FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	iOS aplikace pro správu zaměstnanců
Student:	Bc. Michal Sousedík
Vedoucí:	Ing. Martin Půlpitel
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Cílem práce je navrhnout a implementovat mobilní aplikaci pro správu zaměstnanců v malé a střední firmě.

1. Analyzujte zaměstnanecké procesy v malé a střední firmě.
2. Prozkoumejte zaměstnanecký systém ve firmě Ackee s.r.o.
3. Analyzujte dostupné API a případně navrhněte doplnění.
4. Navrhněte a implementujte mobilní aplikaci.
5. Aplikaci otestujte.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 18. prosince 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

iOS aplikace pro správu zaměstnanců

Bc. Michal Sousedík

Katedra softwarového inženýrství

Vedoucí práce: Ing. Martin Půlpitel

7. ledna 2021

Poděkování

Rád bych zde poděkoval svému vedoucímu diplomové práce Ing. Martinu Půlpitlovi za rady a pomoc při realizaci tohoto projektu. Dále bych rád poděkoval Štefanu Prokopovi, který mě seznámil se serverovou částí projektu a mé přítelkyni, která mi byla velkou oporou.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při 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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 7. ledna 2021 . . . . . .

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

Sousedík, Michal. *iOS aplikace pro správu zaměstnanců.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato diplomová práce se zabývá tvorbou mobilní aplikace pro platformu iOS, která slouží k usnadnění zaměstnaneckých procesů v malé či střední společnosti. Všem zaměstnancům společnosti poskytuje možnost kontroly jejich osobních údajů, jež jsou zaměstnavatelem evidovány a dále umožňuje externím pracovníkům vykazovat faktury za uplynulé měsíce. Důležitou součástí aplikace je správa zaměstnaneckých profilů a všech vykázaných faktur, která je dostupná pouze adminovi aplikace.

Práce obsahuje detailní postup návrhu uživatelského rozhraní společně se zpracovanou architekturou, která je založena na analýze možných řešení a dostupných technologií pro iOS. Velký důraz je kladen na testovací praktiky, kterými se ověřuje správnost implementovaných funkcí. Výsledkem je aplikace, která vznikla realizací vytvořeného návrhu.

Klíčová slova aplikace, zaměstnanec, faktura, Swift, MVVM, iOS

Abstract

The main purpose of this master's thesis is to develop an iOS application that makes employees's administration processes inside a small or medium-sized company easier. One of the essential features is uploading a monthly invoice by an external contractor. All employees will be provided with access to their personal information, which is being stored by their employer. Application admins are authorized to manage any employee's invoices or their profiles.

The thesis consists of a procedure used when designing user interface and description of the system's architecture that is based on a thorough analysis of available solutions and frameworks commonly used by the iOS community. Emphasis is placed on testing which took a vital part during the development phase. The result of this thesis is a fully functional application based on the designed architecture and UI.

Keywords application, employee, invoice, Swift, MVVM, iOS

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Úvod

Která společnost si v dnešní době nepotřebuje vést zevrubné informace o svých zaměstnancích? Pro každodenní chod moderní firmy je takřka nezbytné, aby veškeré zaměstnanecké údaje byly trvale a bezpečně uloženy na dostupném místě, kde s nimi lze jednoduše manipulovat. K dispozici je celá řada aplikací, které slouží právě k tomuto účelu. Zaměstnáváte-li však větší množství osob, určitě se vyplatí investovat do vlastní aplikace, která vyhovuje vašim interně nastaveným procesům. Společnost Ackee v tom má jasno, a proto pracuje na vývoji webového portálu, který slouží ke správě zaměstnaneckých procesů, jež bude na míru odpovídat jejich potřebám. Pro vetší flexibilitu se tvůrci tohoto projektu rozhodli přidat k portálu i mobilní aplikaci.

Projekt mě zaujal díky své povaze reálného produktu, se kterým budou ve výsledku pracovat skuteční uživatelé. Současně si můžu na vlastní pěst vyzkoušet vývoj aplikace pro platformu iOS, k čemuž se již delší dobu odhodlávám. Problematika správy zaměstnanců je komplexní a může mě tedy lecčemu přiučit.

Mobilní aplikace se skládá ze tří částí rozdělených dle role aktuálního uživatele. První částí jsou funkcionality dostupné pro roli zaměstnanec, jehož primárním úkolem je koncem měsíce vykázat fakturu, kterou chce, aby mu společnost uhradila. Další způsoby použití jsou čistě informativního rázu. Zaměstnanec má k dispozici přehled faktur, které doposud vykázal, včetně jejich náhledu. Zobrazit si může svou aktuální hodinovou sazbu nebo osobní údaje. Vedoucí týmu je další z dostupných rolí, která se od role zaměstnanec liší pouze v možnosti zobrazit si seznam členů svého týmu včetně zaměstnaneckých profilů, které obsahují hodinovou sazbu a jejich osobní údaje. Naprosto odlišné funkcionality jsou poskytnuty pro roli admin, jejímž úkolem je správa ostatních zaměstnanců ve firmě. S tím zejména souvisí procesy spojené s kontrolou faktur zaměstnanců, úpravy jejich osobních údajů či nastavení nové hodinové

sazby.

V rámci této diplomové práce se zabývám návrhy uživatelského rozhraní a architektury, s jejichž pomocí implementuji analyzované případy užití, které jsou založeny na výše zmíněných funkcionalitách. Pro ověření správnosti řešení aplikaci otestuji jak automatizovanými, tak uživatelskými testy.

Kapitola 1

Cíl a metodika

1.1 Cíl práce

Cílem této diplomové práce je vytvoření mobilní aplikace pro platformu iOS, která ulehčí zaměstnanecké procesy ve firmě Ackee s.r.o. Aplikace bude podporovat tři druhy uživatelských rolí, kde každá má k dispozici svou množinu funkcí:

- **Zaměstnanec** – Zaměstnanci zobrazím přehled jeho dosavadně vykázaných faktur včetně relevantních informací jako je měsíc a rok, za který je faktura vystavena, aktuální stav a její výsledná částka. Současně zaměstnanci umožním stáhnout pdf reprezentaci faktury a nahrát novou pdf fakturu. Zaměstnanec se bude moci podívat na své uživatelské údaje včetně aktuální hodinové mzdy.
- **Team leader** – Role team leader obsahuje všechny funkce role user, které jsou rozšířené o přehled členů týmu, jehož je leaderem, včetně jejich profilů.
- **Admin** – Připravím kategorizovaný přehled faktur, vykázaných všemi zaměstnanci, dle jejich stavu a data. Admin si bude moci zobrazit a upravit profily všech zaměstnanců. Také mu bude umožněno nastavit zvolenému zaměstnanci novou hodinovou sazbu. Nezbytnou součástí bude také přehled aktuálního navýšení hodinových sazeb u zaměstnanců a přehled průměrného navýšení v rámci zvoleného měsíce.

Dalším cílem je umožnit uživateli přihlášení skrze jeho Google účet a případné odhlášení z aplikace.

Dílčí implementační části aplikace budu validovat sadou automatizovaných jednotkových a UI testů. Design budu testovat za pomoci heuristických testů

a skupiny dobrovolníků, kteří na sebe vezmou role cílových uživatelů a zhodnotí UX aplikace.

1.2 Metodika

Práci si rozdělím do následujících kapitol:

1. **Rešerše** – Představím společnost, která práci zadala a zmapuji technologie, které mohu použít při její implementaci.
2. **Analýza** – Zjistím aktuální stav správy zaměstnanců v zadavatelské firmě a v další vybrané menší až střední společnosti. Identifikuji seznam funkčních a nefunkčních požadavků kladených na výslednou aplikaci.
3. **Návrh** – Vytvořím a otestuji uživatelské rozhraní. Navrhnu architekturu cílového řešení, kterým se budou řídit všechny dílčí identifikované části. Odchylky od této architektury popíši v rámci sekce o identifikovaných funkcionalitách.
4. **Implementace** – Popíši zajímavosti, se kterými jsem se při psaní kódu setkal.
5. **Testování** – Ukáži praktiky, kterými jsem celou aplikaci testoval, abych minimalizoval pravděpodobnost výskytu chyb.

Diagramy vytvořím za pomoci aplikace Enterprise Architect a použiji modelovací jazyk UML 2.0. Aplikaci pro iOS budu vytvářet v XCode IDE, kde kód budu verzovat za pomoci VCS Git. S tvorbou prototypů mi pomůže aplikace Invision a s přípravou ikonek Sketch.

Kapitola 2

Rešerše

2.1 Představení zadavatelské společnosti

Ackee s.r.o je agenturou pro vývoj mobilních a webových aplikací se sídlem v pražském Karlíně, která se specializuje na technologicky náročnější projekty. Mezi její klienty patří Dámejídlo, Equa Bank, Košík.cz a další. Pravidelně publikují blogy o technologiích i o aktuálním dění z jejich firmy. Mají své vlastní open source knihovny, které může kdokoliv začlenit a použít ve svém projektu.

Společnost byla založena v roce 2012 v rámci ČVUT inkubátoru a aktuálně (prosinec 2020) má přes 70 zaměstnanců.

2.2 Možnosti vývoje

Vytvořit aplikaci použitelnou na iOS jde hned třemi způsoby. V této kapitole popisuji dostupné možnosti a uvedu jejich hlavní výhody a nevýhody.

Nativní přístup Přístup, jenž předpokládá použití nativního vývojového jazyka. Pro operační systém iOS se jedná o Swift nebo Objective-C. Takto vytvořenou aplikaci nelze použít na jiných operačních systémech, jako je např. Android. Zato má ovšem vývojář přístup k veškerým senzorům a systémům mobilního zařízení. Xcode IDE je vytvořen pro podporu nativního přístupu a usnadňuje mnoho aspektů vývoje, ať už se jedná o vytvoření prvotní šablony nebo nahrání hotové aplikace do Apple Connect, kde dále putuje přes testování a App Store až do telefonu cílového uživatele.

Mobilní webová aplikace Toto řešení počítá s vytvořením responzivní webové aplikace, kterou si uživatel může otevřít pomocí prohlížeče. Odpadá tedy proces nahrávání a schvalování aplikace na Apple Store, který může oddálit na-

sazení do produkce, a tak teoreticky připravit o zisk. Velkou výhodou tohoto řešení je její dostupnost pro všechny typy uživatelů, neboť ať už uživatel používá mobil s OS Android, iOS nebo je na svém počítači, může aplikaci otevřít a pracovat s ní. Problém může nastat při snaze dodržovat HIG, neboť nelze customizovat uživatelské rozhraní dle platformy a očekávání jejich uživatelů. Při tvorbě složitějších aplikací může vyvstat potíž s výkonem, neboť se spoléhá na prohlížeč. Což s sebou přináší otázku navigace v rámci aplikace, neboť prohlížeč podporuje tlačítko pro přechod na předchozí stránku, jež nemusí být v aplikaci chtěné.

Hybridní přístup Pokud je cílem vytvořit aplikaci, která se bude moci použít napříč platformami a není dostatek času, zvolením tohoto přístupu lze ušetřit značné množství zdrojů. Jak už název naznačuje, jedná se o kombinaci výše zmíněných přístupů. Jádro aplikace je napsáno ve webových technologiích, ale v případě iOS je prezentováno z nativní aplikace prostřednictvím komponenty `WKWebView` [1], která vykresluje DOM elementy. Tento přístup řeší většinu výše zmíněných problémů. Vytváří se pouze jedno implementační řešení, které je společné napříč platformami, čímž se minimalizuje riziko výskytu chyb a potřeba znalosti několika programovacích jazyků. Vytvářet hybridní kód, však může být značně netriviální. Správná funkce a výkonnost operace na iOS nezaručuje stejný výsledek v OS Android a vice versa. Zejména s výkonem může vyvstat spousta potíží. Ty se snaží do jisté míry řešit knihovny jako Ionic, Xamarin, Flutter, React Native apod. pomocí optimalizovaných komponent.

2.3 Programovací jazyk

Spektrum programovacích jazyků, které lze použít, se značně omezí výběrem nativního vývoje, kterým budu pokračovat v implementaci aplikace. Tuto volbu jsem učinil triviálně již v tomto kroku, abych se v této kapitole mohl věnovat jen relevantním programovacím jazykům a nezabýval se možnostmi, které s sebou přináší hybridní a webové aplikace. Učinil jsem tak, jelikož výsledkem této práce má být aplikace pouze pro iOS.

Výběr programovacího jazyka pro nativní vývoj se k roku 2020 velmi zjednodušil. Přesto stojí za to, zvážit následující možnosti.

Objective-C Spojením zpráv zasílajícího objektově orientovaného Smalltalku a jazyka C, vznikl v roce 1984 programovací jazyk Objective-C. Během dosavadní existence tohoto jazyka v něm bylo vytvořeno nespočet veřejně dostupných knihoven a velké množství návodů. Jazyk si také vybudoval silnou komunitu vývojářů, která dokáže rychle zareagovat a poradit, pokud si nováček neví rady. Za skoro 40 let své existence se objevily a opravily snad veškeré možné chyby a tedy aktuální verze by měla být maximálně stabilní. [2]

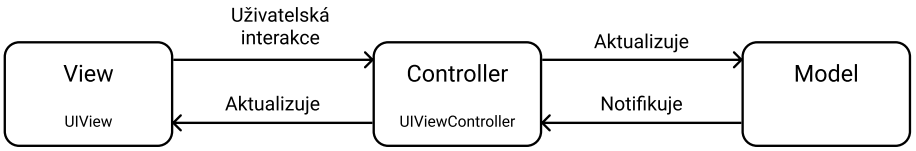
Swift Rok 2014 s sebou přinesl vylepšení v podobě funkcionálního jazyka Swift, jenž vznikl jako projekt společnosti Apple. Velkou výhodou jazyka Swift je koncept optionals, který minimalizuje riziko vzniku NPE tím, že nutí vývojáře přemýšlet nad otázkou „Co když tuto hodnotu nebudu mít?“ Swift je skoro 2.6x rychlejší [3] nežli Objective-C. Díky jeho rychle rostoucí popularitě Apple vyvinul SwiftUI, jenž je nástupcem knihovny UIKit. Dá se tedy předpokládat, že Swift je budoucností vývoje mobilních aplikací pro iOS.

2.4 Architektonický styl

Správně zvoleným architektonickým stylem, kterým lze strukturovat kód, si mohu usnadnit jak samotnou implementaci, tak i následnou údržbu. V této kapitole analyzuji zaběhlé styly globálně používané při vývoji iOS aplikací a porovnám je dle jejich testovatelnosti, udržitelnosti a množství potřebného kódu.

2.4.1 MVC

MVC bylo zvoleno společností Apple jako oficiální architektura pro iOS aplikace. Každá třída odpovídá buďto Modelu, View nebo Controlleru.



Obrázek 2.1: Diagram komunikace komponent architektury MVC

Úkolem **Modelu** je starat se o data, jež odpovídají aktuálnímu stavu aplikace. Modelem může být například reálný objekt z okolního světa.

Za **View** se považuje vše, co je zobrazeno uživateli. Příkladem může být tlačítko, textové pole nebo obrázek. Tyto objekty vytvářejí UI aplikace, jedná se tedy zejména o potomky třídy UIView.

Controller je řídícím prvkem celé aplikace. Instance controlleru konfiguruje zobrazené view a zajišťuje, že view a model jsou synchronizovány. Jedná se o potomky třídy `UIViewController` [4].

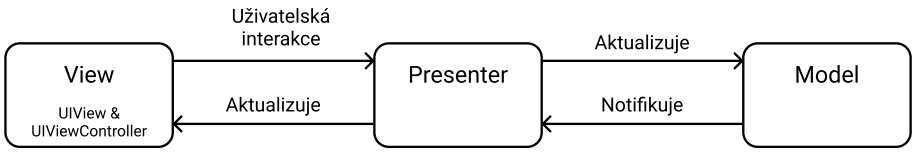
Rozdělením aplikace do vrstev zodpovědnosti se zvyšuje udržitelnost aplikace. Model i view lze použít současně v různých controllerech nebo dokonce v různých projektech. Díky jejich oddělitelnosi od celku jsou snáze testovatelné. S controllerem, jeho recyklací a testováním to bývá složitější. Je tomu

tak díky jeho životnímu cyklu a unikátním pravidlům, jež implementuje pro zvolenou doménu.

Otázkou je, ve které vrstvě implementovat business logiku, komunikaci se serverem nebo perzistenci dat. Dle definice by výše zmíněné mělo být implementováno v controlleru, nicméně díky jeho horší testovatelností, jiné zdroje [5] uvádějí, že se tuto logiku vyplatí implementovat v modelu. Výsledkem implementování veškeré logiky v controlleru jsou tisíciřádkové třídy, ve kterých bývá velmi obtížné se vyznat. Zároveň většina metod těchto tříd má vedlejší účinky, díky nimž je náročné psát znovupoužitelný kód a pokrývat ho testy. Tento styl může být vhodný pro menší aplikace, u kterých je jistota, že se nebude do budoucna rozšiřovat a která se musí dostat do produkce v co možná nejkratším čase.

2.4.2 MVP

„Model, View, Presenter“ stejně jako MVC rozděluje aplikaci na 3 logické části, avšak oproti MVC nepovažuje potomka třídy `UIViewController` za místo pro správu business logiky, nýbrž jako konstrukci obsahující kód pro konfiguraci pouze uživatelského rozhraní [6]. Tímto je řešeno dilema ohledně zodpovědnosti za business logiku. Instance spadající do vrstvy presenter již není závislá na životních cyklech UI komponent a dá se tak snadněji recyklovat a testovat.

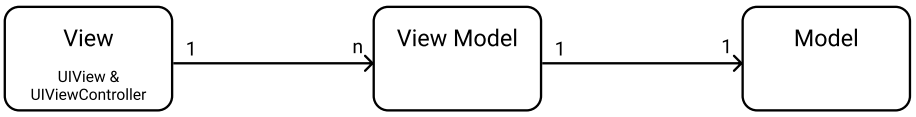


Obrázek 2.2: Diagram komunikace komponent architektury MVP

Vytváření a správa presenteru ovšem potřebuje vcelku velké množství dodatečného kódu pro přeposílání všech uživatelských akcí z view.

2.4.3 MVVM

Architektonický styl, který má za cíl rozšířit možnosti dekompozice aplikace. Princip je podobný jako u MVP. Viewmodel získává informace o interakcích uživatele od view, upraví data získaná z modelu a transformovaná data vrací zpět view, které předpřipravená data pouze zobrazí. Rozdílem je, že view může obsahovat více instancí různých viewmodelů. Tento drobný rozdíl umožňuje intuitivnější rozdělení kódu do logických celků a tím podporuje princip jedné odpovědnosti (SRP).



Obrázek 2.3: Diagram závislostí a kardinalit mezi komponentami architektury MVVM

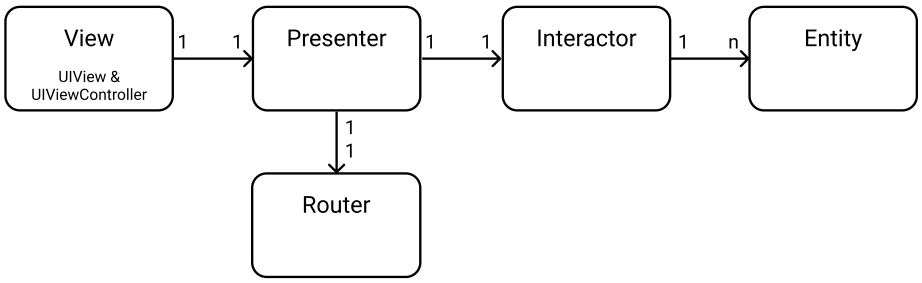
Množství kódu potřebného pro jeho implementaci je srovnatelné s MVP, nicméně se od vývojáře očekává větší rozvaha a důvtip při návrhu aplikace.

2.4.4 VIPER

VIPER se snaží vyřešit „massive viewcontroller“ problém a zlepšit testovatelnost kódu. Skládá se z 5 částí:

- **View** zobrazuje data poskytnutá presenterem a posílá mu zpět akce provedené uživatelem,
- **Interactor** obsahuje business logiku definovanou případem užití,
- **Presenter** připravuje data získaná z interactoru pro zobrazení a reaguje na uživatelské akce,
- **Entity** obsahuje modelová data, která zpracovává interactor,
- **Router** slouží k definování navigace mezi obrazovkami,

Na rozdíl od MVVM separuje logiku spojenou s transformací dat pro prezentační vrstvu (Presenter) a logiku datového modelu (Interactor). Struktura je založena na případech užití, čímž transparentně odráží business analýzu projektu [7]. Nevýhodou je množství souborů, které musí vývojář vytvořit, aby tento styl aplikoval.



Obrázek 2.4: Diagram závislostí a kardinalit mezi komponentami architektury VIPER

2.5 Programovací paradigma

Zvolit programovací jazyk a architektonický styl nestačí. Do takzvané velké trojky patří ještě jedna ingredience a tou je paradigma. Přesná definice je velmi vágní a popisuje programovací paradigma jako všeobecný přístup k psaní kódu [8], který není závislý na žádném konkrétním jazyku ani architektuře. Programovací paradigmata relevantní pro vývoj na platformu iOS jsou následující.

2.5.1 Objektové

Paradigma postavené kolem modelování objektů a jejich vzájemné komunikace prostřednictvím zpráv [8]. Každý objekt má svůj vnitřní stav a veřejné rozhraní, kterým definuje způsob, jakým s ním mohou ostatní objekty komunikovat. Po správném zapouzdření lze vnitřní stav objektu modifikovat pouze pomocí jeho metod, které se zavolají v reakci na obdrženou zprávu od jiného objektu.

2.5.2 Funkcionální

Koncept, který modeluje aplikaci pomocí „pure“ funkcí, což jsou funkce, které pro stejný vstup vrátí vždy stejný výstup a nemají vedlejší efekty [9].

2.5.3 Programování orientované na protokol

Protokolové paradigma lze vnímat jako rozšíření objektově orientovaného přístupu, které radí nezačínat s třídou, nýbrž s protokolem [10]. OOP se potýká s problémem mnohonásobné dědičnosti tříd. Swift ani Objective-C neumožňují, aby třída dědila z více tříd. Při modelování každodenního života je však taková vlastnost běžná, a tak společně se Swiftem 2.0, Apple představil i koncept protokolů. Protokol deklaruje funkce, které by měla implementující třída, struktura

nebo výčtový typ definovat a v případě potřeby jim poskytuje i jejich defaultní implementaci. Díky schopnosti objektu implementovat více protokolů, má vývojář lepší možnosti pro implementaci konceptů okolního světa.

Paradigma zlepšuje testovatelnost aplikace tím, že radí deklarovat atributy typem protokolu, čímž umožňuje jednodušší mockování vztahů mezi objekty.

2.5.4 Reaktivní

Při reaktivním programování se pracuje s asynchronními toky dat (stream) a událostmi. Tok událostí může být například:

- vstup z klávesnice,
- interakce uživatele s GUI aplikace,
- aktualizace GPS lokace.

Takový tok událostí lze pozorovat a podle potřeby na něj reagovat [9].

2.5.5 FRP

FRP je kombinace funkčního a reaktivního programování. Následováním tohoto typu programování, aplikace reaguje na tok událostí dle funkcionálního paradigma. Stavebními bloky jsou:

- **Tok události (Event stream)** – sekvence událostí během času, asynchronní pole;
- **Vnitří stav (State)** – aplikace se vždy nachází v určitém stavu, který je dán vstupními daty;
- **Vedlejší efekt (Side effect)** – zavoláním funkce, která mimoto, že vrátí výsledek tak modifikuje okolní prostředí aplikace (například zavolání funkce, která sčítá dvě čísla, vrací výsledek a zároveň výsledek uloží do databáze), vzniká vedlejší efekt;
- **Immutable object (Nezměnitelný objekt)** – objekt, který je immutable, nemůže být během své existence změněn.

2.6 FRP Knihovny

Knihoven, které podporují funkcionálně reaktivní programování pro iOS bylo za posledních 5 let napsáno několik. V této kapitole analyzuji nejznámější z nich a popíši jejich fundamentální prvky.

2.6.1 RxSwift

RxSwift je jedním ze členů rodiny ReactiveX. ReactiveX definuje principy a operátory, které potomci následují napříč různými jazyky, což umožňuje vývojáři rychlejší adaptaci, pokud přichází z prostředí RxJs, RxJava nebo jiného „RxSourozence".

Observable `Observable` je základním stavebním kamenem RxSwift. Jedná se o asynchronní sekvenci emitující události, které lze pozorovat. `Observable` sekvence je ukončena, emituje-li ukončující nebo chybovou událost [11].

Observer Pozorovatel, který se přihlásí k odběru událostí `Observable` sekvence a dle potřeby na ni reaguje.

Single Nádstavba třídy `Observable`, která vždy emituje buďto jeden element nebo chybu. Používá se zejména při volání HTTP požadavků, které vracejí buďto odpověď nebo chybu.

```
Single.create { single in
    let result = ExternalSource.calculateResult()
    if result.state == .success {
        single(.success(result.value))
    } else {
        single(.error(result.error)
    }
    return Disposables.create()
}
```

Kód 2.1: Příklad použití třídy Single

Completable `Completable` se chová podobně jako Single s tím rozdílem, že neemituje element, ale pouze ukončující nebo chybovou událost. Completable se dá použít například pří ukládání dat do databáze, kde závěrem má být pouze informace o tom, zdali byl proces dokončen úspěšně nebo nastala chyba.

Maybe `Maybe` je variací `Observable`, která svým chováním patří mezi `Single` a `Completable`. `Maybe` buďto:

- emituje pouze jeden element;
- skončí, aniž by emitovala jakýkoliv element;
- emituje chybovou událost.

Subject `Subject` se charakterizuje jako obojí `Observable` i `Observer` (pozorovatel). Může obdržet událost a zároveň může být pozorován jinou komponentou. Obdrženou událost `Subject` bezprostředně emituje dál svým pozorovatelům.

- **PublishSubject** se inicializuje s prázdnou počáteční hodnotou a emituje událost pro své odběratele teprve až některou obdrží.
- **BehaviourSubject** se inicializuje s definovanou počáteční hodnotou. Na rozdíl od `PublishSubject` po přihlášení nového pozorovatele, emituje poslední známou událost.

`PublishRelay` a `BehaviourRelay` jsou analogiemi zmíněných `PublishSubject` a `BehaviorSubject`. Relay objekty však nikdy neemitují chybu nebo ukončující událost [11].

Driver Třída, která umožňuje intuitivní práci s reaktivním paradigma v UI vrstvě [11]. Driver lze pomocí reaktivního rozšíření napojit přímo na UI elementy. Toto propojení funguje díky tomu, že `Driver` upozorňuje své pozorovatele o emitovaných událostech výhradně na hlavním vlákně, které je zodpovědné za správu UI komponent a tím řeší případné chyby, které vznikají, snaží-li se vývojář modifikovat UI element z jiného nežli hlavního vlákna. Další vlastností `Driveru` je, že nikdy neemituje chybovou událost, která by pro UI komponenty znamenala nedeterministickou situaci.

2.6.2 ReactiveSwift

ReactiveSwift je další knihovna, která nabízí účinné nástroje pro práci s toky dat. Ačkoli se ReactiveSwift konceptuálně inspiroval u Reactive Extensions, tak se na rozdíl od RxSwift nejedná o implementaci ReactiveX API, nýbrž o implementaci funkcionálně reaktivního paradigmatu.

Event Jedná se o formální reprezentací faktu, že se něco událo. Může reprezentovat například stisknutí tlačítka nebo příchozí data ze serveru.

Signal `Signal` reprezentuje tok událostí (`Event`) v čase, který lze pozorovat. Pozorovatel může tento tok sledovat a reagovat na získaná fakta, ale nemůže nijak změnit jeho průběh. Oficiální dokumentace [12] přirovnává `Signal` k televiznímu vysílání, které divák může sledovat, ale nemůže jeho obsah nijak ovlivnit.

SignalProducer Jak už název napovídá, `SignalProducer` vytváří signály. Definuje operaci, která začne produkovat tok události v momentě, kdy o ně bude mít pozorovatel zájem. Analogií může být streamovací služba, ve které má uživatel možnost zvolit si jaký seriál má zájem sledovat, přetočit jeho úvodní znělku nebo seriál vypnout [12].

Property Objekt, který obsahuje potencionálně měnící se hodnotu, jejíž změny lze pozorovat. Podobá se třídě `Signal` s tím rozdílem, že vždy vrací poslední známou hodnotu.

	RxSwift	ReactiveSwift
Ukončení datových toků	Pomocí objektu `DisposeBag`	Použitím operátoru $<\sim$
Řešení Hot[1] a Cold[2] sekvencí	Netransparentní, pomocí `Observable`	Rozdělení na `Signal` a `SignalProducer`
Typ emitované chyby	Chyba musí implementovat `ErrorType` protokol	Typ chyby se definuje při vytváření Signálu
Dokumentace	Detailní	Ktrátká a výstižná
Velké množství dostupných návodů	Ano	Ne

Tabulka 2.1: Výsledky porovnání základních vlastností RxSwift a ReactiveSwift

2.7 Komunikace se serverem

Způsob, kterým aplikace komunikuje se serverem, je jedním ze základních pilířů vývoje, jenž je potřeba analyzovat a začlenit do projektu. Swift komunita přišla s řadou řešení, které lze pro tento účel použít.

2.7.1 URLSession

`URLSession` je součástí Foundation knihovny, která tvoří jádro jazyka Swift. Poskytuje nástroje potřebné pro síťovou komunikaci. Jedná se o třídu zodpovědnou za vytváření HTTP a HTTPS požadavků[13]. Pro různé druhy síťové komunikace vytváří úkoly v podobě implementací třídy `URLSessionTask`:

[1]Hot - datový tok nezávislý na pozorovateli, který začíná emitovat události ihned po svém vytvoření.

[2]Cold - sekvence, která začne produkovat události až v momentu přihlášení pozorovatele. S každým novým pozorovatelem se vytváří nová sekvence, což znamená, že každý pozorovatel vždy získá celou sekvenci událostí.

- **URLSessionDataTask** se používá k zasílání/získávání menšího množství dat, neboť ukládá příchozí data do paměti.
- **URLSessionUploadTask** slouží k nahrání souboru na webový server prostřednictví HTTP metod POST a PUT.
- **URLSessionDownloadTask** má za úkol stáhnout soubor ze serveru na disk používaného zařízení.

Úkol (`URLSessionTask`) se po vytvoření nachází v suspendovaném stavu a pro jeho spuštění je zapotřebí na něm zavolat metodu `resume()`. Úkol lze zastavit, spustit nebo zrušit. Po dokončení úkolu `URLSession` vrací výsledek, který může obsahovat buďto získaná data nebo chybu.

```
var dataTask: URLSessionDataTask?
if var url = URL(string: "https://httpbin.org/get") {
  var urlRequest = URLRequest(url: url)
  urlRequest.httpMethod = "GET"
  urlRequest.addValue("application/json",
                      forHTTPHeaderField: "Accept")
  dataTask =
    URLSession(configuration: .default).dataTask(with: url) {
      [weak self] data, response, error in
      if let error = error {
        self?.handleError(error)
      } else if
        let data = data,
        let response = response as? HTTPURLResponse,
        response.statusCode >= 200,
        response.statusCode <= 300 {
          self?.handleSuccess(data)
      }
    }
  dataTask?.resume()
}
```

Kód 2.2: Ukázka použití URLSession pro získání dat ze serveru

2.7.2 Alamofire

Knihovna Alamofire je založena na základě `URLSession`, jejíž koncept se snaží zlepšit tím, že:

- Vytváří rozhraní, které je pro vývojáře srozumitelnější.

- Poskytuje obsáhlý katalog často používaných funkcí, které lze jednoduše použít. Mezi tyto funkce patří například možnost zopakovat požadavek po prvotním neúspěchu nebo přidat interceptor, který bude zaštiťovat proces ověření uživatele.
- Redukuje množství kódu potřebného pro vykonání HTTP požadavku.

```
AF.request("https://httpbin.org/get")
.validate(statusCode: 200..<300)
.responseJSON { [weak self] response in
  guard response.result.isSuccess else {
      self?.handleError(response.result.error)
  }
  self?.handleSuccess(data)
}
```

Kód 2.3: Příklad použití knihovny Alamofire pro získání dat ze serveru

AF je globální reference na sdílenou singleton instanci `Session.default`. Obdobně jako u `URLSession` i `Session` deleguje úkoly na potomky abstraktní třídy `Request`, jejichž význam je popsán v tabulce 2.2.

	AF metoda	Zapouzdřuje
DataRequest	request	URLSessionDataTask
UploadRequest	upload	URLSessionUploadTask
DownloadRequest	download	URLSessionDownloadTask

Tabulka 2.2: Význam potomků třídy Request knihovny Alamofire

2.7.3 Moya

Moya je další ze série knihoven, které ulehčují síťovou komunikaci. Stejně jako je Alamofire postavený na `URLSession`, je Moya postavena na Alamofire, tzn. nedefinuje svou vlastní komunikační vrstvu, nýbrž volá logiku implementovanou knihovnou Alamofire. Vytváří abstraktní strukturu nad síťovou vrstvou, čímž zlepšuje její přehlednost. Zaměřuje se na typovou bezpečnost při volání HTTP požadavků, kterou si vynucuje definovanou strukturou tříd a výčtových typů.

Target Výčtový typ, implementující `TargetType` protokol [14]. Target definuje endpointy, které bude aplikace volat včetně jejich dynamických vlastností a parametrů, kterými jsou:

- `basePath` – adresa serveru obsahující dotazovaná data;

- `path` – umístění dotazovaných dat na serveru;
- `method` – HTTP metoda;
- `headers` – pojmy, které mají být obsaženy v hlavičce požadavku;
- `task` – parametry požadavku;
- `sampleData` – příklad návratových dat, který se používá při testování aplikace;
- `validationType` – HTTP statusy kódů, které reprezentují úspěch provedené operace.

Provider Generická třída poskytována knihovnou Moya [14], která transformuje `Target` na `URLRequest`.

2.8 Tvorba uživatelského rozhraní

Má-li aplikace komunikovat s uživatelem, musí obsahovat rozhraní, které bude zobrazovat data a přijímat vstupy. Existuje několik způsobů, jak UI vytvořit. V této kapitole je analyzuji společně s jejich pozitivy i negativy.

2.8.1 Storyboard

Vizuální nástroj, pomocí kterého lze postupným přetahováním UI komponent na virtuální plátno vytvářet finální podoba UI. Velkou výhodou tohoto řešení je schopnost reagovat na jakékoli změny návrhu okamžitou aktualizací plátna, která vývojáři poskytuje rychlou zpětnou vazbu. Mimo vizuálních prvků lze vytvořit navigaci mezi obrazovkami, což přispívá i k rychlému prototypování a lepší orientaci ve vzniklém projektu. Další výhodou je podpora při definovaní pravidel technologie Auto Layout, která se stará o správné vykreslení komponent na různých typech mobilních zařízení.

Nevýhodou je, že některých složitějších konstrukcí nelze dosáhnout jinak nežli zásahem do kódu. Kvůli tomu se běžně stává, že část uživatelského rozhraní je vytvořena ve storyboardu, část v kódu a bývá značně netriviální řešit chyby, které touto praxí mohou vzniknout. Další nevýhodou je neschopnost recyklovat dříve vytvořené komponenty. U větších projektů může vzniknout taky problém s pomalejším vykreslením plátna, který lze řešit rozdělením storyboardu do více souborů.

2.8.2 Xib

Xib se vytváří stejně jako storyboard v Interface Builderu. Jedná se o starší koncept nežli výše zmíněný storyboard a na rozdíl od něj nepopisuje velké

vizuální celky a komunikaci mezi nimi, ale pouze jeden element. Příkladem může být potomek třídy `UIView` nebo `UIViewController`. Lze jej pokládat za dekomponovaný storyboard, což částečně řeší problémy s výkonem a znovupoužitelností komponent, které jsou vykoupeny ztrátou definice přechodů mezi obrazovkami a tím i globálním přehledem o „flow“ aplikace [15].

2.8.3 Vlastní kód

Interface Builder, který používají storyboard i xib, ve výsledku generuje kód, který si může vývojář napsat sám bez pomoci vizuálních nástrojů. Tento přístup řeší veškeré problémy se znovupoužitelností komponent [15], udržování UI jak v IB, tak v kódu, bezproblémový debuging nebo verzování XML souborů vytvořených IB. Nevýhodou je časová náročnost při tvorbě uživatelského rozhraní, potřeba pokročilejší znalosti UI komponent, principů jejich propojení a umístění.

2.9 Navigace mezi obrazovkami

Jak přecházet mezi `UIViewControllery`, které reprezentují obsah obrazovky zobrazené uživateli nebo alespoň její podmnožinu, nemá jedno ultimátní správné řešení. V této kapitole nastíním, jakými výhodami a nevýhodami identifikovaná řešení disponují.

2.9.1 Segue

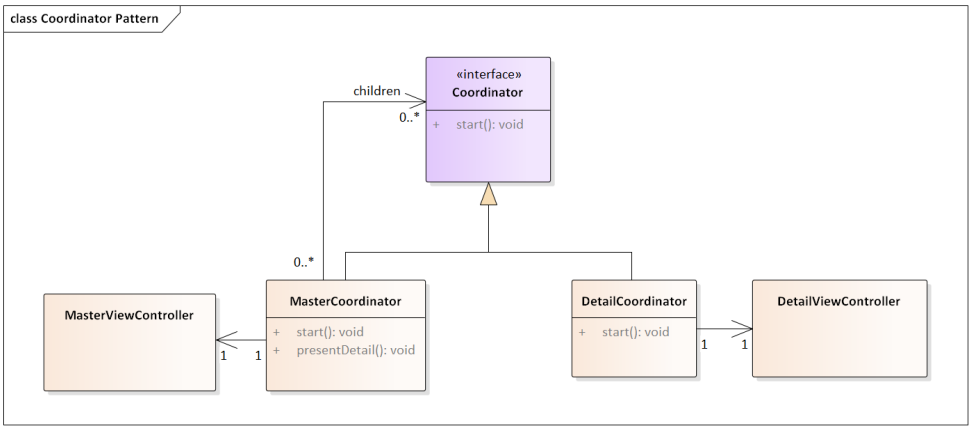
Segue je nástroj úzce spjatý a podporovaný storyboardy. Jedná se o pohodlný způsob, jak vizuálně definovat přechod mezi obrazovkami. Díky Segue máme k dispozici mapu přechodů v aplikaci a je tedy snadnější se v ní zorientovat. Platí to však pouze u menších aplikací, které nedisponují komplexní navigaci. U větších aplikací se storyboard stává nepřehledným kvůli velkému množství přechodů. Segue a metoda `prepare(for:sender:)`, díky které lze přidat přechodům logiku (například předat data jinému viewcontrolleru[3]), dělá viewcontrollery mezi sebou závislé a tím nerecyklovatelné.

2.9.2 Coordinator pattern

V roce 2015 Soroush Khanlou představil na svém blogu teoretický návrh vzoru Coordinator. Jeho hlavním cílem bylo delegovat část logiky, která se zabývá přechody a inicializací viewcontrollerů, do objektů s názvem Coordinator. Použitím tohoto vzoru se viewcontrollery stávají na sobě nezávislé a tak recyklovatelné a testovatelnější.

[3] viewcontrollerem je myšlen potomek třídy UIViewController

Soroush identifikoval Coordinator jako protocol s funkcí `start()`, jejímž úkolem je zobrazit relevantní `UIViewController` [16].



Obrázek 2.5: Diagram tříd - příklad implementace vzoru Coordinator

Je-li potřeba přejít z `MasterViewControlleru` do `DetailViewControlleru`, musí `MasterCoordinator` vytvořit instanci třídy `DetailCoordinator`, zavolat na ní metodu `start()` a uložit ji do lokální proměnné, aby nedošlo k její dealokaci. To s sebou přináší problém v podobě „memory leaků“, neboť nemusí být jednoduché zjistit, že „child“ koordinátor již není aktivní a má se tedy odebrat z pole. Za předpokladu, že je použit `UINavigationController` jakožto kontejner, pomocí kterého se zobrazují viewcontrollery, stačí aby třída `MasterCoordinator` implementovala protokol pro řízení událostí spojených s navigací (`UINavigationControllerDelegate`), a pokud v delegované metodě `navigationController(navigationController:didShow:animated:)` parametr `navigationController` ve své hierarchii neobsahuje instanci třídy `DetailViewController`, je možné `DetailCoordinator` odebrat z proměnné, a tak umožnit jeho dealokaci [17].

2.9.3 Flow Controller

Stejně jako vzor Coordinator, se i vzor Flow Controller potýká s navigací mezi viewcontrollery tak, že přenechává jejich správu jinému objektu. Zatímco vzor Coordinator delegoval navigaci objektu nezávislému na knihovně UIKit, vzor Flow Controller přenechává správu viewcontrollerů potomku třídy UIViewController [18].

`AppFlowController` v rámci metody `viewDidLoad()` inicializuje a zobrazí `MasterViewController`. Je-li potřeba přejít z `MasterViewController` na jeho detail `DetailViewController`, stačí aby `AppFlowController` vytvořil a zobrazil `DetailViewController`.

Flow Controller dědí životní cyklus své nadtřídy `UIViewController`. Tento životní cyklus automaticky řeší dealokaci Flow Controlleru a nevzniká tak problém se správou objektů v paměti jak tomu bylo u vzoru Coordinator 2.9.2.

2.10 OAuth 2.0

Protokol OAuth 2.0 slouží k autorizaci uživatele a umožňuje aplikacím třetí strany přístup k datům uživatele, aniž by znaly jeho přihlašovací údaje. Autorizace probíhá skrze bezpečnostní prvek známý jako access token. Protokol definuje 4 role [19]:

- **Resource Owner** – uživatel, který chce povolit klientské aplikaci přístup ke svým chráněným datům;
- **Client** – klientská aplikace, například mobilní iOS aplikace;
- **Resource Server** – server, na kterém se nachází data uživatele, ke kterým potřebuje klientská aplikace získat přístup;
- **Authorization Server** – server, který se stará o autorizaci uživatele.

Proces přístupu k chráněným datům uživatele lze popsat ve 3 krocích.

1. Workflow začíná, když klientská aplikace požádá autorizační server o autorizační kód. Požadavek obsahuje mimo jiné i rozsah práv, které je potřeba udělit. Uživateli se zobrazí webový prohlížeč, kde je dotázán, zdali chce požadované oprávnění aplikaci poskytnout. Za předpokladu, že uživatel oprávnění udělí, je klientské aplikaci zaslán autorizační kód.
2. Klientská aplikace zašle autorizační kód autorizačnímu serveru a výměnou dostane access token.
3. S pomocí access tokenu získává klientská aplikace od resource serveru chráněná data uživatele.

2.11 Testování

V této kapitole se zaměřím na způsoby testovaní aplikace pro platformu iOS.

2.11.1 Unit testy

Jednotkový test je metoda, která zavolá část produkčního kódu a zkontroluje, zdali je výsledek tohoto volání správný. Framework, který umožňuje definovat

jednotkové testy se nazývá XCTest. Testy, které mají za úkol kontrolu společné logiky se shlukují do tzn. modelových případů, což jsou ve výsledku potomci třídy `XCTestCase`.

GWT struktura Aby byly testy přehledné, měly by následovat jednotnou strukturu. Tělo testovací metody lze rozdělit do třech částí:

- **Given** – První fáze testu, ve které se definují objekty, které se budou následně testovat nebo budou tvořit závislosti testovaného objektu.
- **When** – Zavolání operace, jejíž výsledek se má validovat.
- **Then** – Otestování, zdali výsledek odpovídá očekáváním.

Části kódu, které jsou společné pro všechny testy v rámci modelového případu, lze definovat v metodě `setUp()`, která je volána před každou testovací metodou.

F.I.R.S.T. Testy by měly splňovat následující vlastnosti [20]:

- **Fast** – testy musí proběhnout rychle, jinak je vývojář nebude pouštět pravidelně;
- **Independent** – testy na sobě nesmí být závislé, nejčastějším porušením tohoto pravidla bývá test, který spoléhá, že stav testovaného objektu byl změněn předchozím testem;
- **Repeatable** – výsledky opakovaného volání testu se nesmí lišit v čase nebo napříč prostředími;
- **Self** – výsledkem testu je jednoznačná informace o tom, zdali test proběhl úspěšně či nikoli;
- **Timely** – testy jsou psány dříve než produkční kód.

Pojmenování Testovací metody musí začínat prefixem test, jinak je framework nenalezne a nespustí. Metoda testující scénář v rámci kterého se získávají aktuální hodinové mzdy, může znít například „testRetrievingCurrentHourlyRate“.

Testování asynchronních operací Asynchronní operace vrací návratovou hodnotu okamžitě, ale samotný výpočet může proběhnout mnohem později. Pro jejich testování tedy nebude stačit klasický assert. S validací výsledku se musí počkat až na dokončení asynchronní operace, s čím pomůže třída `XCTestExpectation`. Díky této třídě můžeme definovat hodnotu, kterou očekáváme, že operace v budoucnu vrátí a její validaci provést zavoláním metody `fullfill()`. V nevyhovujících scénářích se použije `XCTFail`.

Testování RxSwift Jak na asynchronní operace již vím, zbývá zjistit, jak naložit s reaktivním kódem. V podkapitole 2.5.4 jsem nastínil, že reaktivní svět se točí kolem streamů, které nedeterministicky emitují hodnoty v závislosti na čase. Pro jejich otestování je zapotřebí nástroj, který zjistí, že:

- stream emituje hodnoty v určitém pořadí,
- stream emituje hodnotu v konkrétním čase.

RxBlocking Pro otestování sekvence s konečným množstvím událostí je knihovna RxBlocking ideálním kandidátem, neboť dokáže zablokovat testovací vlákno, dokud sekvence neemituje chtěné události.

RxTest Je-li cílem získat informaci o čase, kdy se událost objevila, RxBlocking nestačí. K tomuto účelu slouží knihovna RxTest, která používá svůj vlastní plánovač `TestScheduler`. Díky tomuto plánovači je možné [21]:

- definovat časově závislé události, pomocí kterých můžeme mockovat interakci mezi komponentami;
- definovat, v jaký čas očekáváme, že se událost emituje.

2.11.2 UI testy

Pomocí UI testů lze simulovat a testovat interakce uživatele s UI. Uživatelské rozhraní se testuje zasíláním událostí nalezeným UI komponentám a následnou kontrolou jejich vlastností [22].

Xcode umožňuje nahrát uživatelskou interakci, kterou následně převede na odpovídající příkazy, které lze kdykoli znovu přehrát. Jediné, co je potřeba přidat do testovacích metod, je očekávaný stav aplikace po přehrání nahraných příkazů. Komunikuje-li mobilní aplikace se serverem, na jehož přesnou podobu zpětné vazby nelze spoléhat, je nutné takovou komunikaci nahradit mockem. Například knihovna OHHTTPStubs umožňuje definovat jaká data se mají vrátit při zavolání uvedeného požadavku. Pomocí knihovny lze testovat timeout požadavku, chybové stavy aj.

UI testy dokážou ušetřit velké množství času spojeného s kontrolou základní funkčnosti aplikace i po drobných úpravách kódu. Jsou-li UI testy vytvořeny pro všechny hlavní i alternativní scénáře, dostává se vývojáři vizuální (na rozdíl od jednotkových testů) ujištění, že je aplikace funkční.

Kapitola 3

Analýza

3.1 Aktuální procesy správy zaměstnanců v Ackee

Procesy spojené se správou zaměstnanců lze rozdělit do několika segmentů.

Prvním segmentem je správa účtu zaměstnance. Nastoupí-li nový zaměstnanec do Ackee, vytvoří mu uživatel s rolí admin (dále pouze „admin“), na základě jeho osobních údajů, účet s profilem ve webovém portálu Miláčci. Při vytváření profilu je novému zaměstnanci přiřazena pracovní pozice, typ smlouvy, kapacita v hodinách, hodinová sazba a případně speciální role team leader nebo admin. V případě potřeby je zaměstnanci vytvořen účet Google, Redmine[4], aj. Profil zaměstnance může admin libovolně upravit. Pokud je se zaměstnancem rozvázána smlouva, „demiláčkuje“ (deaktivuje) admin jeho učet na webovém portálu Miláčci.

Zaměstnanci jsou rozděleni do týmů s vedoucím (team leader), který si může zobrazit profil každého člena svého týmu.

Hodinová sazba zaměstnance se dle potřeby může měnit. Výše nové hodinové sazby bývá výsledkem společného sezení zaměstnance a personalisty. Nová hodnota včetně data, od kterého sazba vchází v platnost, je následně adminem zanesena do systému.

Proces spojený s fakturami zaměstnanců začíná koncem každého měsíce, kdy externí zaměstnanci (s typem pracovní smlouvy IČO) vykazují faktury, jež chtějí, aby jim společnost uhradila. Zaměstnancům, kteří jsou v zaměstnaneckém poměru nebo pracují na dohodu, se faktura generuje automaticky ve formátu XLSX. Jakmile je faktura přidána do systému, čeká na schválení adminem. Poté, co admin ověří, že údaje na faktuře jsou správné, fakturu schválí. Po

[4]Redmine — aplikace pro řízení projektu, správu úkolů atd.

schválení jsou faktury exportovány do systému Expenses[5]. Firemní účetní připraví příkazy k úhradě schválených faktur a admin je následně označí za zaplacené.

3.2 Správa zaměstnanců ve střední firmě

Pro získání lepší představy o problematice správy zaměstnanců ve firmách jsem požádal zakladatele nejmenované české společnosti, jež aktuálně zaměstnává přes 30 osob, aby mi přiblížil, jaké procesy mají nastavené pro správu svých zaměstnanců. Název dotazované společnosti záměrně neuvádím, neboť si nepřála být jmenována.

Informace o zaměstnancích jsou udržovány v Excel tabulkách. Pro nastavení nové finanční odměny existují dva důvody:

- rozšíření kompetencí ve společnosti,
- doba strávená ve společnosti.

Záznamy o navýšení finanční odměny zaměstnanců jsou ukládány v Excel tabulce, pomocí které se pravidelně generují reporty.

Správa faktur firemních zaměstnanců je řešena pomocí aplikace iDoklad, kde externisti vykazují své faktury, pro které firemní účetní připravuje příkazy k úhradě.

3.3 Rozbor požadavků

Požadavky jsou založeny na rozhovorech se zaměstnanci zadavatelské společnosti a výsledcích analýzy návrhu webového portálu. Část nefunkčních požadavků byla přidána tak, aby konceptuálně odpovídala očekávané funkčnosti mobilní aplikace a zbytek se zakládá na průzkumu trhu s iOS zařízeními.

3.3.1 Funkční požadavky

F1: Autorizace pomocí Google účtu Zaměstnanec se do aplikace přihlašuje pomocí svého Google účtu.

F2: Správa faktur zaměstnance Zaměstnanec si může zobrazit přehled svých vykázaných faktur. Přehled bude seřazen sestupně dle data vytvoření faktury. Zaměstnanec s typem pracovní smlouvy IČO může vykázat fakturu reprezentovanou PDF souborem za aktuální měsíc.

[5]Expenses - aplikace pro správu účtenek a faktur

F3: Faktury všech zaměstnanců Adminovi jsou přístupné faktury všech zaměstnanců Ackee, které jsou filtrovatelné podle stavu a měsíce nahrání faktury. Admin může libovolně měnit jejich stav.

F4: Profily zaměstnanců Team leader vidí seznam členů svého týmu včetně jejich profilů. Admin vidí seznam všech firemních zaměstnanců a může jejich účty libovolně upravovat.

F5: Správa hodinových sazeb zaměstnanců Admin může nastavit novou hodinovou sazbu zaměstnance včetně data, od kterého tato hodnota vejde v platnost. Má k dispozici přehled nárůstu hodinových sazeb u jednotlivých zaměstnanců. Dostupná je také hodnota průměrného navýšení hodinové sazby za zvolený měsíc.

3.3.2 Nefunkční požadavky

N1: Zabezpečená komunikace se serverem Komunikace mezi mobilní aplikací a serverem je zabezpečena OAuth2.0 protokolem.

N2: Snadná rozšiřitelnost Architektura aplikace počítá s rozšířením o nové funkcionality.

N3: Uživatelsky přívětivé rozhraní Design mobilní aplikace odpovídá doporučením pro návrh uživatelského rozhraní od společnosti Apple a následuje principy UX.

N4: Podpora iOS od verze 13.0 Aplikace je dostupná pro zařízení s iOS verzí novější než 13.0. Požadavek je založen na statistice, která k prosinci 2020 eviduje, že 90 % držitelů zařízení s iOS používá verzi novější než 13.0 [23].

N5: Funkční na zařízeních iPhone Uživatelské rozhraní je přizpůsobeno zařízením iPhone.

N6: Česká a anglická lokalizace Pro aplikaci je dostupný český i anglický překlad.

N7: Podpora tmavého režimu Aplikace podporuje funkci tmavého režimu, .

3.4 Uživatelské role

Uživatelské role jsou určeny hierarchickou strukturou zaměstnanců společnosti Ackee. Jedná se o následující role.

Zaměstnanec role definující entitu, která používá aplikaci;

Team leader zaměstnanec, který vede jiné zaměstnance;

Admin zaměstnanec, který spravuje ostatní zaměstnance.

3.5 Návrh případů užití

Výsledkem kombinace identifikovaných požadavků a analýzy uživatelských rolí je seznam případů užití, které definují fungování aplikace.

U1: Přihlásit se pomocí Google login
Primární aktér: Zaměstnanec
Popis: Do aplikace se uživatel přihlašuje pomocí svého Google účtu.

U2: Odhlásit se
Primární aktér: Zaměstnanec
Popis: V ojedinělých případech se může stát, že uživatel potřebuje změnit účet jehož prostřednictvím je přihlášený do aplikace. Aby toho docílil, aplikace umožňuje odhlášení aktuálně přihlášeného uživatele.

U3: Přehled faktur zaměstnance
Primární aktér: Zaměstnanec
Popis: Po přihlášení je uživateli prezentován seznam faktur včetně období, za které byla faktura vykázána a informace, která vyjadřuje, v jakém stavu se faktura nachází. Pokud byla faktura přidána, je k dispozici její odpovídající částka v Kč. Jelikož je pro uživatele nejdůležitější aktuální měsíc, je seznam seřazen sestupně dle měsíce a roku, za které byla nebo měla být faktura přidána.

U4: Přidání faktury ve formátu PDF
Primární aktér: Zaměstnanec
Popis: K období, za které ještě zaměstnanec nevykázal fakturu, je možné nahrát soubor ve formátu PDF, jenž reprezentuje fakturu. Funkce je dostupná pouze pro zaměstnance, kteří měli ve zvoleném období uzavřenou pracovní smlouvu s typem IČO. Tabulka 3.1 popisuje detailní scénář případu užití. Tabulky 3.2 a 3.3 reprezentují alternativní scénáře, které mohou nastat pouze za předpokladu, že dojde k chybě na serveru. U tabulky 3.2 se nevygeneruje faktura pro zaměstnance s typem pracovní smlouvy HPP, DPP nebo DPČ. U tabulky 3.3 existuje období, za které je schválena nebo zaplacena faktura, jejíž souborová reprezentace však chybí. Scénář popsaný tabulkou 3.4 popisuje situaci, kdy nastane chyba při komunikaci se serverem.

Krok	Role	Akce
1	Uživatel	Zvolí období, za které chce nahrát fakturu.
2	Aplikace	Zkontroluje, zdali zaměstnanec měl ve zvoleném období typ pracovní smlouvy IČO a jestli již v tomto období neexistuje schválená faktura.
3	Aplikace	Zobrazí dialog se seznamem dostupných dokumentů ve formátu PDF, které se nachází buď v iCloud Drive nebo přímo v mobilním zařízení.
4	Uživatel	Vybere soubor, který chce nahrát.
5	Aplikace	Odešle soubor na server.
6	Aplikace	Zobrazí, že nahrání souboru proběhlo úspěšně.

Tabulka 3.1: Hlavní scénář přidání faktury ve formátu PDF

Krok	Role	Akce
3a1	Aplikace	Notifikuje uživatele, že ve zvoleném období neměl pracovní smlouvu s typem IČO.
3a2	Uživatel	Pokračuje 1. krokem hlavního scénáře 3.1.

Tabulka 3.2: Alternativní scénář přidání faktury ve formátu PDF – nevalidní typ pracovní smlouvy

Krok	Role	Akce
3a1	Aplikace	Notifikuje uživatele, že za zvolené období je již faktura schválena nebo zaplacena.
3a2	Uživatel	Pokračuje 1. krokem hlavního scénáře 3.1.

Tabulka 3.3: Alternativní scénář přidání faktury ve formátu PDF – nevalidní stav faktury

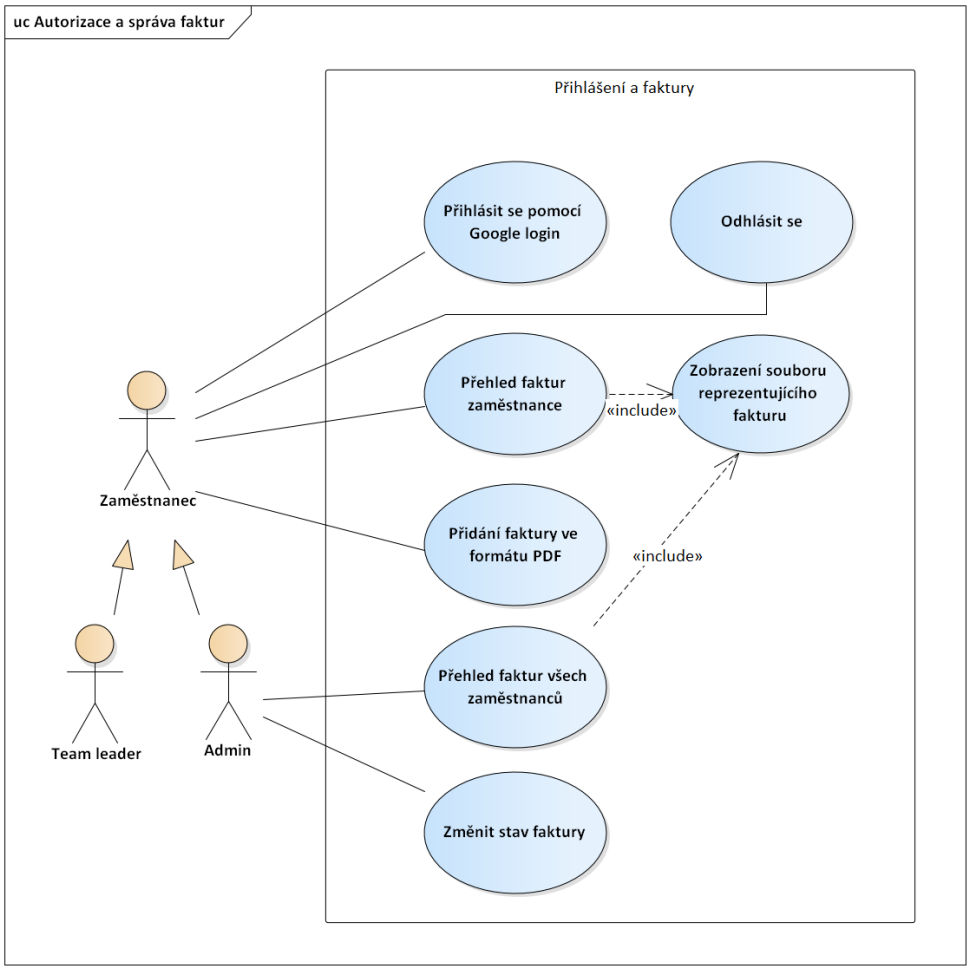
Krok	Role	Akce
3a1	Aplikace	Prezentuje chybovou hlášku uživateli srozumitelnou formou.
3a2	Uživatel	Pokud chce uživatel zkusit soubor nahrát znovu pokračuje 1. krokem hlavního scénáře 3.1.

Tabulka 3.4: Alternativní scénář přidání faktury ve formátu PDF – chyba při komunikaci se serverem

U5: Zobrazení souboru reprezentujícího fakturu
Primární aktér: Zaměstnanec
Popis: Zaměstnanec zvolí období, za které chce zobrazit vykázanou fakturu. Za předpokladu, že faktura ve zvoleném období existuje, je její obsah zobrazen zaměstnanci pomocí webového prohlížeče. Jedná-li se o XLSX formát

(viz. 3.1), může si zaměstnanec otevřít jeho obsah pomocí mobilní aplikace Microsoft Excel.

U6: Přehled faktur všech zaměstnanců
Primární aktér: Admin
Popis: Nezbytnou sekcí systému je seznam faktur všech zaměstnanců. Faktury jsou filtrovány dle období, kdy byly vytvořeny, kde výchozí hodnotou je aktuální měsíc a rok. Položka seznamu obsahuje informaci o stavu, ve kterém se faktura nachází. K dispozici je přehled zaměstnanců, kteří ve zvoleném období nenahráli fakturu.



Obrázek 3.1: Případy užití autorizace a správy faktur

U7: Změnit stav faktury
Primární aktér: Admin
Popis: Ověří-li si admin, že zaměstnancem nahraná faktura je v pořádku,

změní její stav na schválená. Zjistí-li nějaké nedostatky, vrátí ji zpět zaměstnanci. Jakmile získá admin potvrzení o úhradě faktury, označí ji za zaplacenou.

U8: Zobrazit členy týmu
Primární aktér: Team leader
Popis: Team leader má k dispozici seznam členů svého týmu. Seznam je seřazen abecedně dle příjmení člena týmu. V přehledu lze vyhledávat podle jména a příjmení zaměstnance. Po kliknutí na položku seznamu reprezentující člena týmu, je team leaderovi prezentován profil zvoleného zaměstnance.

U9: Zobrazit přehled zaměstnanců
Primární aktér: Admin
Popis: Admin má k dispozici seznam všech zaměstnanců. Vlastnosti seznamu odpovídají vlastnostem popsaných v U8: Zobrazit členy týmu.

U10: Zobrazit profil zaměstnance
Primární aktér: Team leader, Admin
Popis: Uživatel má k dispozici profil vybraného zaměstnance včetně jeho osobních údajů, typu pracovního poměru nebo výše finanční odměny.

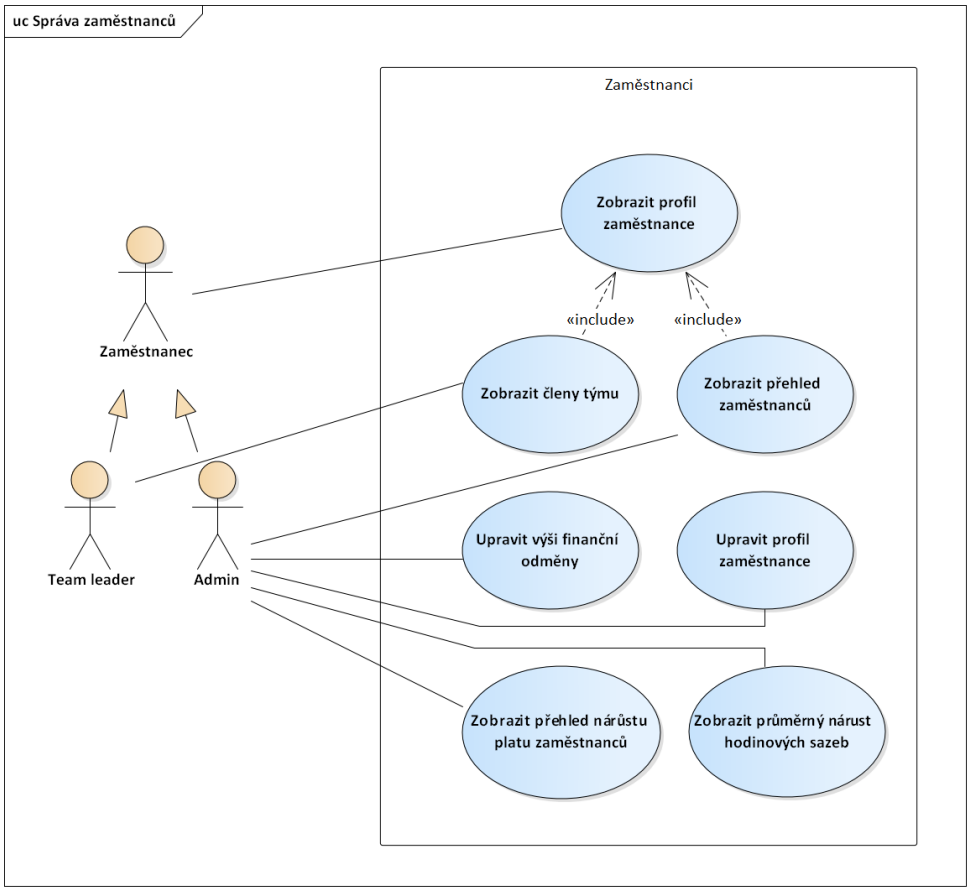
U11: Upravit profil zaměstnance
Primární aktér: Admin
Popis: Admin zvolí zaměstnance jehož profil chce upravit. Aplikace mu umožní změnu jeho jména, příjmení, pracovní pozice, titulu, hodinové kapacity, telefonního čísla, osobního emailu a typu pracovní smlouvy. Po dokončení úprav admin uloží provedené změny. Po úspěšném uložení dat je admin přesměrován na profil zaměstnance.

U12: Upravit výši finanční odměny
Primární aktér: Admin
Popis: V profilu zaměstnance se vyskytuje informace o výši jeho hodinové sazby. Tato hodnota lze změnit kliknutím na tlačítko symbolizující akci úpravy. Aplikace zobrazí číselné pole, do kterého admin zadá novou hodnotu hodinové sazby. Zvolí datum, od kterého vejde nastavená hodinová sazba v platnost a proces dokončí uložením.

U13: Zobrazit přehled nárůstu platu zaměstnanců
Primární aktér: Admin
Popis: Informace o změnách hodinových sazeb jednotlivých zaměstnanců jsou dostupné v seznamu, který je seřazen abecedně dle příjmení zaměstnance.

U14: Zobrazit průměrný nárůst hodinových sazeb
Primární aktér: Admin

Popis: Admin zvolí období, za které chce vidět průměrnou hodnotu nárůstu hodinové sazby u všech zaměstnanců.



Obrázek 3.2: Případy užití správy zaměstnanců

3.6 Výběr technologií

V rámci kapitoly 2 jsem analyzoval praktiky, jež mohu zvolit při řešení problémů, které s sebou přináší implementace mobilní aplikace. V této podkapitole se zaměřím na výběr vhodných technologií a popíši důvody, které mě k dané volbě přivedly.

3.6.1 Architektonický styl

Po důkladné analýze dostupných architektonických stylů jsem dospěl k závěru, že styl MVVM nejlépe odpovídá potřebám projektu, neboť je vhodný pro jeho rozsah. Umožňuje bezproblémovou testovatelnost business logiky. Poskytuje nástroje k atomickému dělení logických celků do samostatných viewmodelů,

čehož využiji, bude-li potřeba v rámci jednoho viewcontrolleru zobrazit více na sobě nezávislých dat. Styl je doporučován zadavatelskou společností, což usnadní tamním vývojářům pokračovat na projektu. Při výběru hrála do jisté míry roli i má předchozí zkušenost s tímto stylem, jež mi usnadní rychlejší adaptaci.

3.6.2 Programovací jazyk a paradigma

Jazyk Swift byl pro mě jasnou volbou, neboť se v posledních letech stal standardem pro tvorbu nativních iOS aplikací. Díky tomu, že je jazykem multiparadigmatovým, nejsem omezen při výběru ani při počtu volených možností. Implementace bude zajisté obsahovat objekty, které mezi sebou budou komunikovat, z čehož vyplývá objektové paradigma. Pro přehlednost a lepší testovatelnost kódu budu pro logické celky nejdříve definovat protokoly. Dle definice 2.5 takto použité protokoly znamenají použití protokolového paradigma. Mým plánem je také použít některou z funkcionálně reaktivních knihoven, takže FRP bude také obsaženo. Aplikuji tedy všechna analyzována paradigmata. Výsledkem by měl být přehledný kód, který bude snadno rozšiřitelný, modulární a dobře testovatelný.

3.6.3 FRP knihovna

RxSwift knihovnu jsem upřednostnil před ReactiveSwift zejména díky velkému množství dostupných online návodů. Dalším důvodem je má znalost ReactiveX API díky předešlé zkušenosti s RxJs. Hlavní překážkou v tomto rozhodnutí byla skutečnost, že nebudu moci použít operátor $<\sim$, který se stará o propojení `Observables` a o ukončení datových toků, což by mi ušetřilo psaní zbytečného „boilerplate“ kódu.

3.6.4 Komunikace se serverem

Výsledná volba založená na průzkumu vhodných řešení pro síťovou komunikaci se vzdáleným serverem bude knihovna Alamofire. Tato knihovna poskytuje robustní možnosti pro získávání/zasílání dat z/na server. Zároveň minimalizuje množství potřebného kódu pro vykonání stejné operace oproti `URLSession`. Nicméně bych se rád inspiroval strukturou volání síťové vrstvy definovanou knihovnou Moya a v upravené formě ji použil.

3.6.5 Tvorba uživatelského rozhraní

Uživatelské rozhraní budu vytvářet ve Storyboardu. Je intuitivní pro začátečníky, neboť umožňuje navrhovat UI komponenty a měnit jejich vlastnosti pomocí vizuálního nástroje. Na rozdíl od tvorby UI pomocí kódu, u kterého je zapotřebí pokročilejší znalost vlastností a funkcí komponent definovaných kni-

hovnou UIKit. Bonusem je možnost použití GUI pro tvorbu Auto Layout omezení.

3.6.6 Navigace mezi obrazovkami

Z počátku jsem uvažoval o použití Segue jakožto způsobu navigace mezi obrazovkami. Zdál se být přirozeným výběrem, jelikož jsem zvolil storyboard na tvorbu uživatelského rozhraní. Toto řešení však přidává nemalé množství kódu do viewcontrollerů a činí je tak hůře testovatelnými. Mým cílem bylo delegovat kód zodpovědný za inicializaci a přechod viewcontrollerů do třídy nezávislé na knihovně UIKit, což umožňuje vzor Coordinátor. Nepříjemností je nutnost držet zobrazené koordinátory v lokálních proměnných a hlídat si jejich odstranění. Tento nedostatek je vyvážen množstvím dostupné dokumentace, diskuzí a návodů třetích stran.

Kapitola 4

Návrh

4.1 Uživatelské rozhraní

Pro vytvoření návrhu uživatelského rozhraní použiji několikakrokový proces. Na základě případů užití sestavím seznam úloh, které detailně popisují systém z pohledu uživatele. Získané úlohy roztřídím a seskupím podle jejich logického významu a stanovených kritérií. Pomocí kategorizovaných úloh vytvořím graf závislostí mezi jednotlivými úlohami, který mi pomůže s návrhem low-fidelity prototypu. Výsledkem přidání vizuálních prvků vznikne high-fidelity prototyp, který budu dále validovat pomocí heuristických testů a který bude sloužit jako předloha pro tvorbu finálního uživatelského rozhraní mobilní aplikace.

Jednotlivé kroky budou rozděleny dle role uživatele, přičemž v textu této diplomové práce budu prezentovat pouze návrhy spojené s rolí zaměstnanec. Ostatní části návrhu pro role team leader a admin se budou pro svou rozsáhlost nacházet v příloze.

4.1.1 Seznam úloh

Díky případům užití 3.5 jsem pro roli zaměstnanec identifikoval následující úlohy:

- Přihlaš se pomocí Google účtu.
- Zadej Google účet.
- Zobraz přihlášení.
- Odhlaš se.
- Zobraz přehled svých faktur.

- Vyber libovolnou fakturu.
- Otevři souborovou reprezentaci faktury.
- Zobraz PDF reprezentaci faktury.
- Zobraz XLSX reprezentaci faktury.
- Přidej PDF reprezentaci faktury.
- Vyber PDF soubor.
- Notifikuj o úspěšně nahrané faktuře.
- Obnov seznam.
- Zobraz svůj profil.
- Zobraz výši své hodinové sazby.
- Zobraz nadcházející výši své hodinové sazby.
- Zobraz přehled svého finančního ohodnocení za uplynulé měsíce.
- Zobraz chybovou hlášku.
- Zobraz stav načítání.

4.1.2 Skupina úloh

Úlohy je potřeba seskupit dle jejich významu. Hlavním kritériem je, zdali úloha reprezentuje zobrazení dat nebo uživatelský vstup. Barevně odliším úkoly, které jsou na sobě logicky závislé. V rámci kategorie seřadím úlohy sestupně dle jejich důležitosti.

Kategorie:

- stavové hlášení,
- správa faktur,
- autorizace,
- detail zaměstnance.

Úlohy reprezentující zobrazení:

- Zobraz stav načítání.
- Zobraz chybovou hlášku.
- Zobraz přehled svých faktur.
- Zobraz PDF reprezentaci faktury.
- Zobraz XLSX reprezentaci faktury.
- Zobraz notifikaci o úspěšně nahrané faktuře.
- Zobraz výši své hodinové sazby.
- Zobraz nadcházející výši své hodinové sazby.
- Zobraz přehled svého finančního ohodnocení za uplynulé měsíce.
- Zobraz svůj profil.
- Zobrazit přihlášení.

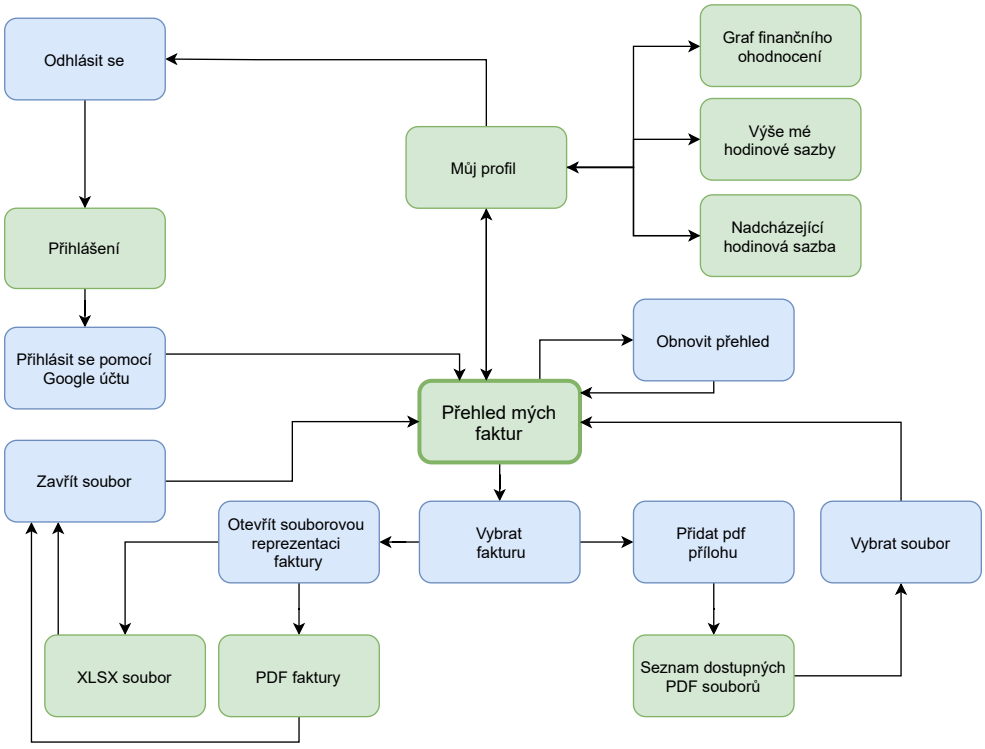
Uživatelské akce:

- Přihlaš se pomocí Google účtu.
- Zadej Google účet.
- Odhlaš se.
- Přidej PDF reprezentaci faktury.
- Nahraj jiný PDF soubor.
- Vyber PDF soubor.
- Vyber libovolnou fakturu.
- Otevři souborovou reprezentaci faktury.
- Obnov seznam.

4.1.3 Graf úloh

Mezi seskupenými úlohami se snadněji hledají závislosti, které vyjádřím pomocí grafu. Definuji přechody mezi úlohami zobrazujícími obsah, kterých lze dosáhnout prostřednictvím uživatelských akcí.

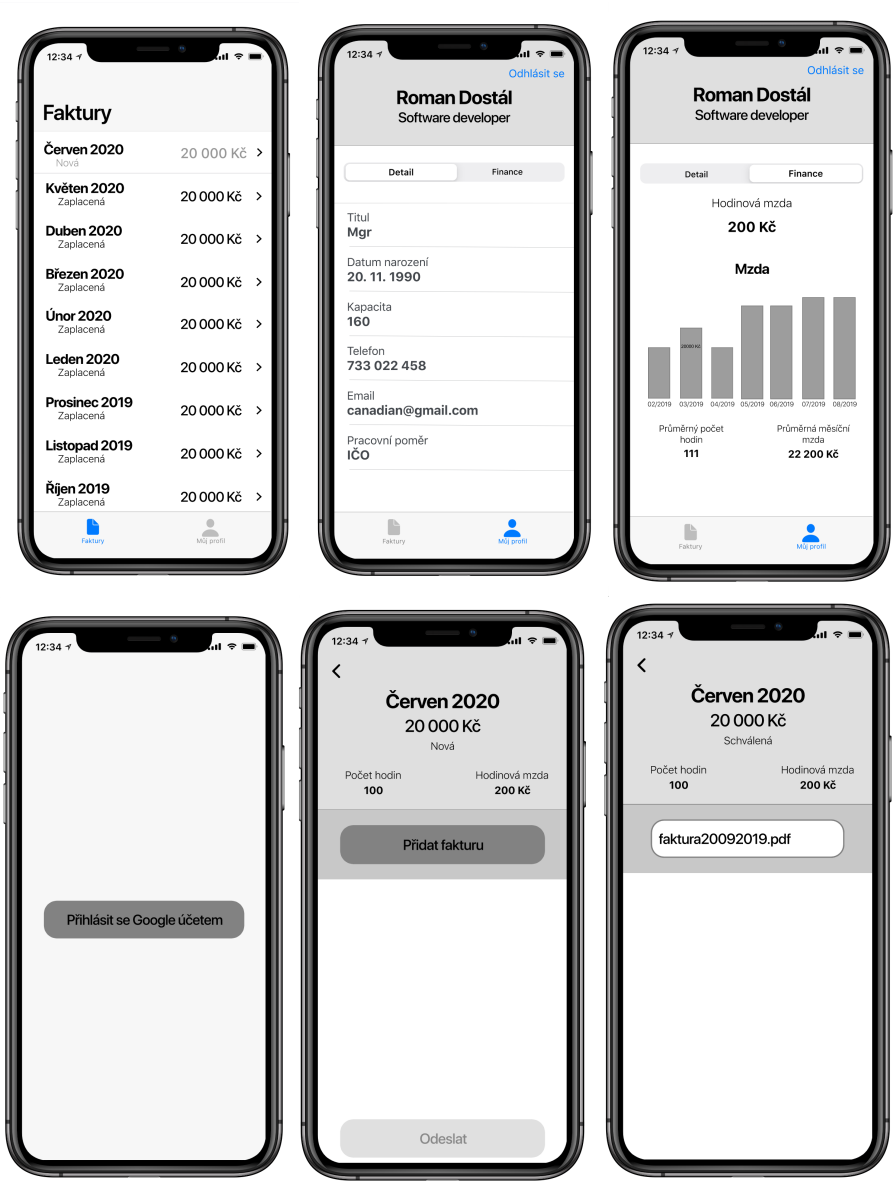
Výsledný graf úloh dostupných pro zaměstnance, je znázorněn na obrázku 4.1, kde zelená barva definuje zobrazovací úlohy a modrá uživatelské akce. Šipka reprezentuje možnost a směr přechodu.



Obrázek 4.1: Graf úloh pro roli zaměstnanec

4.1.4 Lo-fi prototyp

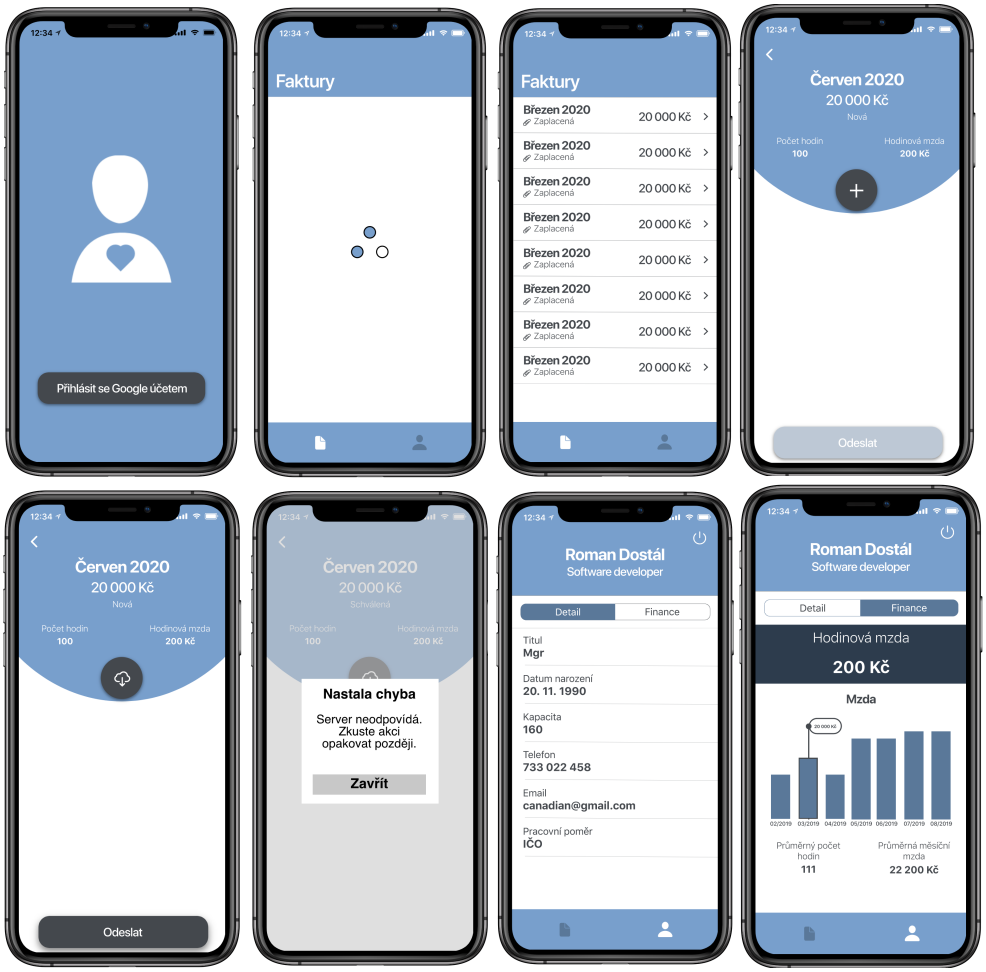
Za pomoci přechodů mezi prioritizovanými úlohami vytvořím low-fidelity prototyp, kterým si ujasním přibližnou podobu aplikace. První verzi tohoto prototypu vytvořím v papírové formě, kterou poté převedu do prototypovacího nástroje Invision.



Obrázek 4.2: Lo-fi prototyp aplikace pro roli zaměstnanec

4.1.5 Hi-fi prototyp

Lo-fi prototyp rozšířím o jednobarevnou paletu založenou na pastelové verzi modré firemní barvy společnosti Ackee. Přidám řadu vizuálních komponent včetně animací během přechodů mezi obrazovkami. Výsledkem bude high-fidality prototyp, který představuje předpokládaný design aplikace a který dokáže suplovat reálnou interakci uživatele s aplikací.



Obrázek 4.3: Hi-fi prototyp aplikace pro roli zaměstnanec

4.1.6 Testování prototypu

Prototyp je zapotřebí podrobit heuristickým testům, které poskytnou zpětnou vazbu o tom, zdali je design uživatelsky přívětivý a neporušuje základní UX principy.

4.1.6.1 Nielsenovo heuristické vyhodnocení

Nielsonova heuristická analýza je test UI, který lze provést bez přítomnosti skutečných uživatelů. Jedná se o 10 pravidel neboli Nielsonovo desatero, jež uživatelské rozhraní nesmí porušovat:

1. viditelnost stavu systému (systém nesmí zamrznout nebo musí dát uživateli najevo, že probíhá načítání);

2. shoda mezi systémem a realitou (zachování konvencí z reálného světa, funkce ikonky musí odpovídat její podobě);
3. minimální zodpovědnost uživatele (uživatel má vždy možnost vrátit zpět provedenou operaci);
4. shoda s použitou platformou a obecnými standardy (aplikace odpovídá HIG);
5. prevence chyb (uživateli není umožněno zadat špatnou hodnotu);
6. kouknu a vidím (aktuální pozice uživatele v aplikaci je viditelná);
7. flexibilita a efektivita (aplikace podporuje módy pro běžného a zkušeného uživatele);
8. estetika a minimalistický design (UI zobrazuje vždy jen relevantní informace a možnosti);
9. smysluplné chybové hlášky (případné chyby jsou prezentovány uživateli ve srozumitelném jazyce);
10. návod a dokumentace.

Test byl proveden dvěma hodnotiteli a jeho výsledek je shrnut v tabulce 4.1. Tabulka obsahuje identifikovaný problém, číslo porušeného pravidla a prioritu nalezené chyby, kde 1 znamená zanedbatelný problém a 5 těžký problém, který uživatel nestrpí. Detailní výstupy obou hodnotitelů jsou k dispozici v příloze této diplomové práce.

Problém	Pravidlo	Priorita
Odhlášení je symbolizováno ikonou pro vypnutí.	2	4
Vrácení faktury je symbolizováno ikonou pro zrušení.	2	5
Po nahrání faktury již nelze operaci vzít zpět.	3	4
Po úpravě hodinovky již nelze operaci vzít zpět.	3	1
Po zaplacení faktury již nelze operaci vzít zpět.	3	1
Viditelná tlačítka na položce v seznamu faktur.	4	3
Funkce potvrzování akcí není přítomná.	5	1
Nadbytečnost detailu faktury	8	3

Tabulka 4.1: Výsledky heuristické analýzy

Problémy s prioritou větší nežli 2 upravím v rámci UI cílové aplikace. Bude se jednat o následující úpravy:

- Změna ikony pro odhlášení.
- Změna ikony pro vrácení faktury.
- K dispozici bude možnost znovunahrání faktury za předpokladu, že faktura již není schválená.
- Zaplacené faktury bude uživatel moci vrátit do stavu schválená nebo nová.
- Akce na položce seznamu budou viditelné až po přejetí po položce doleva.
- Detail faktury bude odstraněn. Funkce nahrání a stažení souborové reprezentace faktury bude dostupná po kliknutí na položku seznamu.

4.2 Architektura

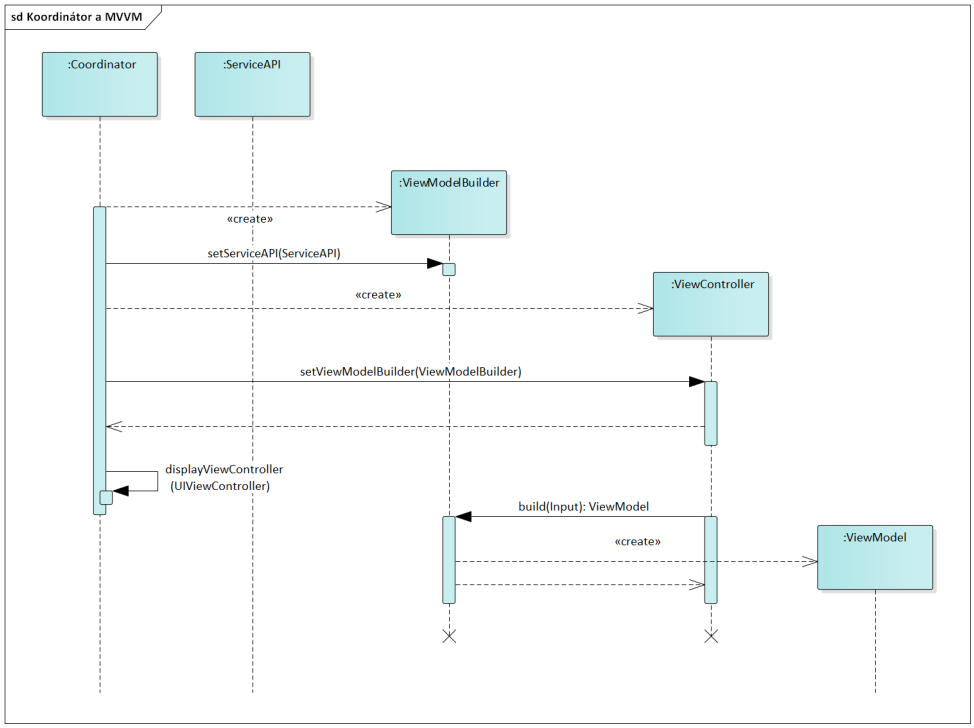
V rámci této podkapitoly navrhnu architekturu mobilní aplikace. Nejdříve se zaměřím na kombinaci architektonického stylu MVVM a vzoru Coordinator. Dále vytvořím strukturu potřebnou k vytvoření síťové komunikace, a nakonec tyto dvě části propojím. Na základě navrženého uživatelského rozhraní lze identifikovat hlavní vizuální celky aplikace. Pro každý celek bude existovat potomek třídy `UIViewController`, který bude zodpovědný za prezentování dat uživateli. Viewcontroller je inicializován pomocí svého koordinátoru a deleguje svou business logiku na jeden nebo více viewmodelů.

4.2.1 Inicializace koordinátoru

Koordinátor vytváří jiný koordinátor, kde na samotném vrcholku hierarchie stojí `AppCoordinator`, který je jako jediný inicializován po spuštění aplikace v třídě `SceneDelegate`.

4.2.2 Inicializace viewcontrolleru

Za inicializaci viewcontrolleru je zodpovědný jeho koordinátor. Ten nejdříve vytvoří jeho instanci, kterou za pomoci objektu `BaseNavigationController` nebo `BaseTabBarController` zobrazí uživateli. Dále implementuje funkci reprezentovanou typealiasem `ViewModelBuilder`, v rámci které poskytne závislosti, které jsou nezbytné pro vytvoření viewmodelu. Jeho inicializaci však ponechá viewcontrolleru, který si danou instanci vytvoří sám v rámci svého životního cyklu. Pro lepší přehlednost jsem v rámci sekvenčního diagramu 4.5 modeloval `ViewModelBuilder` jako objekt.



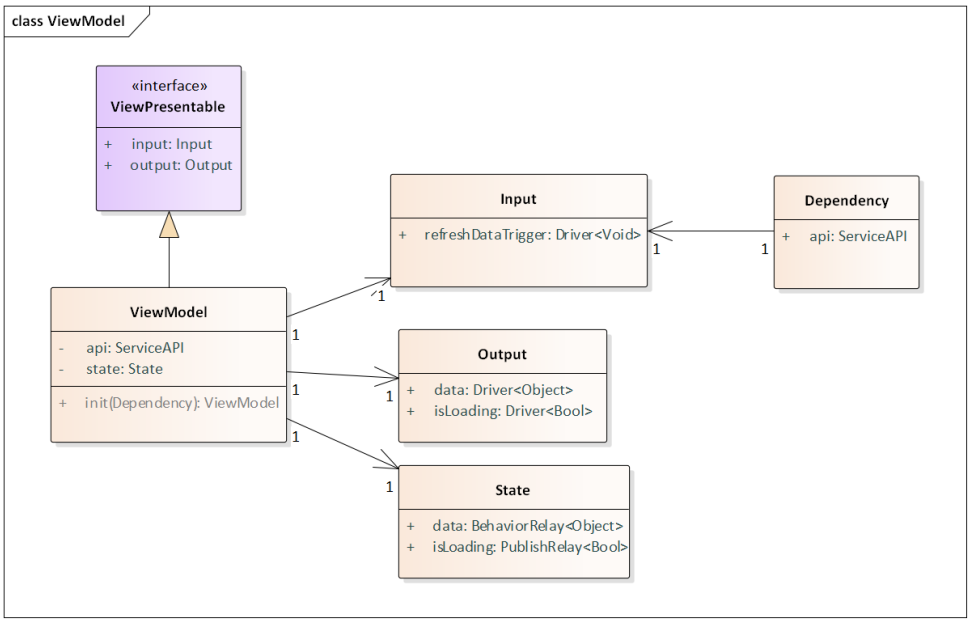
Obrázek 4.4: Sekvenční diagram - inicializace MVVM architektury

4.2.3 Struktura viewmodelu

Každý viewmodel implementuje svůj protokol viewpresentable, který charakterizuje jeho chování. Protokol definuje strukturu vstupů, která je přijímána z viewcontrolleru a strukturu výstupů, která je posílána zpět. Vstupy jsou datové toky emitující události spojené s interakcí uživatele. Příkladem vstupu může být kliknutí na tlačítko nebo požadavek na obnovení seznamu faktur. Výstupy jsou datové toky emitující události spojené se stavem modelových dat. Například se jedná o data získaná ze serveru nebo informaci, že nastala chyba. Datové toky vstupů i výstupů jsou reprezentovány instancemi třídy `Driver` 2.6.1. Pro vytváření instance viewmodelu je definován typealias s názvem `ViewModelBuilder`, což je definice funkce, která dostává parametrem závislosti nutné pro inicializaci viewmodelu, jehož instanci následně vrací. Mezi závislosti patří vstupy z viewcontrolleru nebo instance tříd, které načítají data. Jedná se například o referenci na objekt, který zprostředkovává komunikaci se serverem.

Viewmodel si kromě vstupů a výstupů drží také proměnnou reprezentující jeho aktuální stav. Ten je tvořen kombinací instancí tříd `BehaviourRelay` a `PublishRelay`. Při inicializaci se viewmodel přihlásí k odběru událostí získaných z viewcontrolleru a specifikuje logiku, která má proběhnout při jejich obdržení.

Dalším krokem je vytvoření struktury reprezentující výstup, který lze získat transformací dat vnitřního stavu.



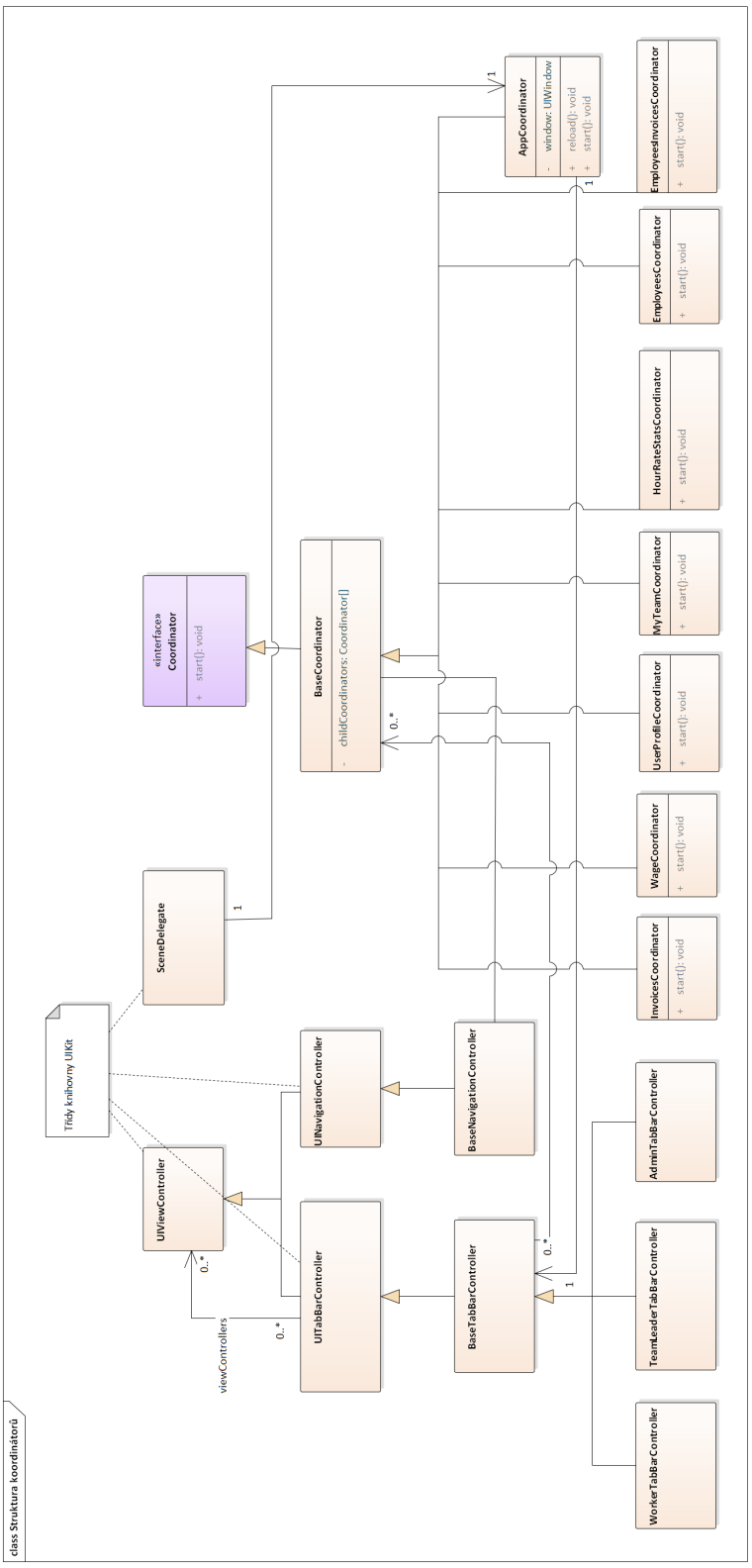
Obrázek 4.5: Diagram tříd - struktura viewmodelu

4.2.4 Struktura koordinátorů

Struktura koordinátorů je vyobrazena pomocí třídního diagramu 4.6.

Použité konstrukce:

- **BaseTabBarController** — Potomek třídy `UITabBarController`, který slouží k podpoře vzoru Coordinator. Třída si drží list koordinátorů, jejichž viewcontrollery prezentuje jako panelové karty. Nastavuje vlastnosti panelové lišty.
- **WorkerTabBarController** — Potomek `BaseTabBarController`, který definuje dostupné obrazovky prostřednictvím koordinátorů, které si role zaměstnanec může zobrazit. Jedná se o:
  - `InvoicesCoordinator`,
  - `WageCoordinator`,
  - `UserProfileCoordinator`.
- **TeamLeaderTabBarController** — Stejně jako `WorkerTabBarController` určuje obrazovky dostupné pro určitou roli uživatele. V tomto případě se jedná o roli team leader.



Obrázek 4.6: Diagram tříd - struktura koordinátorů

- **AdminTabBarController** — Jak už název napovídá, stará se o vymezení funkcí přístupných pro roli admin.

- **BaseNavigationController** — Potomek třídy `UINavigationController`, který upravuje vlastnosti navigační lišty.

- **Coordinator** — Protokol definující, že každý koordinátor musí implementovat metodu pro zobrazení svého viewcontrolleru. Poskytuje funkce pro přidání a odebrání „child“ koordinátoru.

- **BaseCoordinator** —Abstraktní třída, inicializující pole „child“ koordinátorů.

- **AppCoordinator** — První koordinátor vytvořený po spuštění aplikace. Za podmínek, že uživatel není přihlášený, spustí koordinátor prezentující obrazovku s přihlášením. Pokud je uživatel přihlášený, vytvoří a zobrazí na základě jeho role odpovídajícího potomka třídy `BaseTabBarController`. Proces je modelován pomocí diagramu 4.9.

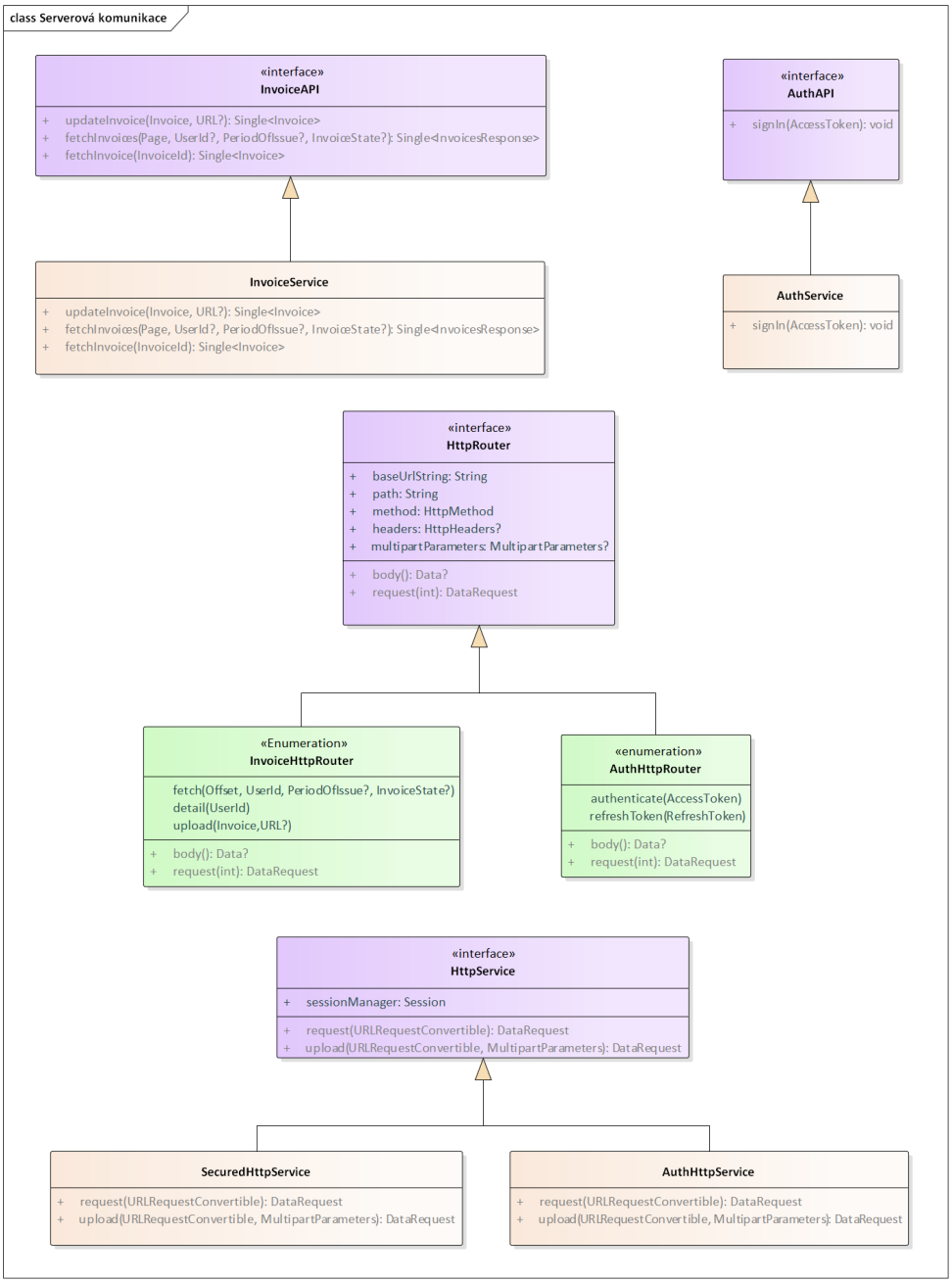
- **InvoicesCoordinator** — Koordinátor zodpovědný za inicializaci viewcontrolleru, jenž prezentuje přehled faktur přihlášeného zaměstnance.

4.3 Načítání a odesílání dat

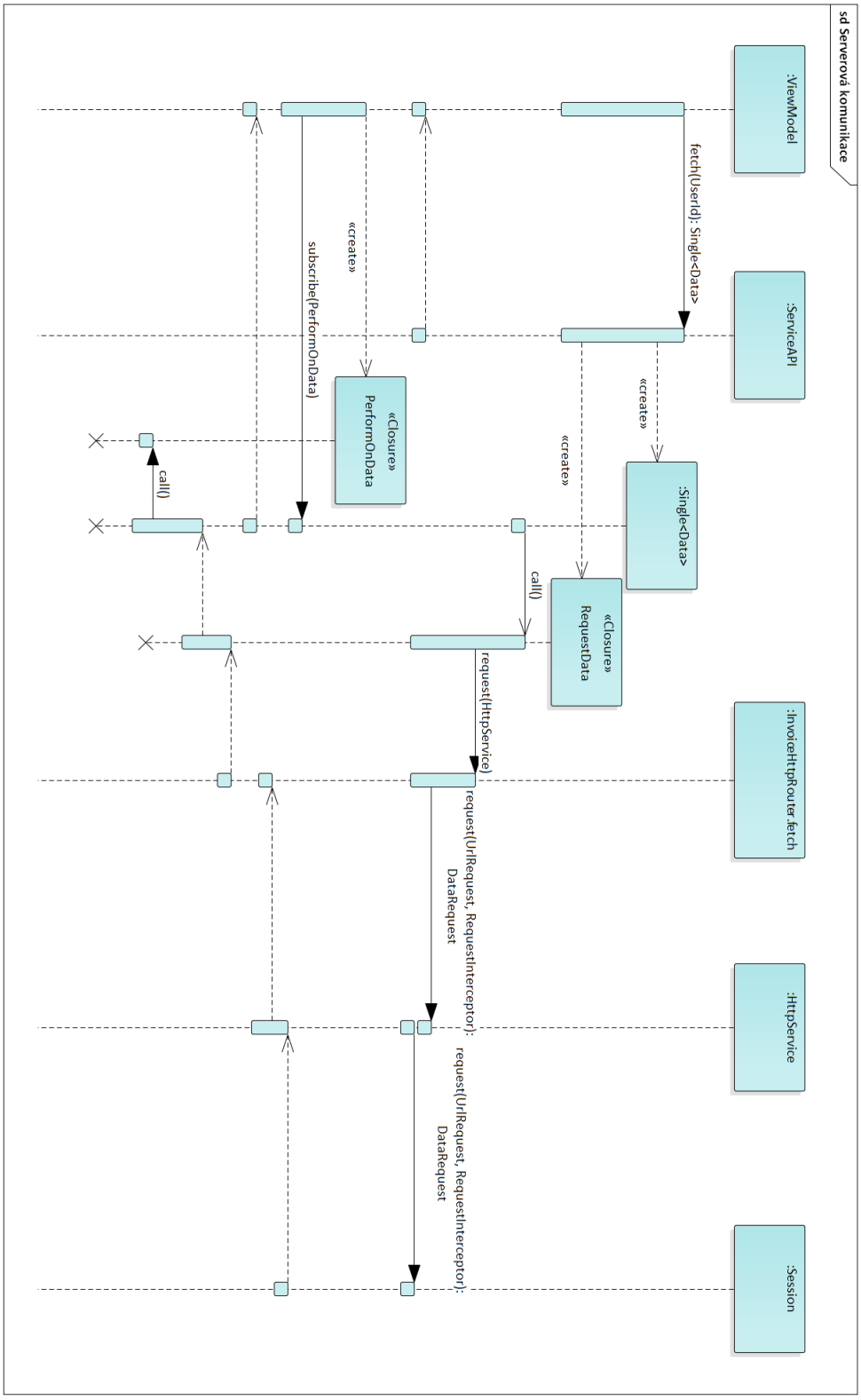
Mobilní aplikace bude zasílat HTTP požadavky serveru, který poskytuje REST API, jež vrací JSON reprezentaci dat. Struktura síťové vrstvy, pomocí které bude aplikace komunikovat se serverem, je inspirována principy představenými knihovnou Moya 2.7.3 a vzorem APIRouter [24].

Každý požadavek na serverový koncový bod je reprezentován hodnotou výčtového typu, který implementuje protokol `HttpRouter`. Tento protokol slouží k vytvoření onoho požadavku včetně jeho atributů. Vytvořený požadavek je dále předán třídě `HttpService`, která přidává dodatečné funkcionality v podobě interceptorů, jenž lze zavolat v různých fázích zpracování požadavku. Pomocí `Session` vytváří a vrací `DataRequest`.

Pro každý výčtový typ implementující `HttpRouter` existuje třída, která volá hodnoty reprezentující požadavek na server, předává mu instanci `HttpService` a reaguje na navrácená data. Tyto třídy se nazývají `ServiceAPI` a vrací `Single` 2.6.1, který emituje po obdržení data získaná ze serveru. Proces je znázorněn na sekvenčním diagramu 4.8.



Obrázek 4.7: Diagram tříd - serverová komunikace



Obrázek 4.8: Sekvenční diagram - serverová komunikace

4.4 Struktura logických celků

4.4.1 Faktury přihlášeného zaměstnance

`InvoicesViewController` zobrazuje přehled faktur přihlášeného zaměstnance. Přehled podporuje stránkování a možnost znovunačtení. Aby tyto funkce byly dostupné, musí `InvoicesViewModel` na vstupu přijímat:

- `refreshTrigger` - požadavky na aktualizaci seznamu faktur,
- `loadNextPageTrigger` - požadavky na načtení další stránky s fakturami.

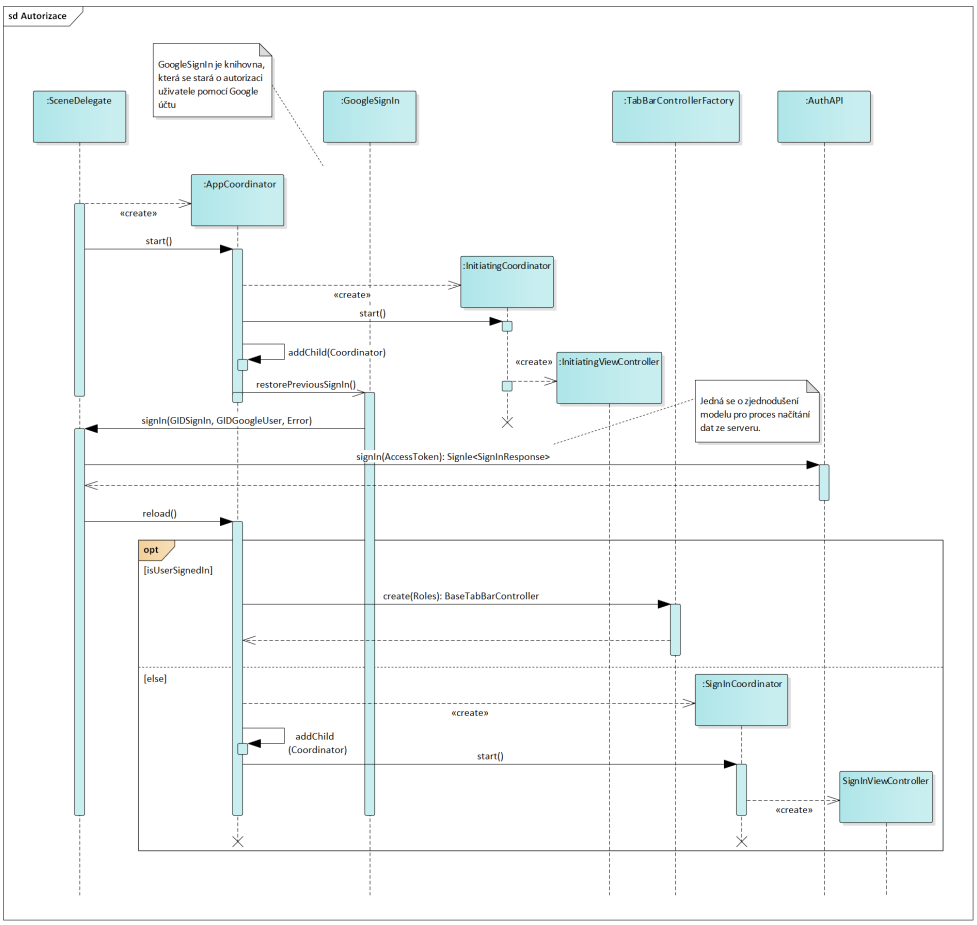
Souborové reprezentace faktur jsou dostupné po kliknutí na položku seznamu. Tyto akce jsou delegovány na `InvoicesActionViewModel`, kde pokud již soubor existuje, je viewcontrolleru zpět zaslána událost s URL, kde se soubor nachází. Viewcontroller na událost reaguje zobrazením URL pomocí třídy `SFSafariViewController` z knihovy UIKit. Za předpokladu, že soubor neexistuje, je emitována událost, že je zapotřebí vybrat soubor pro odeslání na server. Zvolení PDF souboru umožňuje třída `UIDocumentPickerViewController`. Po výběru je PDF soubor zaslán na server.

4.4.2 Autorizace

Autorizace uživatele probíhá dvoufázově pomocí protokolu OAuth2.0. Na základě Google účtu aplikace obdrží od autorizačního serveru Googlu access token (dále pouze „GAT“), pomocí kterého se získá z Ackee serveru access (dále pouze „AAT“) a refresh token. AAT se poté přidává do hlavičky každého HTTP požadavku, prostřednictvím kterého se získávají chráněná data uživatele. Konkrétně se jedná o hlavičkovou položku s názvem Authorization s hodnotou ve tvaru Bearer AAT.

Po startu aplikace se `AppCoordinator` pokusí načíst naposledy přihlášeného uživatele, zavoláním metody `restorePreviousSignIn()`, jež nabízí knihovna GoogleSignIn. Pokud uživatel nebyl v minulosti přihlášen, je mu prezentován `SignInViewController`, který mu umožní přihlášení. Jinak je zaslán autorizační požadavek na Ackee server pro obdržení AAT. Společně s AAT je získáno jméno uživatele a seznam jeho aktuálních rolí. Na základě těchto rolí `TabBarControllerFactory` vytvoří potomka třídy `BaseTabBarController`, který zobrazí pouze funkce dostupné pro danou roli.

Za automatické přidání AAT do hlavičky požadavku je zodpovědná třída `AccessTokenInterceptor`, která je definována v rámci `SecuredHttpService` a která před odesláním požadavku modifikuje jeho hlavičku.



Obrázek 4.9: Sekvenční diagram - autorizace

4.4.3 Faktury všech zaměstnanců

Struktura je velmi podobná fakturám přihlášeného zaměstnance. Pouze položka seznamu zobrazuje místo období, jméno zaměstnance, který ji vykázal. Přehled lze filtrovat dle období a stavu faktury. `EmployeesInvoicesViewModel` tedy musí navíc na vstupu obdržet tyto datové toky:

- `periodOfIssueTrigger` - období, za které chce uživatel vidět faktury, se změnilo;
- `invoiceStateTrigger` - stav, podle kterého chce uživatel filtrovat faktury, se změnil;
- `invoiceChanged` - faktuře se změnil stav.

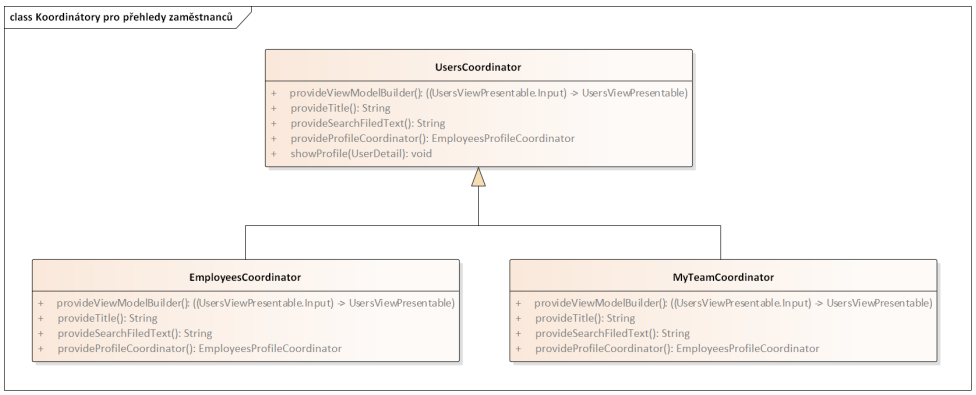
Na výše zmíněné datové toky reaguje viewmodel zavoláním `InvoicesAPI`, který zašle požadavek upravený o nastavené filtry na server.

Další funkcí je možnost změnit stav faktury. Každý stav reprezentovaný výčtovým typem `InvoiceState` má množinu dostupných stavů, do kterých z něj lze přejít, reprezentovanou proměnnou `allowedTransitions: [InvoiceState]`. Po přetažení doleva po položce seznamu faktury se zobrazí seznam dostupných stavů. Zvolí-li uživatel pro fakturu nový stav, je tato informace emitována do třídy `InvoicesActionViewModel`, která na ni reaguje zasláním POST požadavku informujícího server o změně stavu faktury.

4.4.4 Členové týmu a zaměstnanci

Přehled členů týmu a přehled všech zaměstnanců se liší převážně v zobrazených datech. Vizuální komponenty jsou shodné a postačí tedy jeden viewcontroller `UsersViewController`. Drobné rozdíly jsou v názvu panelové karty („Můj tým“ vs „Zaměstnanci“) a v zástupném symbolu pro vyhledávání dle jména („Vyhledat člena týmu“ vs „Vyhledat zaměstnance“). Tyto odlišnosti jsou do viewcontrolleru promítnuty modifikací provedenou pomocí příslušných koordinátorů.

Největší překážkou je rozdílný přistup v načítání zobrazených zaměstnanců. Pro získání členů týmu existuje `MyTeamViewModel` a pro získání seznamu všech zaměstnanců `EmployeesViewModel`. Oba viewmodely rozšiřují abstraktní třídu `UsersViewModel` a přetěžují její funkci `load()`, která za pomoci `UserAPI` získává data ze serveru.



Obrázek 4.10: Diagram tříd - koordinátory pro přehledy zaměstnanců

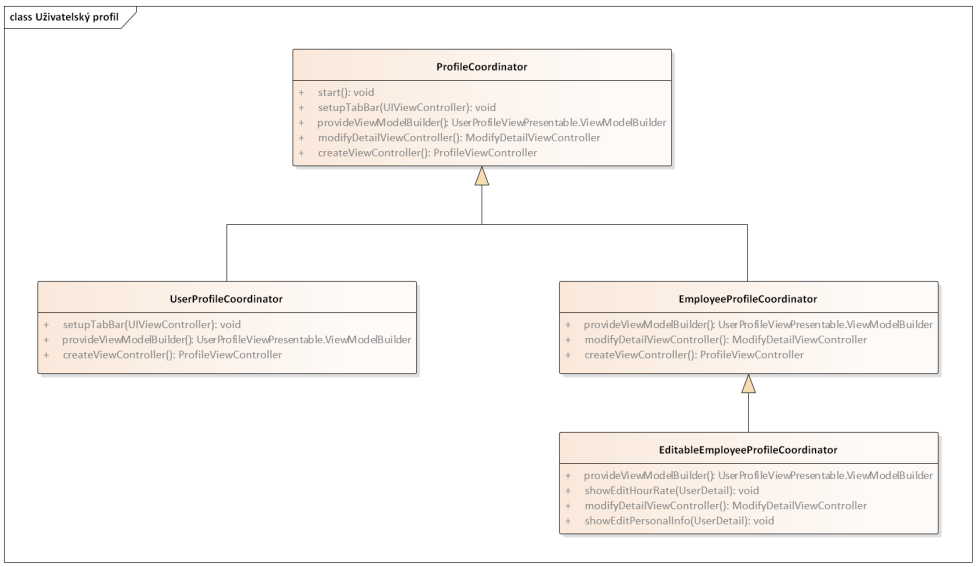
4.4.5 Uživatelský profil

Obrazovka uživatelského profilu existuje ve třech podobách, kde každá zobrazuje jiné informace 4.2.

	Osobní údaje	Hodinová mzda	Úprava osobních údajů	Úprava hodinové mzdy
Profil uživatele	Ano	Ne	Ne	Ne
Profil člena týmu	Ano	Ano	Ne	Ne
Profil zaměstnance	Ano	Ano	Ano	Ano

Tabulka 4.2: Dostupné funkce v různých typech uživatelských profilů

Stejně jako u seznamu zaměstnanců i zde existuje pouze jeden viewcontroller a rozdíly jsou stanoveny pomocí metody `modifyViewController()` koordinátory, jejichž struktura je znázorněna na diagramu 4.11.



Obrázek 4.11: Diagram tříd - uživatelský profil

Přechod na child viewcontroller `ProfileViewController` umožňuje přechod do dvou viewcontrollerů:

- `EditHourlyRateViewController` - umožňuje úpravu hodinové mzdy;
- `EditPersonalInfoViewController` - slouží ke změně osobních údajů.

`EditableUserProfileViewModel` definuje atribut `router` reprezentovaný typem `PublishRelay<UserDetail>`, k jehož odběru se přihlásí inicializující koordinátor. Obdrží-li koordinátor informaci o záměru uživatele zobrazit jeden ze dvou „child“ viewcontrollerů, vytvoří a spustí relevantní „child“ koordinátor.

4.4.6 Mzda zaměstnance

Obrazovka se mzdou zaměstnance se skládá z hodnoty aktuální/následující hodinové sazby a grafu finančních odměn rozdělených dle období. Pro lepší přehlednost budou pro tyto nezávislé části vytvořeny dva viewcontrollery (`WageChartViewController` a `HourlyRateViewController`), které budou zobrazeny v rámci `WageViewController`, který bude sloužit jako jejich kontejner.

Kapitola 5

Implementace

V této kapitole se zaměřím na zajímavosti a problémy, se kterými jsem se potýkal při realizaci aplikace.

5.1 Obnovení tokenu

Access token obdržený z Ackee serveru (AAT) má svou trvanlivost. Po uplynutí určité doby přestává být token validní a je potřeba jej obnovit. Během procesu získávání AAT je obdržen také refresh token a expirační doba, která je vyjádřena počtem sekund, po jejichž uplynutí se stane AAT nevalidním. Aktuálně je doba platnosti AAT nastavena na 60 minut. Jakmile tato doba uplyne, koncové body serveru, které požadují, aby byl požadavek autentizován, vrátí kód 401, což znamená, že byl odepřen přístup k požadovaným datům. Takto vzniklou situaci jsem vyřešil pomocí již zmíněného interceptoru `AccessTokenInterceptor`, který přidává dodatečnou logiku k fázím životního cyklu požadavku. Interceptor implementuje protokol `RequestInterceptor`, který poskytuje nástroje potřebné pro specifikování, za jakých podmínek se má daný požadavek zkusit znovu odeslat na server.

Odepře-li server přístup k chráněným datům, aplikace automaticky obnoví AAT pomocí refresh tokenu. Obnovené tokeny jsou uloženy a původní požadavek znovu odeslán na server. Parametrem metody `completion(retryResult:)` definuji zdali se má požadavek opakovat. Předáním hodnoty výčtového typu `.retry` se požadavek odešle znovu, zatímco pro hodnotu `.doNotRetry` se proces opakování ukončí. Aby nedošlo k zacyklení algoritmu při opakovaném obdržení status kódu 401, je každý požadavek zopakován maximálně 3x.

Může nastat situace, že proces obnovy tokenu započal a je znovu vyžadován jiným požadavkem. Pro vyřešení této situace existuje proměnná `isRefreshing`, která si drží informaci o tom, zdali je token obnovován. V kladném případě je

nový požadavek zablokován. Toto řešení není ideální, neboť za předpokladu několika paralelně probíhajících požadavků, je zopakován pouze první z nich, pro zbytek je prezentována chyba. Možným řešením je kritická sekce obsahující proces obnovení tokenu, jež by byla přístupná pouze jednomu vláknu, které by sekci před začátkem operace uzamklo a po dokončením odemklo pro jiná případná vlákna.

```
func retry(_ request: Request,
           for session: Session,
           dueTo error: Error,
           completion: @escaping (RetryResult) -> Void) {
        guard let statusCode = request.response?.statusCode
        else { completion(.doNotRetry); return }
        if statusCode == 401 {
            guard !isRefreshing
            else { completion(.doNotRetry); return }
               guard request.retryCount < 3 else {
                    completion(.doNotRetry)
                    return
            }
            self.refreshTokens {
            [userSettingsAPI] (signInModel, error) in
                if error != nil {
                    completion(.doNotRetry)
                    return
                }
                guard let signInModel = signInModel else {
                    completion(.doNotRetry)
                    return
                }
                userSettingsAPI.saveCredentials(
                    credentials: signInModel.credentials
                )
                completion(.retry)
            }
        } else {
            completion(.doNotRetry)
        }
    }
```

Kód 5.1: Opakování požadavku při expiraci access tokenu

5.2 Práce s chybami

Chyby jsou běžnou součástí každé aplikace a je důležité se s nimi umět vypořádat. Ať už se jedná o chyby vzniklé špatnou implementací nebo chyby vzniklé během serverové komunikace, uživateli by vždy měly být prezentovány srozumitelnou formou. Za komunikování chyb je zodpovědný viewcontroller, který, vznikne-li problém, vytvoří `UIAlertController` s textem popisujícím vzniklou situaci. Z důvodu dodržení DRY principu jsem implementoval konstrukt, který prezentuje upozorňovací dialogy na jednom místě a je dostupný všem viewcontrollerům. Rozšířil jsem tedy třídu `UIViewController` o metodu, která dle typu předané chyby, zobrazí uživatelsky přívětivý dialog.

```
extension UIViewController {
    func handle(_ error: Error,
                retryHandler: (() -> Void)?) {
        let alert = UIAlertController(
            title: L10n.errorOccured,
            message: error.localizedDescription,
            preferredStyle: .alert
        )
        alert.addAction(UIAlertAction(
            title: L10n.cancel,
            style: .default
        ))
        switch error.resolveCategory() {
        case .retryable:
            alert.addAction(UIAlertAction(
                title: L10n.retry,
                style: .default,
                handler: { _ in
                    if let retryHandler = retryHandler {
                        retryHandler()
                    }
            }))
            break
        case .nonRetryable:
            break
        }
        present(alert, animated: true)
    }
}
```

Kód 5.2: Prezentování vyskytnuté chyby uživateli

Pro některé chyby, má smysl uživateli nabídnout možnost operaci, jež chybu způsobila, opakovat. Prostřednictvím dialogu poskytnu tlačítko, po jehož stisknutí dojde k opětovnému pokusu o provedení operace, která je předána jako jeden z parametrů metody `handle(error:retryHandler:)`. Abych rozlišil, zdali se chyby lze zbavit opakovaným spuštěním, rozřadil jsem chyby do kategorií `retryable` a `nonRetryable`.

5.3 Stránkování

Seznamy zaměstnanců, či faktur mohou obsahovat velké množství záznamů, které se vyplatí pro rychlejší zpětnou vazbu získávat a prezentovat uživateli po částech. Serverové API umožňuje stránkování rozšířením požadavku o parametry:

- **limit** - počet položek ve stránce,
- **offset** - množství položek, jež se má přeskočit.

Důležitým milníkem bylo zjistit, kdy načíst nová data. Po otevření přehledu je uživateli prezentována první stránka položek. Zobrazením poslední položky seznamu, je emitována událost o načtení další stránky.

Zprvu jsem informaci o dosažení konce seznamu získával prostřednictvím metody `reachBottom`, což bylo mé rozšíření třídy `Reactive`, které sledovalo, zdali uživatel dosáhnul spodní pozice tabulky. Pouze tehdy byla emitována žádost o načtení další stránky. Toto řešení nicméně nefungovalo správně pokud během načítání došlo k chybě, kdy se pozice nezměnila a kvůli použité funkci `distinctUntilChanged()`, se již další událost neemitovala. Proto jsem toto řešení nahradil metodou `willDisplayCell(cell:indexPath:)`, díky které lze sledovat, jaká položka je aktuálně uživateli prezentována a jedná-li se o poslední položku, je načtena další stránka.

Logiku spojenou s držením informace o pořadovém číslu aktuálně načtené stránky a načtených datech jsem delegoval na třídy `PaginationUISource` a `PaginationSink<T>` [25]. `PaginationUISource` definuje následující datové toky:

- `refresh` - emituje požadavek na obnovení seznamu,
- `loadNextPage` - emituje požadavek na načtení další stránky.

`PaginationSink<T>`, kde T je generický parametr, jenž odpovídá typu načítaných dat. Třída slouží k reagování na události obdržené z `PaginationUISource`.

Použity jsou následující proměnné:

1. `loadResults` – slovník, jehož klíčem je číslo stránky a hodnotou jsou data odpovídající stránky;
2. `currentPageNumber` – číslo nejnovější stránky;
3. `loadNext` – událost požadující načtení další stránky je transformována na číslo stránky, která je další v pořadí;
4. `reload` – událost požadující obnovení seznamu je transformována na číslo -1, abych dokázal odlišit, kdy se jedná o znovunačtení první stránky a kdy je slovník `loadResults` prázdný;
5. `start` – vytvoří ze dvou datových toků pouze jeden.

```
let loadResults = BehaviorSubject<[Int: [T]]>(value: [:])
let currentPageNumber = loadResults
            .map { $0.keys.max() ?? 1 }
let loadNext = uiSource.loadNextPage
            .withLatestFrom(currentPageNumber)
            .map { $0 + 1 }
let reload = uiSource.refresh
            .map { -1 }
let start = Observable.merge(reload, loadNext)
```

Kód 5.3: Stránkování - inicializace pomocných proměnných

Na základě proměnných definovaných v ukázce kódu 5.3 jsem vytvořil proměnnou `page`, která pomocí funkce `loadData(pageNumber:)` získává požadovanou stránku. Abych byl schopný reagovat na chyby vzniklé během serverové komunikace, materializoval jsem tento datový tok. Nebýt funkce `share()`, každý nový pozorovatel proměnné `page` by odeslal nový požadavek na server.

```
let page = start
            .flatMap { pageNumber in
                loadData(pageNumber == -1 ? 1 : pageNumber)
                .map{ (pageNumber: pageNumber, items: $0) }
                .materialize()
                .filter { $0.isCompleted == false }}
            .share()
```

Kód 5.4: Stránkování - získání nové stránky

Kód 5.5 se stará o přidání nově obdržené stránky do slovníku `loadResults`.

```
_ = page.compactMap { $0.element }
        .withLatestFrom(loadResults) { (pages: $1, newPage: $0) }
        .filter { $0.newPage.pageNumber == -1
                || !$0.newPage.items.isEmpty }
        .map { $0.newPage.pageNumber == -1 ?
            [1: $0.newPage.items]
            : $0.pages.merging([$0.newPage],
                               uniquingKeysWith: { $1 })}
        .subscribe(loadResults)
```

Kód 5.5: Stránkování - uložení nové stránky do slovníku

Na závěr dochází k definici veřejných proměnných, jejichž události lze pozorovat ve viewmodelu.

- `isLoading` – signalizuje průběh načítání stránkovaných dat;
- `elements` – emituje všechny doposud načtené záznamy;
- `error` – pokud dojde k chybě během síťové komunikace, tak ji emituje.

```
self.isLoading = Observable.merge(start.map { _ in 1 },
                                  page.map { _ in -1 })
            .scan(0, accumulator: +)
            .map { $0 > 0 }
            .distinctUntilChanged()
self.elements = loadResults
            .map { $0.sorted(by: { $0.key < $1.key })
            .flatMap { $0.value }}
self.error = page
            .map { $0.error }
            .compactMap{ $0 }
```

Kód 5.6: Stránkování - aktualizace veřejných proměnných

Problémem tohoto řešení je jeho špatná rozšiřitelnost. Je-li potřeba stránkovat a současně filtrovat záznamy seznamu, musí se vytvořit nová implementace tříd `PaginationUISource` a `PaginationSink<T>`. Například pro filtrování seznamu zaměstnanců jsem vytvořil třídu `PaginationWithSearchSink<T>`, která kromě stránkování navíc reaguje na změnu vyhledávaného jména zaměstnance.

5.4 Zobrazení grafu

Aplikace poskytuje uživateli vizuální náhled na výši jeho finanční odměny během uplynulých měsíců pomocí sloupcového grafu. Z počátku jsem zamýšlel graf vytvořit sám bez pomoci externích knihoven. Avšak kvůli velké komplexitě jsem nakonec upřednostnil použití externí knihovny Charts, která generování grafu velmi zjednodušuje.

Z faktury se extrahuje její hodnota a vykázané období, které se namapují na třídu `BarChartDataEntry`, která pomocí dvourozměrného souřadnicového systému definuje sloupec grafu.

Pro zobrazení grafu slouží komponenta `BarChartView`. Pole instancí třídy `BarChartDataEntry` se musí obalit do dvou dalších tříd předtím nežli je možné jej poskytnout UI komponentě. Výsledkem je univerzální zpracování grafu, které svou podobou nezapadá do zbytku aplikace. Graf jsem tedy ostyloval.

Požadavek	Kód

Skrýt horizontální a vertikální linky	`rightAxis.enabled = false`
	`leftAxis.enabled = false`
	`xAxis.drawAxisLineEnabled = false`
	`xAxis.drawGridLinesEnabled = false`

Skrýt legendy	`legend.enabled = false`
Umístit popis osy x pod graf	`xAxis.labelPosition = .bottom`
Upravit barvu nezvoleného sloupce grafu	`colors = [Asset.Colors.chartBarDefault.color]`
Upravit barvu zvoleného sloupce grafu	`highlightColor = Asset.Colors .chartBarHighlighted.color`
Skrýt hodnoty sloupců	`setDrawValues(false)`
Pro prezentování sloupců vytvoř animaci	`animate(xAxisDuration: 1.5, yAxisDuration: 1.5, easingOption: .easeInQuad)`

Tabulka 5.1: Atributy pro změnu stylu sloupcového grafu

Tabulka 5.1 znázorňuje úpravy provedené na grafu. Po provedení těchto změn, graf barevně ladí se zbytkem aplikace, nicméně popisky sloupců jsou pořadová čísla, nikoli názvy období. Pro změnu popisku sloupce, musí osa x specifikovat `valueFormatter`, jenž umí pro daný sloupec získat a zobrazit období, pro které byla faktura vystavena.

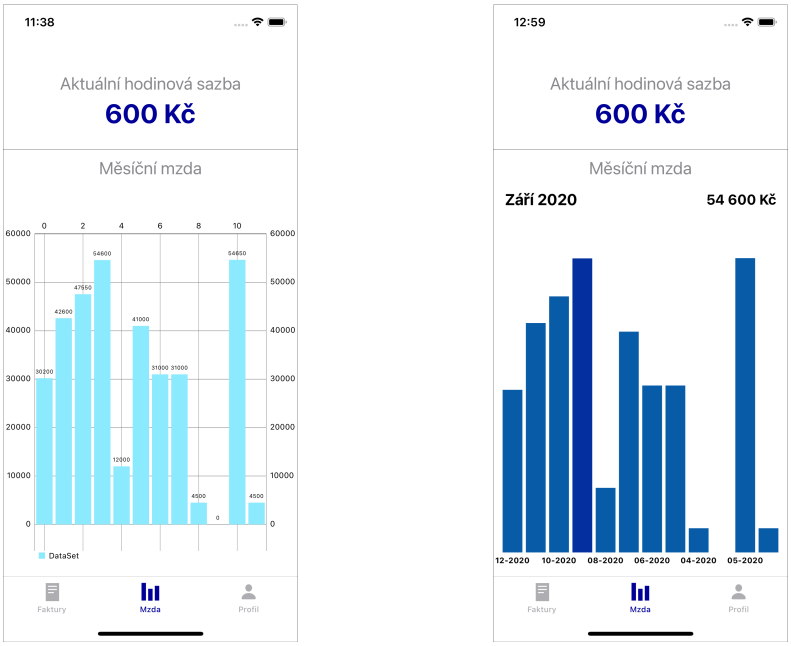
Jelikož jsem skryl hodnoty sloupců, není možné zjistit konkrétní výši finanční odměny za uvedený měsíc. Proto chci, aby se po označení sloupce, nad grafem zobrazilo zvolené období a jemu odpovídající hodnota. Toho lze dosáhnout za pomoci `ChartViewDelegate` a metody `chartValueSelected(chartViewBase:`

`chartDataEntry:highlight:`). V rámci které přidávám text do dvou `UILabel` komponent reprezentujících požadované hodnoty. O této funkci se ovšem uživatel nedozví, dokud neklikne na některý ze sloupců. Proto je po načtení grafu automaticky zvolen aktuální měsíc.

Bude-li uživatel zaměstnaný delší dobu, graf se díky velkému množství zobrazených sloupců může stát nepřehledným. Implementoval jsem proto animaci, která po načtení zobrazí graf ve své úplnosti a poté se přesune na zvolený počet (aktuálně 5) posledních měsíců.

```
func animateChart(){
        let delayScaling = durationOfInitialAnimation
        var delayHighlight = durationOfInitialAnimation
                             + durationOfScalingAnimation + 0.5

        let val = Double(chartView.barData.entryCount)
                  / Double(self.visibleBarsTrashHold)

        if val > 1.0 {
            DispatchQueue.main.asyncAfter(
                deadline: .now() + delayScaling) {
                [weak self, val] in
                guard let self = self else {return}

                let scaleX = CGFloat(val)
                chartView.zoomAndCenterViewAnimated(
                    scaleX: scaleX, scaleY: 1,
                    xValue: 0, yValue: 0,
                    axis: .left,
                    duration: durationOfScalingAnimation,
                    easingOption: .easeInCubic)
            }
        } else { delayHighlight = durationOfInitialAnimation }

        DispatchQueue.main.asyncAfter(
            deadline: .now() + delayHighlight) {
            [weak self] in
            chartView.highlightValue(x: 0, dataSetIndex: 0)
        }
    }
```

Kód 5.7: Graf - animovaný přechod



(a) Graf před úpravami (b) Graf po úpravách

Obrázek 5.1: Panelová karta se mzdou zaměstnance

5.5 Knihovny použité pro práci s kódem

SwiftLint Velké množství kódu s sebou přináší i velké množství potencionálních chyb. Nemusí se jednat pouze o chyby funkční, nýbrž také o chyby spojené s kvalitou kódu. Pro vývoj udržitelné aplikace je zapotřebí dodržovat sadu definovaných principů, mezi které patří například DRY, stanovená maximální komplexita tříd a další. Na některé chyby upozorňuje sám Xcode. Pro přísnější analýzu kódu jsem použil knihovnu SwiftLint. Pravidla jsou definována v konfiguračním souboru, jehož podobou jsem se inspiroval u šablony poskytované zadavatelskou společností.

SwiftGen Výchozí možností jak specifikovat zdroje (obrázky, barvy, lokalizovatelné texty, apod.) je použitím textového identifikátoru. Za předpokladu, že se název zdroje změní nebo přestane existovat, je časově náročné zjistit místa, kde byl zdroj použit a tato místa opravit. Pro vyřešení tohoto problému jsem zvolil knihovnu SwiftGen, která automaticky vygeneruje Swift kód pro všechny definované zdroje v projektu. Díky tomu můžu použít místo `UIColor(named: "chart-highlighted")!` staticky definovanou proměnnou `Asset.Colors.chartHighlighted.color`.

Kapitola 6

Testování

Tato kapitola prezentuje praktiky použité při tvorbě jednotkových a UI testů, které vznikly během vývoje. Po dokončení byla aplikace otestována skupinou dobrovolníků. Průběh i výsledky testů jsou předmětem této kapitoly.

6.1 Automatizované testy

Mým cílem bylo aplikaci zevrubně otestovat jednotkovými a UI testy, kterými bych mohl kontrolovat, že hlavní funkcionality fungují i po provedených změnách.

6.1.1 Jednotkové testy

Testy musí být standardizované, udržitelné a vždy musí testovat pouze izolovanou část kódu. Pro testování jsem zvolil nativní XCTest framework, kdy všechny metody implementují GWT strukturu, čímž maximalizuji přehlednost a ulehčím debugování.

Stran pojmenování lze uvést, že se mi v minulosti v praxi osvědčilo pojmenovávání veškerých testovacích metod na základě strukturovaného formátu typu „JménoMetody_VstupníParametry_OčekávanýVýstup“, a to pokud je testovaný blok větvený a má tedy více scénářů. Jestliže některý z testů selže, vývojář má okamžitou zpětnou vazbu, ve které vidí, který scénář selhal. Také pomáhá udržet test coverage na maximu, neboť je vývojář nucen si uvědomit možné vstupy a výstupy testovaných funkcí. Tento formát se pro mě stal nedílnou součástí každodenní praxe, zejména vyvíjím-li pomocí TDD.

6.1.2 UI testy

Po provedení větších změn v implementaci jsem pokaždé musel zkontrolovat, zdali uživatelské rozhraní správně reaguje. Tato neustálá validace byla časově

náročná, a proto jsem implementoval UI testy, které tyto úkoly provedly samostatně.

Aplikace je závislá na datech ze serveru, což pro mé testy představovalo potencionální problém, pokud by server byl nedostupný nebo by se podoba obdržených dat změnila. Pomocí knihovny OHHTTPStubs se mi podařilo nahradit odpovědi z koncových bodů serveru vlastními daty. Další překážkou byl proces přihlašování, který přináší jistý stochastismus. Aby byly testy nezávislé a opakovatelné, musel bych před každým testem aplikaci uvést do továrního nastavení. Kromě toho by se každý test stal závislým na externí knihovně GoogleSignIn, na kterou deleguji přihlašovací proces. Proto každý UI test může spustit aplikaci s přepínači:

- **-mockUser** – koncové body relevantní pro roli user jsou nahrazeny,
- **-mockTeamLeader** – koncové body relevantní pro roli team leader jsou nahrazeny,
- **-mockAdmin** – koncové body relevantní pro roli admin jsou nahrazeny,
- **-withoutGoogleSignIn** – přihlašování přes Google účet je vypnuto.

Podle předaných přepínačů se po spuštění aplikace ve třídě `SceneDelegate` nahradí odpovídající logika. UI testy jsem vytvořil pro všechny hlavní průchody aplikací. Pro ilustraci uvádím dva z nich.

Přihlášení pomocí Google účtu Test spustí aplikaci a zkusí najít panelovou kartu s názvem „Profil“, pokud ji najde, je zřejmé, že uživatel byl v minulosti přihlášen a pro pokračování testu je nutné jej odhlásit. Dalším krokem je otevření dialogu s Google přihlášením. Test vyplní jméno a heslo uživatele Test User a přihlásí se. Podaří-li se uživatele po přihlášení znovu odhlásit, test proběhl úspěšně.

Faktury zaměstnance

1. Test se přihlásí pod účtem s rolí zaměstnanec.
2. Je zobrazeno 10 faktur (leden 2020 – říjen 2020).
3. Test zkontroluje, že přehled neobsahuje fakturu za prosinec 2019.
4. Test se posune dolů na konec seznamu.
5. Aplikace načte dalších 10 faktur.

6. Test zkontroluje, že přehled nově obsahuje fakturu za prosinec 2019.
7. Test obnoví seznam.
8. Test zkontroluje, zdali přehled již neobsahuje fakturu za prosinec 2019.

6.2 Uživatelské testy

Po dokončení hlavní implementační části jsem aplikaci nechal otestovat skupinou dobrovolníků, kteří se vžili do role skutečných uživatelů aplikace.

Uživatelům byla zprvu aplikace prezentována jakožto pracovní projekt třetí strany, který jsem dostal za úkol otestovat. Skutečná povaha projektu jim byla prozrazena až po skončení experimentu, čímž jsem eliminoval jejich případnou zaujatost.

Testování se zúčastnili tři testeři, kteří mají dlouholetou zkušenost s používáním iOS zařízení. Byla jim vysvětlena povaha projektu, představena jejich fiktivní role a povinnosti, které s sebou jejich nová role přináší. Zprvu se ocitli v roli vedoucího týmu (team leader). Před začátkem samotného testování byli dobrovolníci detailně seznámeni se scénářem a průběhem testování, abych již dále nikterak neovlivňoval výsledky experimentu. Dobu vypracování jednotlivých částí scénáře jsem měřil a uživatelskou interakci zaznamenával. Záznam je dostupný v přílohách této diplomové práce. Pro sběr výsledků jsem zvolil způsob „Story telling“. Po dokončení každé položky scénáře tester zaznamenal své názory do dotazníku.

Jakmile tester dokončil scénář pro vedoucího týmu, byl nově zasazen do role admina a byl mu představen nový scénář. Zbytek experimentu probíhal totožně jako u předchozí role.

6.2.1 Scénáře

Scénáře jsou rozděleny dle role uživatele, který je provádí.

Vedoucí týmu

1. Přihlaš se do aplikace. Tvůj email je test.teamLeader@ack.ee a heslo: ******.
2. Je konec měsíce a ty musíš vykázat fakturu, kterou chceš, aby ti společnost uhradila. Nahraj fakturu za prosinec 2020. Jedná se o soubor s názvem „faktura-prosinec.pdf“.
3. Nemůžeš na svém bankovním účtu najít výplatu za leden. Zjisti, jaké číslo účtu jsi uvedl ve faktuře za leden 2020.

4. Bohužel jsi ve faktuře vykázané za prosinec 2020 vyplnil špatné číslo účtu. Nahraj novou fakturu, tentokrát soubor „faktura-corrected.pdf“.

5. Máš před sebou roční schůzku se svým nadřízeným. Zjisti, jaká je tvá aktuální hodinová sazba, abys věděl, o jaké navýšení si můžeš říct.

6. Karel Novák, člen tvého týmu, je aktuálně na měsíční dovolené. Bohužel zapomněl před odchodem předat návrh PPC vizuálu pro novou kampaň. Nezbývá ti nežli mu zavolat na osobní telefon. Zjisti telefonní číslo Karla Nováka.

7. Přešel jsi na nového operátora, který ti neposkytnul možnost zachovat si stávající číslo. Sdělil jsi pracovníku HR, aby ti změnil telefonní číslo v osobních údajích. Už je to pár dní a tebe zajímá, zdali bylo telefonní číslo změněno. Zkontroluj své telefonní číslo.

Admin

1. Spolupracovník si půjčil tvůj telefon, aby zjistil emailovou adresu jednoho ze členů svého týmu. Zapomněl se však odhlásit. Přihlaš se do aplikace. Tvůj email je test.admin@ack.ee a heslo: ******.

2. Je konec měsíce a na tebe čekají dvě faktury. Zkontroluj, že zaměstnanec Jakub Novák vykázal práci spojenou pouze s tvorbou nové kampaně pro aplikaci Miláčci a Pavel Novák strávil prací 160 hodin. Pokud údaje na faktuře sedí, označ ji za schválenou. Jinak ji vrať zpět zaměstnanci.

3. Jeden ze zaměstnanců si stěžuje, že mu nepřišly za září peníze na účet. Zjisti, zdali Karel Novák nahrál za září 2020 fakturu.

4. Firemní účetní ti poslal email, že byly vytvořeny příkazy k úhradě všech schválených faktur za prosinec 2020. Označ všechny schválené faktury za zaplacené.

5. Karel Novák letos dostudoval magisterské studium a stal se inženýrem. Při ročním sezení s Karlem jste se domluvili, že se mu od 1. února 2021 navýší hodinová sazba o 20 %. Uprav titul Karla Nováka a nastav mu od 1. února 2021 o 20 % vyšší hodinovou sazbu.

6. Tvůj nadřízený chce vědět, o kolik se průměrně navýšily hodinové sazby u zaměstnanců za září 2020. Zjisti průměrnou hodnotu navýšení hodinové sazby za toto období.

6.2.2 Dotazník

Po splnění každého bodu scénáře, tester odpověděl na následující tři otázky:

- Narazil jste na něco, co se Vám při splňování úkolu líbilo?
- Setkal jste se při výkonu úkolu s problémem, který Vás frustroval?
- Napadá Vás, jak by šel proces vylepšit?

6.2.3 Výsledky testování

Podrobné záznamy o výsledcích testování jsou k dispozici v příloze. Pro lepší přehlednost uvádím pouze problémy, které testeři identifikovali u jednotlivých kroků výše popsaných scénářů.

Vedoucí týmu

- **Úkol 2**: Nelze rozlišit jaký je rozdíl mezi obdobími, které jsou bez nebo s nahranou fakturou.
- **Úkol 4**: Málo intuitivní umístění možnosti znovu nahrát fakturu.

Admin

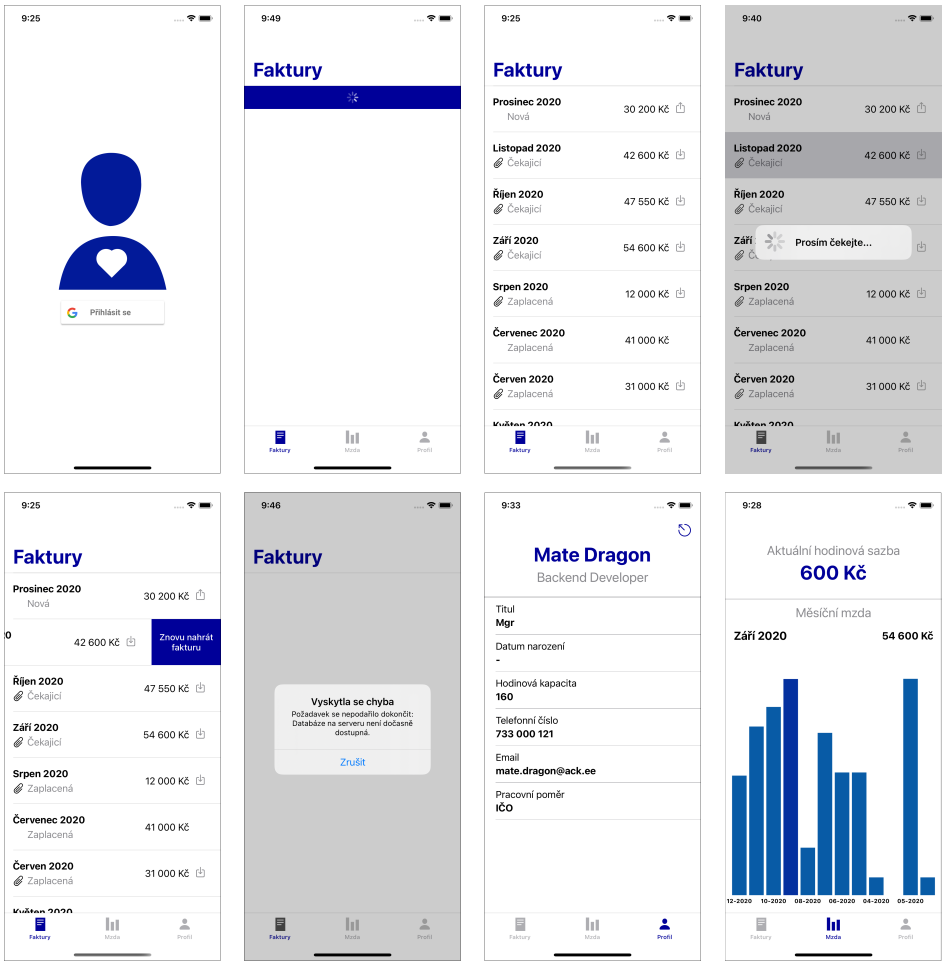
- **Úkol 2**: Umístění akcí dostupných pro fakturu není intuitivní. Dostupné akce pro fakturu jsou popsány stavem, do kterého se má po jejich stisknutí přejít, místo popisu akce, která se provede (místo „Nová“ použít „Vrátit zpět“).
- **Úkol 3**: Chybí filtrování faktur dle jména zaměstnance.
- **Úkol 4**: Možnost označit více faktur současně.
- **Úkol 5**: Velikost stisknutelného pole pro nastavení hodinové sazby není dostatečná. Místo fajfky pro uložení, raději ikonu diskety.
- **Úkol 6**: Šipka pro volbu období není klikatelná.

6.2.4 Identifikované problémy

Vyhodnocením dotazníkových bloků testerů jsem identifikoval následující problémy a způsoby, jak je vyřešit.

Problém	Řešení
Akce pro vrácení faktury zaměstnanci je pojmenována stavem „Nová“.	Místo názvu stavu, do kterého se faktura dostane po stisknutí, je akce pojmenována „Vrátit zpět“.
Akci „znovu nahrát fakturu“ je obtížné najít.	Přidání kontextového menu po dlouhém stisknutí řádky seznamu.
Šipka dolů u rozbalovací nabídky pro volbu období není stisknutelná.	Celá komponenta pro volbu období musí být stisknutelná.

Tabulka 6.1: Způsoby řešení problémů odhalených během uživatelského testování



Obrázek 6.1: Finální podoba aplikace pro roli zaměstnanec

Závěr

Cílem této diplomové práce bylo vytvořit mobilní aplikaci, která ulehčí správu zaměstnanců v malé či střední firmě. Jedná se zejména o procesy spjaté se správou faktur a osobních údajů zaměstnanců.

Nejprve jsem v rámci rešerše prozkoumal praktiky a technologie, které jsou asociovány s vývojem aplikací pro iOS, ze kterých jsem v rámci analýzy vybral ty, které nejvíce vyhovují mým účelům. Díky rozhovorům se zadavatelskou společností a analýze wireframů webového portálu jsem určil požadavky kladené na vyvíjenou aplikaci.

Než jsem začal s implementací, navrhnul jsem, jak bude vypadat uživatelské rozhraní, k čemuž jsem zvolil standardizované postupy, které jsou v práci detailně popsány. Následoval návrh architektury aplikace, který odpovídá praktikám, technologiím a požadavkům zvoleným v rámci analýzy. Nejdříve jsem vytvořil obecnou strukturu, která je společná pro všechny důležité části aplikace. Jedná se o jednotnou formu prezentační vrstvy, způsobu autorizace uživatele nebo komunikace se serverem. Od obecně platných funkcionalit jsem přešel ke konkrétnímu návrhu jednotlivých logických celků, jenž jsem použil při realizaci samotné aplikace.

Kapitola implementace pojednává o zajímavých problémech, se kterými jsem se při realizaci projektu setkal a jakým způsobem jsem se s nimi vypořádal. Uživatelské rozhraní bylo otestováno pomocí heuristického a uživatelského testování. Správnost implementovaných funkcionalit byla ověřena sadou jednotkových a UI testů.

Definované funkční i nefunkční požadavky jsem splnil a rozšířil jsem je o přehled finančních odměn zaměstnance vizualizovaných pomocí sloupcového grafu. Přestože jsem příznivce objektově orientovaného programování musím přiznat, že funkcionální konstrukce dokážou být čitelnější a lépe udržitelné. Ačkoli

mi FRP povaha kombinace Swiftu a RxSwift neumožnila vždy použít GoF návrhové vzory, jsem rád, že se mi jich podařilo pár uplatnit (Factory, Singleton, Chain of Responsibility, Template Method, Observer a další).

Tvorba UI ve storyboardu není bohužel vhodným řešením, neboť se obrazovky nedají recyklovat. Tento nedostatek by vyřešilo, kdybych vytvářel UI v kódu. Vezmu-li však v potaz, že se jedná o mou první iOS aplikaci a první zkušenost se Swiftem, storyboard mi definitivně pomohl v rychlejší adaptaci na novou platformu. Ukládání uživatelského access tokenu do UserDetails nebylo dobrou volbou, neboť pro uchovávání citlivých dat se více hodí Keychain. Díky použitým protokolům a abstrakci funkce ukládání uživatelských detailů by však tato změna neměla být nijak časově náročná na implementaci.

Projekt by mohl být dále rozšířen o správu dovolených a push notifikace, které by upozornily zaměstnance, že ještě nenahrál fakturu. Během vývoje jsem si vytvářel seznam „nice to have“ funkcionalit, které jsem ve volných chvílích implementoval. Mezi ty, které by bylo možné ještě přidat, patří například ukazatel průběhu nahrávání, stahování souboru reprezentujícího fakturu nebo použití knihovny ACKLocalization pro lokalizaci aplikace.

Hlavním přínosem této diplomové práce byla pro mě možnost rozšířit si povědomí o programovacích technikách spojených s platformou iOS. Usvědčilo mě to v názoru, že se svět netočí kolem OOP a stojí za to, rozšířit si obzory i v jiných programovacích paradigmatech, které v mnoha oblastech poskytují kvalitnější nástroje.

Literatura

[1] Saccomani, P.: Native Apps, Web Apps or Hybrid Apps? What's the Difference? *MobiLoud*, [cit. 2020-12-15]. Dostupné z: `https://www.mobiloud.com/blog/native-web-or-hybrid-apps`

[2] Blair, I.: Objective C vs. Swift: Which is Better? (A Definitive Guide). *buildFire*, [cit. 2020-12-15]. Dostupné z: `https://buildfire.com/objective-c-or-swift`

[3] Karczewski, D.: Swift vs Objective-C: Which Should You Pick For Your Next iOS Mobile App? *Ideamotive*, [cit. 2020-12-15]. Dostupné z: `https://www.ideamotive.co/blog/swift-vs-objective-c-which-should-you-pick-for-your-next-ios-mobile-app`

[4] Hillegass, A.; Keur, C.: *iOS Programming: The Big Nerd Ranch Guide*. Atlanta: Big Nerd Ranch Guides, 7 vydání, 2020, ISBN 9780135264843.

[5] Laso-Marsetti, D.: Model-View-Controller (MVC) in iOS – A Modern Approach. *Ray Wenderlich*, [cit. 2020-12-15]. Dostupné z: `https://www.raywenderlich.com/1000705-model-view-controller-mvc-in-ios-a-modern-approach`

[6] Oulladi, S. E.: iOS Swift : MVP Architecture. *Medium*, [cit. 2020-12-15]. Dostupné z: `https://saad-eloulladi.medium.com/ios-swift-mvp-architecture-pattern-a2b0c2d310a3`

[7] Katz, M.: Getting Started with the VIPER Architecture Pattern. *Ray Wenderlich*, [cit. 2020-12-15]. Dostupné z: `https://www.raywenderlich.com/8440907-getting-started-with-the-viper-architecture-pattern`

[8] Toal, R.: Programming Paradigms. *LMU Blog*, [cit. 2020-12-17]. Dostupné z: `https://cs.lmu.edu/~ray/notes/paradigms`

[9] Singh, N.: *Reactive Programming with Swift 4*. Birmingham: Packt Publishing, 2018, ISBN 9781787120211.

[10] Nyisztor, K.: Protocol Oriented Programming in Swift. *PluralSight*, [cit. 2020-12-17]. Dostupné z: https://www.pluralsight.com/guides/protocol-oriented-programming-in-swift

[11] Community, R.: *RxSwift*. 2020, [cit. 2020-12-17]. Dostupné z: https://github.com/ReactiveX/RxSwift/blob/main/Documentation/GettingStarted.md

[12] Community, R. S.: *Reactive Swift*. 2019, [cit. 2020-12-17]. Dostupné z: https://github.com/ReactiveCocoa/ReactiveSwift/blob/master/Documentation/ReactivePrimitives.md

[13] Apple: *URLSession*. 2020, [cit. 2020-12-17]. Dostupné z: https://developer.apple.com/documentation/foundation/urlsession

[14] Community, M.: *Moya*. 2019, [cit. 2020-12-17]. Dostupné z: https://github.com/Moya/Moya/blob/master/docs/Examples/Basic.md

[15] Veselý, D.: Programování uživatelského rozhraní na iOS v Ackee (Storyboards vs. Xib vs. kód). *Ackee*, [cit. 2020-12-15]. Dostupné z: https://www.ackee.cz/blog/programovani-uzivatelskeho-rozhrani-na-ios-v-ackee-storyboards-vs-xib-vs-kod

[16] Laso-Marsetti, F.: Coordinator Tutorial for iOS: Getting Started. *Andrew Kharchyshyn*, 2018, [cit. 2020-12-17]. Dostupné z: https://www.raywenderlich.com/158-coordinator-tutorial-for-ios-getting-started

[17] Khanlou, S.: Back Buttons and Coordinators. *KHANLOU*, 2017, [cit. 2020-12-17]. Dostupné z: https://khanlou.com/2017/05/back-buttons-and-coordinators

[18] Jabrayilov, M.: Navigation with Flow Controllers. *Swift with Majid*, 2019, [cit. 2020-12-17]. Dostupné z: https://swiftwithmajid.com/2019/02/20/navigation-with-flow-controllers

[19] Parecki, A.: OAuth 2 Simplified. *Aaron Parecki*, 2020, [cit. 2020-12-17]. Dostupné z: https://aaronparecki.com/oauth-2-simplified/#making-authenticated-requests

[20] Newton, D.: Writing Your F.I.R.S.T Unit Tests. *Dzone*, 2017, [cit. 2020-12-17]. Dostupné z: https://dzone.com/articles/writing-your-first-unit-tests

[21] Todorov, M.: Testing with RxBlocking, part 1. *rx-martin*, 2017, [cit. 2020-12-17]. Dostupné z: `http://rx-marin.com/post/rxblocking-part1`

[22] Sundell, J.: Getting started with Xcode UI testing in Swift. *Swift By Sundell*, 2017, [cit. 2020-12-17]. Dostupné z: `https://www.swiftbysundell.com/articles/getting-started-with-xcode-ui-testing-in-swift`

[23] Support, A. S.: *iOS and iPadOS Usage*. 2020, [cit. 2020-12-20]. Dostupné z: `https://developer.apple.com/support/app-store`

[24] Messaoudi, A.: Write a Networking Layer in Swift 4 using Alamofire 5 and Codable Part 1: API Router. *Medium*, 2017, [cit. 2020-12-26]. Dostupné z: `https://medium.com/@AladinWay/write-a-networking-layer-in-swift-4-using-alamofire-and-codable-part-1-api-router-349699a47569`

[25] Bhanushali, A.: *RxSwift MVVM*. 2018, [cit. 2020-12-28]. Dostupné z: `https://github.com/ajaybhanushalid2k/RxSwift_MVVM`

Příloha **A**

Seznam použitých zkratek

MVC Model, View, Controller

MVP Model, View, Presenter

MVVM Model, View, View Model

VIPER View, Interactor, Presenter, Entity, Routing

UI User Interface

API Application Programming Interface

HIG Human Interface Guidelines

NPE Null Pointer Exception

IB Interface Builder

PDF Portable Document Format

XLSX XML Spreadsheet

IČO Identifikační číslo osoby

TDD Test Driven Development

DRY Do Not Repeat Yourself

GoF Gang of Five

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `user-interface` ..... adresář souborů spojených s tvorbou UI
  - `user-testing` ..... adresář se záznamy o uživatelském testování
  - `prototyp` ..... Invision prototyp UI
- `enterprise-architect` ... návrh projektu v aplikaci Enterprise Architect
- `src`
  - `app` ..... zdrojový kód implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS