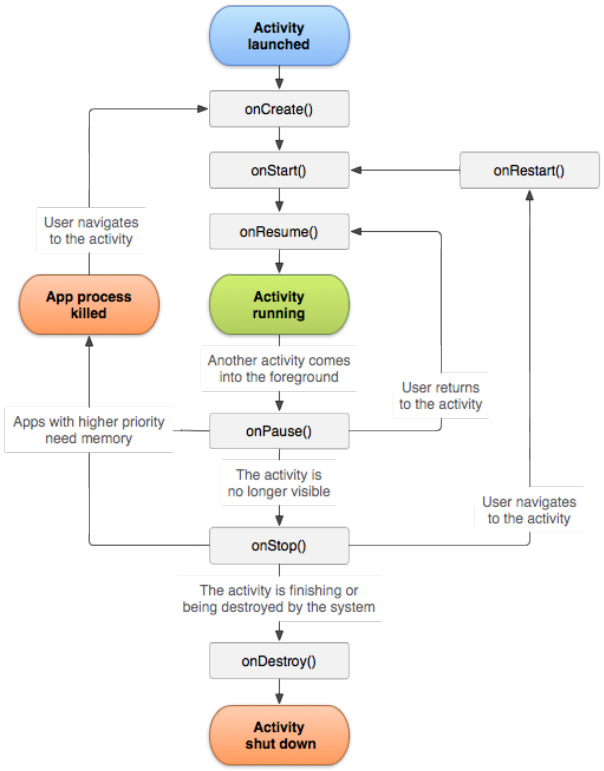
Mezi různými aktivitami lze libovolně přepínat, například na základě uživatelského vstupu.

Například pokud je zobrazena aktivita s domovskou stránkou aplikace a uživatel klikne na svou peněženku v seznamu peněženek, je mu zobrazena aktivita s detaily peněženky.

4.4.1.1 Životní cyklus aktivity

Každá aktivita má svůj vlastní životní cyklus, během kterého v závislosti na stavu dochází k volání metod aktivity. Například při spuštění aktivity je v aktivitě zavolána metoda `onCreate()`. Popis životního cyklu je na následujícím obrázku.

Díky metodám volaným během životního cyklu aktivity lze zjistit, kdy dochází k vytvoření a zániku aktivity, případně, kdy uživatel opustil, nebo se navrátil do aktivity a na základě toho provádět potřebné operace v kódu.



Obrázek 4.1: Životní cyklus aktivity [1]

4.4.1.2 Uživatelské rozhraní

Uživatelské rozhraní, tedy určení rozložení jednotlivých komponent, je běžně definováno ve zvláštním XML souboru. V tomto souboru lze pomocí kompo-

nent z knihovny Android SDK vytvářet různé typy rozhraní. Součástí Android knihovny jsou běžně používané komponenty (tlačítka, textová pole, posuvníky...).

Jednotlivé komponenty jsou podtřídy třídy `View`. Třídy typu `View` reprezentují jeden typ objektu, který je zobrazován uživateli. Například `TextView` je `View` zobrazující textové řetězce, `ImageView` slouží k zobrazování obrázků.

Ke každé aktivitě která zobrazuje uživatelské rozhraní, obvykle existuje XML soubor, ve kterém je definováno uživatelské rozhraní k dané aktivitě. V aktivitě se pak během vytváření aktivity v metodě `onCreate()` nastavuje, který XML soubor s rozhraním má aktivita načíst.

4.4.1.3 Fragment

Součástí jednotlivých aktivit mohou být třídy dědící z třídy `Fragment`. Fragmenty slouží k zobrazování části uživatelského rozhraní v aktivitě. Aktivitu lze tedy rozdělit do několika fragmentů, kde v každé části aktivity se zobrazuje jiný fragment. Aktivita obsahující fragment, může jednotlivé fragmenty za svého běhu zobrazovat, skrývat a vykonávat s nimi další operace. Díky tomu, lze například rozdělit obrazovku na dvě části a v každé zobrazovat jiný fragment. Toho se dá využít například u tabletů, kdy je zobrazeno více fragmentů naráz.

Příkladem využití fragmentu může být aktivita zobrazující taby (záložky). Každý takový tab je reprezentován jedním fragmentem a aktivita pak má za úkol zobrazovat fragmenty se záložkami na základě uživatelského vstupu.

4.4.2 Komunikace s API

Aplikace komunikuje s API pomocí knihovny Retrofit v kombinaci s knihovnou OkHttp. Koncové body, které se volají, jsou definovány v rozhraních, kde pomocí anotací u metod se nastavuje volané URL a parametry volání. Volání API pak probíhá z aktivit, které potřebují získat tato data.

V následující části je ukázka kódu s rozhraním, kde jsou definovány metody a pak způsob, jakým se volá REST API.

```
interface TransactionService {

    @POST("/api/transaction/create/")
    fun createTransaction(@Body newTransactionDto: NewTransactionDto)
        : Call<Unit>
}
```

Ukázka kódu 4.5: Ukázka rozhraní s REST API metodou

```
getApp()
.getTransactionService()
.createTransaction(newTransactionDto)
.enqueue(object : CustomCallback<Unit?>(activity as Activity) {
        override fun onSuccess(responseObject: Unit?) {
            // zavolano v pripade uspesneho pozadavku
        }

        override fun onError(call: Call<Unit?>, response: Response<Unit?>) {
            // zavolano v pripade neuspesneho pozadavku
        }
})
```

Ukázka kódu 4.6: Ukázka volání REST API

Při inicializaci knihovny OkHttp jsou nastaveny interceptory starající se o přidávání tokenů a zabezpečení aplikace. Ke každému požadavku je přidán přístupový token, který je získán přihlášením uživatele. V případě, že dojde k expiraci tokenu, je token obnoven pomocí tzv. refresh tokenu.

Kapitola 5

Testování

V této části jsou popsány postupy použité při testování serverové aplikace a Android aplikace. Testy jsou dále rozděleny podle kategorií příslušných testů.

Některé testy se spouští pomocí skriptu v Gradle souboru. Výsledky z těchto testů lze pak vidět buď přímo ve výpisu při spuštění skriptu, nebo v nástroji SonarQube, spuštěného v Docker kontejneru.

5.1 Kategorie testů

5.1.1 Smoke testy

Smoke testy slouží ke kontrole systému, zda jsou po změnách v systému, všechny klíčové části funkční. Cílem je odhalení chyb, které by mohly ohrozit chod systému.

5.1.2 Jednotkové testy

Jednotkové testy slouží k otestování nejmenších částí kódu. Testují se tedy hlavně jednotlivé metody a třídy. Takovýto test může sloužit například k ověření logiky servisních tříd. Dále je pomocí testů ověřena funkčnost algoritmu pro výpočet plateb.

Pro testování je na jednotkové testy v aplikaci využit testovací framework JUnit 5.

5.1.3 Integrační testy

Integrační testy slouží k testování funkčnosti komunikace mezi jednotlivými komponentami aplikace. Integrační testy jsou spuštěny po proběhnutí jednotkových testů.

5.2 Testování API systému

Pro otestování API byla využita kombinace jednotkových a integračních testů. Pro testování aplikace, která používá framework Spring, jsou připraveny nástroje ulehčující testování. Třídy a jejich metody testující API jsou nakonfigurovány pomocí anotací obsažených v knihovnách Spring a JUnit 5.

Při každém testu je odeslán HTTP požadavek na API a je kontrolováno, zda API pro daný vstup vrátí očekávaná data. K zasílání požadavků je použita třída `MockMvc`. Po zaslání je ve většině testů zkontrolován stavový HTTP kód a obsah odpovědi, která je přijata.

V některých případech je třeba nastavit vlastní chování některých Spring Bean.

Pokud je potřeba nastavit vlastní chování některé Spring Bean, je využit tzv. mock. Pomocí něho lze nastavit chování libovolné Spring Bean. Například lze nastavit chování autentizace tak, že není třeba při každém testu autentizovat uživatele, ale uživatel je automaticky přihlášen.

Pomocí anotací `@Sql` a `@SqlGroup` se před a po spuštěním testů spouštějí databázové skripty. Tímto způsobem lze připravit testovací data. V ukázce kódu je skript vytvářející testovací data a skript, která tato data po testu maže.

Dále je u testu použita anotace `@WithMockUser`. Pomocí této anotace je vytvořen dočasně uživatel, který je autentizovaný. Před voláním metod tak není třeba žádat o token, ale stačí volat pouze požadavky na API.

```
@SpringBootTest
@AutoConfigureMockMvc
@RunWith(SpringRunner::class)
@SqlGroup(
        Sql(value = ["/db_scripts/create/create_test_users.sql"]),
        Sql(value = ["/db_scripts/drop/delete_test_users.sql"],
            executionPhase = Sql.ExecutionPhase.AFTER_TEST_METHOD)
)
internal class UserControllerTest {

    @Autowired
    private lateinit var mockMvc: MockMvc

    @Test
    @WithMockUser
    fun `test check user exists`() {
        mockMvc.perform(get("/api/user/exists/jan.novak"))
                .andExpect(status().isOk)
                .andExpect(content().string(`is`("true")))
    }
}
```

Ukázka kódu 5.1: Příklad testování API

5.3 Testování Android aplikace

Testování Android aplikace probíhalo pomocí frameworku Espresso a jako volné testování bez scénáře. Po přidání důležité funkce bylo daná funkce otestována a zároveň bylo otestováno, zda jsou stále funkční všechny klíčové funkce aplikace. Při testování byly testovány i krajní případy použití aplikace.

5.3.1 Testování pomocí frameworku Espresso

Testování uživatelského rozhraní aplikace probíhá pomocí frameworku Espresso. Pro jednotlivé testy, se v kódu definují testovací scénáře pomocí připravených funkcí. V kódu se tedy definují jednotlivé operace, například na co se má během testu kliknout, do jakého místa vepsat text a další potřebné operace. Zároveň lze ověřovat, zda je zobrazováno očekávané uživatelské rozhraní.

Test je spuštěn v běžící aplikaci buď v Android emulátoru, nebo na připojeném zařízení a postupně se vykonávají jednotlivé operace definované v testu.

Příklad takového testu je v následující ukázce kódu, kde je testováno přihlášení uživatele. Scénář takového testu je následující:

1. Je spuštěna aplikace.
2. Je vyplněno textové pole pro přihlašovací jméno textem `test_account4`.
3. Je vyplněno textové pole pro přihlašovací jméno textem `qwert`.
4. Je kliknuto na přihlašovací tlačítko.
5. Proběhne kontrola, zda je úspěšně přihlášeno a je zobrazena domovská stránka. aplikace

Tento scénář je pak zapsán v kódu testu. Pomocí funkce `onView()` se vybere `View` a dále se definuje, jaká operace se má nad tímto `View` provést. V ukázce jsou použité operace `replaceText()`, které nastaví do textových polí text, dále je použita operace `click()`, která provede kliknutí na požadované `View` (v tomto případě na tlačítko přihlášení). Na konci testu je ověřeno, že se uživateli zobrazila aktivita s domovskou stránkou `HomeScreenActivity`.

```
@RunWith(AndroidJUnit4::class)
class LoginActivityTest {

    @Rule
    @JvmField
    var mActivityTestRule = IntentsTestRule(LoginActivity::class.java)

    @Test
    fun testSuccessfulLogin() {
        //vyplneni uzivatelskeho jmena
        onView(withId(R.id.username_tv))
            .perform(replaceText("test_account4"), closeSoftKeyboard())
        //vyplneni hesla
        onView(withId(R.id.password_tv))
            .perform(replaceText("qwert"), closeSoftKeyboard())
        // kliknuti na prihlasovaci tlacitko
        onView(withId(R.id.sign_in_button))
            .perform(click())

        Thread.sleep(500) // cekani na odpoved serveru

        intended(hasComponent(
                    ComponentName(
                        InstrumentationRegistry
                            .getInstrumentation()
                            .targetContext,
                        HomeScreenActivity::class.java)))
    }
}
```

Ukázka kódu 5.2: Příklad testování přihlášení uživatele

5.4 Výsledky testování

Serverová i mobilní aplikace byly otestovány pomocí metod zmíněných v této kapitole. Během testování bylo nalezeno několik menších chyb, které byly následně odstraněny.

Kapitola 6

Nasazení a spuštění aplikace

6.1 Serverová aplikace

Nasazení a spuštění aplikace probíhá pomocí aplikace Docker. Serverová aplikace je rozdělená do několika kontejnerů. Pro běh aplikace jsou potřeba kontejnery obsahující databázový systém a kontejner obsahující samotnou aplikaci. Dále je připraven kontejner s nástrojem SonarQube. Nástroj SonarQube je potřeba při prvním spuštění nakonfigurovat pro práci s aplikací. Popis kontejnerů je v následující tabulce.

Název kontejneru	Popis kontejneru	Adresa v lokální síti
db	PostgreSQL databáze	`localhost:5432`
wallet_app	Serverová aplikace vystavující API	`localhost:8080`
sonarqube	Nástroj SonarQube	`localhost:9000`

Tabulka 6.1: Připravené kontejnery

Pro každý kontejner je připraven tzv. Dockerfile, tedy soubor obsahující příkazy nutné ke spuštění kontejneru. Součástí tohoto souboru je jaký obraz se používá v kontejneru a další nutné příkazy nastavující kontejner. V nastavení databázového kontejneru se například dají nastavit přihlašovací údaje a zkopírovat SQL skripty, které se spustí pří spuštění kontejneru.

```
FROM postgres:latest

ENV POSTGRES_PASSWORD 1234
ENV POSTGRES_USER postgres
ENV POSTGRES_DB wallet

COPY create_script_app.sql /docker-entrypoint-initdb.d/
COPY create_script_oauth.sql /docker-entrypoint-initdb.d/
```

Ukázka kódu 6.1: Ukázka databázového Dockerfile

Aby nemusel být každý kontejner spouštěn příkazem, je spuštění zajištěno pomocí nástroje `Docker Compose`. Nastavení jednotlivých kontejnerů je v souboru `docker-compose.yml` a pro spuštění celé aplikace se pak volá příkaz `docker-compose up`. Ukázka takového souboru je v další ukázce.

```
version: '2'
services:
  db:
    build: ./docker/postgres
    ports:
      - "5432:5432"
  sonarqube:
    build: ./docker/sonarqube
    ports:
      - "9000:9000"
  wallet_app:
    build: .
    container_name: wallet_app
    links:
      - db:db
    ports:
      - "8080:8080"
    depends_on:
      - db
    tty: true
```

Ukázka kódu 6.2: Ukázka docker-compose

V souboru se u jednotlivých kontejnerů nastavuje umístění jejich `Dockerfile`, případně na kterých dalších kontejnerech závisí a jaké vystavují porty a další parametry. U kontejneru `wallet_app` (kontejner obsahující serverovou aplikaci) je nastaveno, že jeho spuštění závisí na kontejneru `db` (kontejner s databází), protože aplikace nemůže běžet bez spuštěné databáze.

Při použití nástroje `Docker Compose` je mezi jednotlivými kontejnery vytvořena síť, kde se změní jméno sítě kontejneru z `localhost` na jméno kontejneru, nebo na nastavené v parametru `links`. Pokud je tedy potřeba se z kontejneru `wallet_app` připojit ke kontejneru `db`, nebude se kontejner připojovat přes adresu `localhost:5432`, ale přes adresu `db:5432`.

Pro jednodušší spuštění aplikace je připraven skript v jazyce Bash. Pro sestavení a spuštění aplikace je připraven skript `buildAndRun.sh`. Sestavení aplikace probíhá pomocí nástroje Gradle. Pomocí tohoto skriptu je spuštěn pouze kontejner s databází a aplikací. Po spuštění je aplikace připravena přijímat klientské požadavky na API. Pokud je potřeba zavolat spuštění všech kontejnerů (například kontejner s nástrojem SonarQube), je potřeba zavolat příkaz `docker-compose up`.

6.2 Android aplikace

Pro sestavení Android aplikace je využit nástroj Gradle. Pro sestavení aplikace je potřeba zavolat z `root` složky aplikace příkaz `gradle build`. Po dokončení

tohoto příkazu jsou ve složce `app/build/outputs/apk/` vygenerovány soubory typu `apk`. Tyto soubory je možné instalovat do Android zařízení. Před instalací aplikace v zařízení je potřeba povolit v nastavení instalaci aplikací třetích stran.

Android aplikace má nastavenou adresu serverové aplikace lokální sítě. Pokud je potřeba použít jinou síť, musí se adresa nastavit v kódu a znovu musí proběhnout sestavení a instalace aplikace.

Před spuštěním mobilní aplikace musí být spuštěná serverová aplikace.

Závěr

Cílem práce bylo vytvořit serverovou aplikaci a mobilního klienta pro správu společných výdajů napsané v jazyce Kotlin. V práci jsou popsány možnosti použití jazyka Kotlin při vývoji aplikací pro Android a při vývoji aplikací používajících framework Spring. Práce dále obsahuje analýzu, návrh a postupy při implementaci jak mobilní, tak serverové aplikace.

Vytvořená aplikace umožňuje spravovat společné výdaje. Obsahuje možnosti mít rozdělené výdaje do několika tzv. peněženek a v těchto peněženkách, pak mohou uživatelé přidávat a odebírat transakce. Uživatelé pak mají přehled, kolik v jednotlivých peněženkách dluží ostatním uživatelům. Transakce si uživatelé mohou rozdělit do kategorií.

V budoucnosti je možné aplikaci rozšířit o další funkcionality, například propojení aplikace s platebním systémem třetí strany a možností uskutečňovat platby mezi uživateli přímo v aplikaci.

Literatura

[1] Understand the Activity Lifecycle, Android Developers. [Online], [cit. 2019-04-27]. Dostupné z: https://developer.android.com/guide/components/activities/activity-lifecycle

[2] JetBrains readies JVM-based language - InfoWorld. [Online], [cit. 2019-04-27]. Dostupné z: https://www.infoworld.com/article/2622405/jetbrains-readies-jvm-based-language.html

[3] Reference - Kotlin Programming Language. [Online], [cit. 2019-04-20]. Dostupné z: https://kotlinlang.org/docs/reference/

[4] Unchecked Exceptions — The Controversy. [Online], [cit. 2019-04-20]. Dostupné z: https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

[5] Exceptions - Kotlin Programming Language. [Online], [cit. 2019-04-20]. Dostupné z: https://kotlinlang.org/docs/reference/exceptions.html

[6] Static Fields - Calling Kotlin from Java - Kotlin Programming Language. [Online], [cit. 2019-04-20]. Dostupné z: https://kotlinlang.org/docs/reference/java-to-kotlin-interop.html

[7] Null Safety - Kotlin Programming Language. [Online], [cit. 2019-04-20]. Dostupné z: https://kotlinlang.org/docs/reference/null-safety.html

[8] Extensions - Kotlin Programming Language. [Online], [cit. 2019-04-20]. Dostupné z: https://kotlinlang.org/docs/reference/extensions.html

[9] Type Checks and Casts: 'is' and 'as' - Kotlin Programming Language. [Online], [cit. 2019-04-20]. Dostupné z: https://kotlinlang.org/docs/reference/null-safety.html

[10] Introducing Kotlin support in Spring Framework 5.0. [Online], [cit. 2019-04-21]. Dostupné z: https://spring.io/blog/2017/01/04/introducing-kotlin-support-in-spring-framework-5-0

[11] Android KTX, Android Developers. [Online], [cit. 2019-04-27]. Dostupné z: https://developer.android.com/kotlin/ktx

[12] RebelLabs Developer Productivity Report 2017: Why do you use the Java tools you use? - RebelLabs. [Online], [cit. 2019-04-30]. Dostupné z: https://jrebel.com/rebellabs/developer-productivity-report-2017-why-do-you-use-java-tools-you-use/

[13] Version Control Systems Popularity in 2016 - RhodeCode. [Online], [cit. 2019-04-30]. Dostupné z: https://rhodecode.com/insights/version-control-systems-2016

[14] Patel, N.: *Spring 5.0 Projects: Build seven web development projects with Spring MVC, Angular 6, JHipster, WebFlux, and Spring Boot 2*. Packt Publishing, 2019, ISBN 9781788391979.

[15] paxdiablo: What algorithm to use to determine minimum number of actions required to get the system to "Zero" state? StackOverflow, [cit. 2019-04-09]. Dostupné z: https://stackoverflow.com/q/877728/

[16] All unique triplets that sum up to a given value. GeeksForGeeks, [cit. 2019-04-10]. Dostupné z: https://www.geeksforgeeks.org/unique-triplets-sum-given-value/

Příloha **A**

Seznam použitých zkratek

GUI Graphical user interface

XML Extensible markup language

JVM Java Virtual Machine

POJO Plain Old Java Object

CSRF Cross Site Request Forgery

MVC Model View Controller

IDE Integrated Development Environment

XSS Cross Site Scripting

REST Representational State Transfer

DAO Database Access Object

DTO Data Transfer Object

HTTP Hypertext Transfer Protocol

CRUD Create Read Update Delete

Příloha **B**

Seznam použitých pojmů

Boiler-plate kód Sekce kódu, které se dají použít na vícero místech s tím, že se téměř nebo vůbec nemění

Endpoint Koncový bod, místo které slouží ke komunikaci klienta se serverem. Každý endpoint je identifikován svojí unikátní síťovou adresou.

Interceptor Místo, ve kterém dochází k úpravě příchozího, nebo odchozího HTTP požadavku. Běžně se využívá k logování, přidání autorizačních parametrů, nebo úpravě hlaviček požadavku.

CRUD operace Shrnuje základní operace nad záznamem v úložišti – Create, Read, Update, Delete. Tedy vytvoření, čtení, editování a mazání záznamu.

Příloha C

Obsah přiloženého USB

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF