FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Mobilní aplikace pro cestovatelský deník
Student:	Bc. Jan Steuer
Vedoucí:	Ing. Tomáš Nováček
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Cílem práce je vytvořit mobilní aplikaci (frontend i backend), která poskytne uživatelům rozhraní pro plánování a správu turistických, respektive služebních cest.

Proveďte následující kroky:
1) Analyzujte dostupné aplikace na trhu zabývající se podobným tématem.
2) Proveďte návrh a implementaci vlastního řešení, kde
a) serverová část bude obsluhovat požadavky architektonickým stylem REST,
b) mobilní aplikace pro Android bude se serverem komunikovat.
3) Implementace mobilní aplikace by měla podporovat organizaci cest mezi více uživateli.
4) Proveďte uživatelské testy.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 7. října 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Mobilní aplikace pro cestovatelský deník

Bc. Jan Steuer

Katedra softwarového inženýrství
Vedoucí práce: Ing. Tomáš Nováček

20. června 2022

Poděkování

Velice rád bych poděkoval vedoucímu mé diplomové práce Ing. Tomáši Nováčkovi za cennou zpětnou vazbu, korekturu a aktivní přístup během tvorby mé diplomové práce. Také děkuji všem účastníkům uživatelského testování.

Prohlášení

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

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

V Praze dne 20. června 2022 . . . . . .

Odkaz na tuto práci

Steuer, Jan. *Mobilní aplikace pro cestovatelský deník*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Tato diplomová práce se zabývá vývojem mobilní aplikace, která poskytuje uživatelům rozhraní pro společné plánování cest. O zajištění funkcí pro organizaci a správu výletů mezi účastníky se stará webová služba, která komunikuje s mobilní aplikací pomocí architektonického stylu REST. Autor v této práci postupně popisuje jednotlivé fáze vývoje obou částí – backendu i mobilní aplikace. Zároveň podrobněji rozebírá návrh a implementaci autentizace uživatelů pomocí obnovování JWT tokenů. Výstupem této diplomové práce je funkční a uživatelsky otestovaná mobilní aplikace pro organizaci cest.

Klíčová slova mobilní aplikace, backend, REST API, společné plánování cest, Android, Kotlin, Ktor, JWT

Abstract

This diploma thesis deals with the development of a mobile application that provides users with an interface for group travel planning. The web service communicates with the mobile application using the REST architectural style and ensures the actual functionality of organizing and managing trips. In this work, the author describes step by step the various stages of development of both parts – the backend and the mobile application. The author also covers the design and implementation of user authentication accomplished by refreshing JWT tokens in this work. The output of this diploma thesis is a functional and user-tested mobile application that allows organizing trips in a group.

Keywords mobile application, backend, REST API, group travel planning, Android, Kotlin, Ktor, JWT

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Úvod

S kamarády jezdím o prázdninách na vodu. Zkušenější vodáci naplánují řeky ke splutí a místa, kde budeme spát. Nemají to ale jednoduché. Obtížnost splutí jediné řeky se v závislosti na průtoku velmi liší. Také musí vymyslet místa pro nástup a výstup. Místa, kde se dá bezpečně vylodit, ale zároveň poblíž zaparkovat s auty.

Jednotlivé řeky mohou být od sebe vzdáleny desítky až stovky kilometrů. Ne každá řeka je krásná a ne všechny řeky jsou vhodné pro naší úroveň kajakování. Během jedné dovolené tak často projedeme celou zemi.

Protože nás většinou bývá více, naši výpravu tvoří několik posádek se svými auty. Řidiči se musí vždy před jízdou domluvit na společném postupu.

„Kam teda jedeme?“ ptá se Lukáš.

„Však jsem ti to už říkal, dneska končíme u Salzy,“ odpovídá Tomáš, náš nejzkušenější vodák.

„A ten kemp, kde spíme, se jmenuje jak?“

„Camp Campanula. Ale po cestě se ještě budeme stavovat na nákup v Bille,“ ukazuje Tomáš na mapě.

Během jízdy volá Šimon, člen druhé posádky: „Čau Tome, kde je ta Billa, u které se máme sejít?“

Domluva mezi tolika lidmi, zvláště když je plán cesty složitější, bývá občas komplikovaná. Pamatovat si všechna nástupní a výstupní místa, kempy, aktivity po cestě. Kdo v kterém autě platil za benzín a jakou částku?

Následující den sedíme v kolečku u stanu a přesouváme si mobil z ruky do ruky. Tomáš na něm má fotku výstupního místa. „Tady budeme vystupovat, ale pozor, vemte to zprava, protože vlevo může být vír.“ Pak se zadívá do svého sešitku z loňského roku a zkontroluje, jaký byl tehdy průtok: „Tak by tam ten vír asi být neměl. Dívám se kolik vody teklo před rokem a letos je to ještě větší bída.“

Naplánovat takovou dovolenou není jednoduché, zvláště, když se občas mění plány za pochodu. Ne vždy se podaří dostat novou informaci ke všem

členům skupiny. Pak nastávají zbytečné komplikace, které kazí zážitek z cesty.

Cíl práce

Cílem této diplomové práce je vytvořit mobilní aplikaci pro Android, která poskytne uživatelům rozhraní pro společné plánování cest. O zajištění funkcí pro organizaci a správu výletů mezi účastníky se postará webová služba, jež bude součástí výstupu praktické části této práce. Kvůli snadné dostupnosti bude webová služba nasazena na cloudu. Mobilní aplikace pak bude s touto službou komunikovat pomocí architektonického stylu REST a zobrazovat výsledky uživatelům.

Členění práce

Práce je rozdělena do několika samostatných kapitol. V první kapitole seznamuje čtenáře s již existujícími mobilními aplikacemi na trhu, které poskytují svým uživatelům funkce srovnatelné s tématem této diplomové práce.

Druhá kapitola se zabývá analýzou požadavků. Přesněji rozebírá jednotlivé funkční a nefunkční požadavky a zaobírá se cílovými uživateli aplikace.

V pořadí třetí kapitola slouží pro přiblížení vybraných technologií při návrhu backendu. Dále popisuje zvolenou architekturu a doplňuje text o databázový model a řešení autentizace. V závěru pak představuje vybrané body z implementace webové služby.

Předposlední kapitola seznamuje čtenáře s mobilní aplikací. V úvodu rozebírá návrh uživatelského rozhraní. Poté vysvětluje doporučené rozvržení Android aplikací s konkrétními příklady. Na konec představuje čtenáři vybrané technologie a principy užité během vývoje aplikace.

Poslední kapitola popisuje průběh uživatelského testování a doplňuje tuto sekci o způsob vzdáleného sledování pádů mobilní aplikace. V samotném závěru se pak čtenář může dočíst o vyhodnocení uživatelského testování.

Kapitola 1

Přehled vybraných cestovatelských aplikací

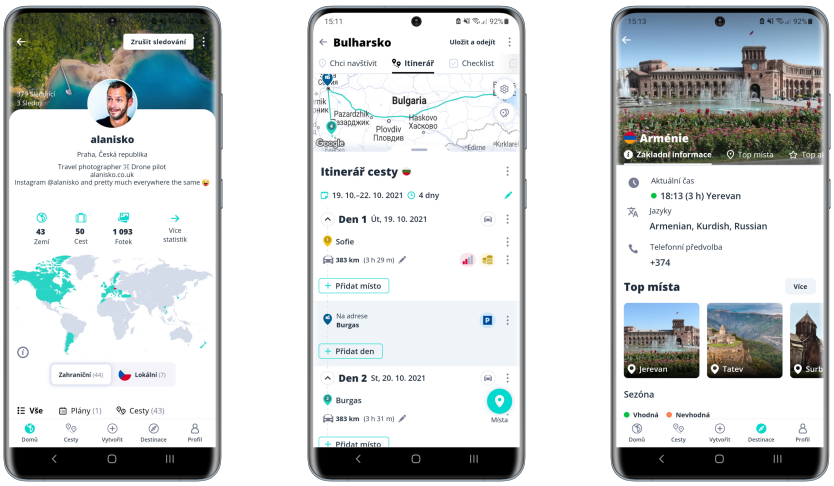
V této kapitole se věnuji popisu pěti mobilních aplikací dostupných na Google Play[1], které nabízí svým uživatelům funkce srovnatelné s tématem této diplomové práce. Zkoumané aplikace umožňují uživatelům vytvořit itinerář cesty a sdílet jej s ostatními členy. Mimo to poskytují další pokročilé funkce, ať už se jedná o cestovatelského průvodce, doporučení okolních bodů zájmů, statistiky či jiné. V kapitole objasňuji čtenáři charakteristické rysy jednotlivých aplikací a porovnávám jejich nabízené služby. V neposlední řadě si dovoluji subjektivně vyjádřit mé dojmy a pocity z používání neplacených částí hodnocených aplikací. Cílem této kapitoly je zamyšlení se nad způsobem řešení dostupných alternativ, které poté budu moci využít během implementace vlastní aplikace.

1.1 Worldee

V srpnu 2017 si Tomáš Zapletal uvědomil fakt, že mu na trhu chybí jednoduchý nástroj pro organizaci cest po světe, který by mu zároveň posloužil jako jeho cestovatelský deník. Neuplynuly ani dva roky od původního nápadu a první verze webové služby – jež kombinovala prvky stírací mapy světa a fotek z výletů – byla oficiálně spuštěna. O rok později vývojáři tohoto českého startupu přidali do aplikace plánovač cest a v červnu 2021 vydali zbrusu novou mobilní aplikaci. [1] Ta se skládá se tří hlavních komponent:

Plánovač cest Tato komponenta slouží pro vytvoření nového plánu cesty a zahrnuje tvorbu itineráře, seznam úkolů, pole pro poznámky a tzv. *inspirativní destinace*. Itinerář umožňuje den po dni naplánovat cestu přidáním konkrétních míst k navštívení. Mezi jednotlivými místy lze zadat způsob dopravy, označit místa k zaparkování, vložit výdaje a přidat

[1] Distribuční služba poskytující digitální obsah, zejména mobilní aplikace, uživatelům operačního systému Android.



Obrázek 1.1: Ukázka aplikace Worldee

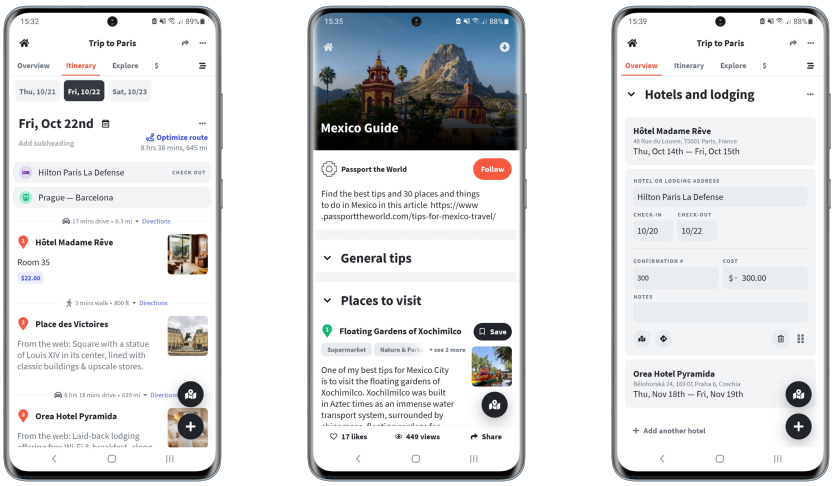
fyzickou náročnost trasy. *Inspirativní destinace* je pak seznam cest od ostatních uživatelů, které aplikace sama navrhuje na základě autorova plánu cesty.

Cesty ostatních uživatelů Proběhlou cestu lze sdílet s ostatními uživateli aplikace, kteří ji mohou využít jako inspiraci pro naplánování svého cestovatelského zážitku. Autor cesty seznamuje zájemce se zážitky doplněnými o fotografie, přesným plánem cesty a mapovým podkladem. Fotografie je možné spustit jako prezentaci, během které je vždy místo pořízení aktuální fotky zvýrazněno na mapě.

Průvodce destinacemi V aplikaci lze vyhledat konkrétní zemi a zobrazit si o ní základní informace. Těmi jsou aktuální čas, vhodná doba k cestování, typ zásuvek, jazyk, bezpečnostní situace, dopravní pravidla a také krátká charakteristika daného státu. Mimo to lze ke každé zemi najít přehled míst k navštívení se stručným popisem.

Aplikace částečně funguje jako sociální síť pro cestovatele, neboť je možné sledovat profily dalších uživatelů, jejichž cesty se zobrazují v osobním kanále. Profil uživatele pak dokumentuje cestovatelskou dráhu dobrodruha ve formě stírací mapy, kde navštívené země jsou barevně zvýrazněny na jinak šedé mapě světa.

Na papíře se jeví Worldee jako skvělá mobilní aplikace, v praxi však nemám z aplikace právě ten nejlepší pocit. Plánovač cest je zatím velmi jednoduchý. Do itineráře nelze přidávat dokumenty. Rovněž není možné přidat aktivity po trase. Vložené výdaje jsou navíc pouze statické a nejde s nimi nijak pracovat. Přidání ubytování a dopravy je také velmi stručné bez možnosti doplnění



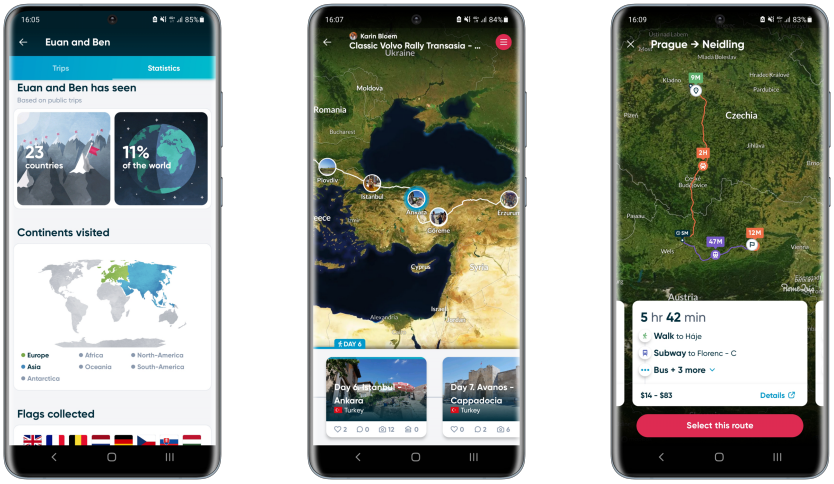
Obrázek 1.2: Ukázka aplikace Wanderlog

dodatečných informací. Na druhou stranu musím vyzdvihnout průvodce destinacemi jakožto opravdu užitečnou službu. Stejně tak mě bavilo číst příspěvky ostatních uživatelů. Bohužel však aplikaci sráží nezdařená optimalizace způsobující sekání, padání a dlouhé načítání příspěvků.

1.2 Wanderlog

Y Combinator, projekt pomáhající úspěšně rozjet startupy, popisuje Wanderlog jako komplexní aplikaci pro plánování příští dovolené se zaměřením na vynikající design. [2] Ve Wanderlogu můžou uživatelé vytvářet itineráře, objevovat nová místa k návštěvě, spravovat lety a hotelové rezervace a sdílet cestovní rady s přáteli a spolucestujícími. Přestože je mobilní aplikace na trhu teprve dva roky, získala si přízeň více než 100 tisíc lidí a obdržela krásné hodnocení 4,7 z 5. [3] Mobilní aplikace je konkrétně tvořena těmito částmi:

Plánovač cest Plánovač cest umožňuje vytvořit itinerář, objevovat okolní pamětihodnosti na základě aktuálního plánu trasy a zapisovat finanční výdaje. Cestu lze sdílet s dalšími spolucestujícími, kteří mohou upravovat itinerář v reálném čase. Ubytování a dopravu je možné přidat do aplikace ručně, anebo přeposláním emailu s potvrzením o ubytování/spoji na unikátní emailovou adresu. Wanderlog sám pak z přeposlaného emailu dodá do itineráře potřebné údaje. Ke každé položce v itineráři lze také přidat cenu. U ubytování a bodů zájmů jsou automaticky staženy základní informace z Google Map – uživatel tak vidí hodnocení, kontakt či stručný popis místa s přiloženými fotkami. Placená verze Wanderlogu navíc umí optimalizovat trasu mezi zadanými body.



Obrázek 1.3: Ukázka aplikace Polarsteps

Průvodce Uživatelé aplikace mohou sledovat cesty ostatních uživatelů. Každý člen Wanderlogu má navíc možnost sepsat cestovatelského průvodce. Průvodce může kombinovat itinerář, tedy přesný časový plán cesty, s body zájmů, které autor doporučuje čtenářům k navštívení.

Aplikace je z mého pohledu velmi intuitivní a graficky přitažlivá. Líbila se mi funkce načtení hotelu nebo cesty přeposláním emailu s potvrzením. Dokonce lze v aplikaci přidat letenku vložením čísla letu a zbylé informace pak aplikace načte sama. Co se mi na druhou stranu nepozdávalo je fakt, že nejdou vkládat dokumenty. Rovněž mě zklamala sekce s výdaji, ve které není možné rozdělit útratu mezi spoluúčastníky.

1.3 Polarsteps

Přes tři miliony lidí používá Polarsteps, aby mohli vést svůj cestovatelský deník s minimem úsilí. [4] Přesně takto zní proklamace aplikace od samotných tvůrců. Polarsteps umožňuje plánování trasy, nicméně její hlavní předností je trasování uživatele. Jakmile dobrodruh cestuje s aplikací v kapse, Polarsteps jeho pohyb zaznamenává a značí na podrobné mapě. Cestu je dokonce možné obohatit o fotografie a popisky. Jan Beneš na Google Play pak aplikaci hodnotí těmito slovy: „Doporučuji všem cestovatelům. Výborná appka k vedení cestovního deníku. Kdekoli jsem, jen kliknu přidat, aplikace zjistí moji lokaci, vyberu nebo vyfotím pár výstižných fotek a uložím.“ [4] V aplikaci také lze sdílet aktuální polohu online, aby tak přátelé věděli, kde se jejich známý nachází. Polarsteps zahrnuje zejména tyto funkce:

Plánování trasy Umožňuje uživatelům vytvořit základní plán trasy pomocí průjezdních bodů. U každého bodu lze nastavit datum příjezdu a počet nocí strávených v daném místě. Mezi jednotlivými body je aplikace schopna navrhnout několik tras podle způsobu dopravy a spočítat cenu.

Trasování cesty Jak již bylo napsáno, tato funkcionalita je klíčovou doménou Polarsteps. Polarsteps automaticky trasuje uživatele a jeho pouť zobrazuje na mapě světa. Poutník může v průběhu cesty přidávat fotky, videa a myšlenky, a obohacovat tak svůj cestovatelský deník. Může také ukládat oblíbená místa po trase. Systém hledání polohy je navíc podle vývojářů šetrný k baterii v telefonu a funguje i bez přístupu k internetu. [4] Celou trasu si může poutník po zaplacení nechat zaslat jako fotoknihu. V aplikaci lze také sledovat i cesty včetně popisků od ostatních světoběžníků.

Cestovatelský průvodce Zobrazuje rady a tipy, kde se dá dobře ubytovat, najíst a jaké památky navštívit. Píšou jej pouze editoři.

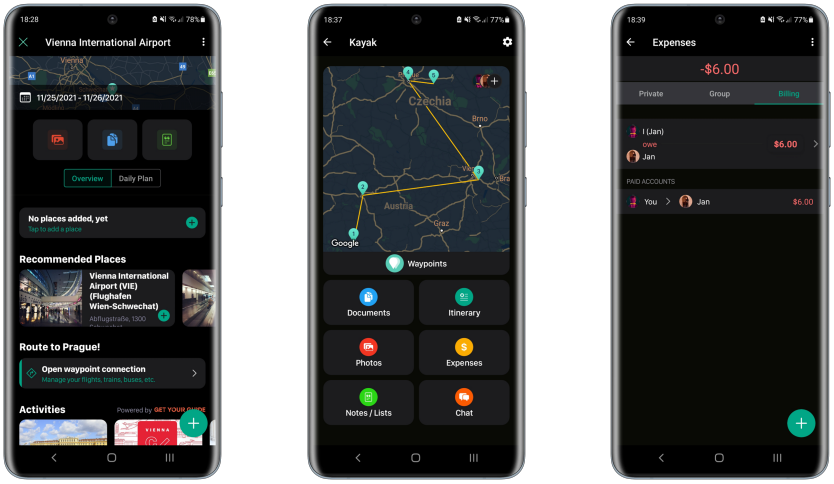
Pokročilé statistiky V profilu uživatele jsou k dispozici relevantní statistiky. Uživatel tak může dohledat seznam navštívených států, informaci o počtu uražených kilometrů, nejdelší vzdálenost od domova a podobně. Statistické hodnoty se aktualizují pouze během pohybu uživatele.

Aplikace není vhodná k plánování trasy, neboť obsahuje velmi omezené množství funkcí. V čem ale Polarsteps opravdu vyniká, je tvorba cestovatelského deníku, který je vizuálně nádherně zpracovaný, a přitom jednoduchý na procházení.

1.4 Lambus

Před pár lety měl Němec Hans Knöechel potíže s nalezením takové cestovatelské aplikace, jež by splňovala jeho náročné podmínky. Přál si, aby pro veškeré informace související s jeho cestami mohl použít pouze jednu aplikaci. Jelikož ale taková podle něj v té době neexistovala, rozhodl se naprogramovat si vlastní. [5] Tímto způsobem spatřil světlo světa Lambus, aplikace určená ke společnému plánování cesty, která nabádá uživatele k několika krokům – od počáteční inspirace cest ostatních cestovatelů, přes samotné plánování itineráře, koupení jízdenek, sledování informací o letu a počasí až po evidenci výdajů. [6] Přesněji Lambus obsahuje tyto komponenty:

Plánování cesty Rozhraní pro plánování cesty zahrnuje několik možností. Uživatel může přidávat cestovní dokumenty, fotografie a poznámky k cestě. Rovněž může chatovat s ostatními společníky. V neposlední řadě mohou cestovatelé evidovat výdaje během cesty, které aplikace poté rozpočte mezi ostatní. Každý tak ví, kolik má komu zaplatit. Nesmím



Obrázek 1.4: Ukázka aplikace Lambus

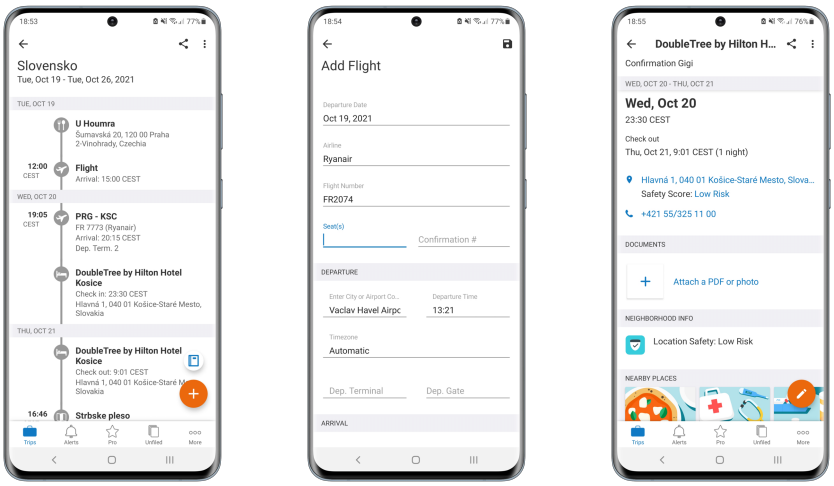
opomenout ani hlavní funkci plánovače, a to tvorbu itineráře. V něm lze přidávat průjezdní body, evidovat způsob dopravy mezi nimi nebo načíst údaje o letu přímo z čísla letu. Lambus také umožňuje nalézt spojení mezi dvěma body a rovnou si přes aplikaci koupit palubní lístek.

Inspirace pro cesty Autoři aplikace nabízí uživatelům inspiraci k cestám. U každé nabízené destinace či cestovatelského plánu je možné si přečíst krátké povídání nebo prohlédnout fotky či samotné body zájmů. Ty jdou navíc zobrazit na mapě.

Lambus je dle mého uživatelsky přívětivá aplikace s pěkným grafickým rozhraním. Oceňuji, že evidence výdajů dovoluje rozpočítat útratu mezi cestovateli. Také mě příjemně překvapila obrazovka s bodem zájmu. Po rozkliknutí bodu totiž aplikace doporučuje cestovatelům zajímavá místa v okolí a navrhuje aktivity, kterých je možné se poblíž zúčastnit. Lambus navíc zobrazuje celkové hodnocení daného místa a kontakt. Přestože ale aplikace u některých míst čítá stovky recenzí, zobrazuje Lambus pouze zlomek slovních ohodnocení, které jsou navíc neaktuální. Za největší nevýhodu Lambusu ovšem vidím absenci přidání ubytování, které by mnoho uživatelů jistě ocenilo. U zbylých bodů zájmů pak lze bohužel zadávat výlučně základní informace.

1.5 TripIt

TripIt je užitečný pomocník, s jehož pomocí mohou uživatelé sestavit kompletní itinerář své cesty. Aplikace zajišťuje přehled o nadcházejících plánech.



Obrázek 1.5: Ukázka aplikace TripIt

Ručí také, že veškeré potřebné dokumenty od letenek po hotelové rezervace budou pro uživatele na jednom místě. [7] Aplikace je rovněž schopna informovat o pandemické situaci nebo míře bezpečnostního rizika v naplánované lokalitě. Placená verze umí navíc sledovat údaje o letu v reálném čase a notifikovat uživatele o zpoždění nebo zlevnění letenky. Také zvládá zobrazit podrobnou mapu letiště či říct, jak dlouho potrvá čekání ve frontě na letišti. Funkcionalita aplikace je nicméně soustředěna pouze do jedné části:

Cesty Uživatel v aplikaci vidí proběhlé a naplánované cesty. U každé cesty může cestovatel zaznamenat údaje o ubytování, restauracích, parkování, dopravě, letenkách, aktivitách a dalších. Ke všem bodům zájmů lze také přidat fotografie či PDF[2]dokumenty. Pokud turista neví, jak se na zadaný bod dostat, má možnost si přímo v aplikaci vyhledat způsob dopravy spolu s cenou. Jestliže navíc potřebuje v okolí najít bankomat, lékárnu, kavárnu či jiný podnik, může si přes aplikaci zobrazit různě vzdálené eventuality i s hodnocením.

Aplikace dovoluje mít všechny potřebné dokumenty a přehled o cestě na jednom místě i bez přístupu k internetu. Líbila se mi originální funkce *Neighborhood Safety Score*, která počítá míru bezpečnostního rizika v dané lokalitě (a různých kategoriích). Taky považuji za propracované hledání způsobu dopravy, kdy TripIt nabízí uživatelům několik alternativ s ohledem na cenu a čas. Na závěr bych podotkl, že aplikace poskytuje dostatek druhů aktivit, které lze do plánu přidat. Opatření kvůli onemocnění *Covid-19* jsou rovněž přehledně

[2]Z anglického Portable Document Format, jedná se o formát souborů, který je přenositelný mezi různými druhy softwaru i hardwaru.

zakomponovány. Zklamala mě ale nepřítomnost uživatelských recenzí u hotelů a restaurací. Také mě mrzí „osekanost“ základní verze.

1.6 Shrnutí

V této kapitole jsem popsal pět vybraných mobilních aplikací pro Android. U každé aplikace jsem vysvětlil její základní funkce a vyjádřil mé postřehy z používání. Zjistil jsem, že naplánovat itinerář cesty a sdílet jej s dalšími spolucestovateli umí všechny mnou zkoumané aplikace. Počet dalších funkcí se ovšem mezi aplikacemi značně liší. Některé aplikace vynikají ve tvorbě cestovatelského deníku. Jiné naopak překonávají ostatní v množství aktivit a údajů, které lze v itineráři zaznamenat. Mnohé aplikace informují uživatele o zajímavostech, opatřeních či restrikcích po trase. Pozorované aplikace nejlépe charakterizuje následující seznam:

- Worldee – cestovatelský deník s podrobnými údaji o jednotlivých státech,
- Wanderlog – plánovač cest s mnohými doplňujícími informacemi o bodech zájmů,
- Polarsteps – cestovatelský deník, který vzniká sám pohybem uživatele,
- Lambus – plánovač cest, jenž rozpočte výdaje mezi spoluúčastníky,
- TripIt – plánovač cest s cennými informacemi o bezpečnosti a opatřeních.

Výhodou studovaných aplikací je fakt, že kombinují funkce mnoha jiných aplikací, které by jinak uživatel musel mít nainstalované v telefonu. Podrobný přehled jejich funkcí uvádím na závěr v tabulkách 1.1 a 1.2.

Během psaní této kapitoly jsem si ověřil skutečnost, že se na trhu vyskytuje nepřeberné množství aplikací usnadňující uživatelům cestování. Většina z nich má mnoho pokročilých funkcí, nicméně jednoduchou a přehlednou tvorbu itineráře, který lze navíc sdílet s dalšími členy, obsahovala pouze jedna ze studovaných aplikací – TripIt.

Tabulka 1.1: Přehled funkcionalit aplikací během tvorby itineráře

Tvorba itineráře	Worldee	Wanderlog	Polarsteps	Lambus	TripIt
body zájmu	✓	✓	✓	✓	✓
ubytování	✓	✓			✓
způsob dopravy	✓	✓	✓	✓	✓
zobrazení na mapě	✓	✓	✓	✓	✓
přidání dokumentů/fotek				✓	✓
evidence platebních výdajů	✓	✓		✓	
vyúčtování				✓	
poznámky	✓	✓		✓	✓
chat				✓	
načtení letu z kódu		✓		✓	

Tabulka 1.2: Přehled ostatních funkcionalit aplikací

Funkcionalita	Worldee	Wanderlog	Polarsteps	Lambus	TripIt
doporučení okolních bodů	✓	✓		✓	✓
cestovatelský průvodce	✓	✓	✓	✓	
statistiky	✓		✓		
opatření proti *Covid-19*				✓	✓
vyhledání (optimální) cesty		✓	✓	✓	✓
počasí na trase				✓[3]	✓[4]
doplňující informace[5]		✓		✓	
načtení dokumentů z e-mailu		✓		✓	✓

[3] Pouze v placené verzi.
[4] Dostupné pouze ve webové verzi.
[5] Doplňující informace o bodech zájmu, například informace z *Wikipedie*, recenze apod.

Kapitola 2

Analýza požadavků

V úvodu této kapitoly popisuji klíčové pojmy, jež se vyskytují později v této práci. Rovněž zde rozebírám funkční požadavky, které jsem sestavil na základě analýzy stávajících mobilních aplikací. Na závěr pak seznamuji čtenáře s cílovými uživateli budoucí aplikace a doplňuji text o dva diagramy případů užití.

2.1 Klíčové pojmy

- **Cesta** – cesta je soubor aktivit, ubytování, dopravy a míst, jež má uživatel v plánu navštívit. Cestu můžeme chápat i jako výlet nebo expedici a je zpravidla časově omezená. Cestu lze sdílet s dalšími uživateli, kteří se poté mohou podílet na spoluorganizaci. Skupinu uživatelů jedné cesty je taky možné pojmenovat jako výpravu.

- **Bod zájmu** – každá cesta se skládá z bodů zájmů, které si uživatel či uživatelé naplánovali. Bodem zájmu se myslí aktivita, ubytování, doprava nebo místo na trase.

2.2 Funkční požadavky

Cílem funkčních požadavků je vymezení hranic systému, umožnění přesnějšího odhadu pracnosti a vyjasnění si zadání se zákazníkem. Mimo to se snaží zachytit omezení, která jsou na informační systém kladena. [8] Důležitým předpokladem je fakt, že musí být jednoznačné a snadno ověřitelné. Pro zpřesnění dělíme požadavky na funkční a nefunkční.

V úvodu bych rád podotkl, že v pozdější fázi této kapitoly uvádím sousloví „uživatel s patřičným oprávněním“. Tímto termínem mám na mysli uživatele, který je buď administrátor nebo editor.

Pro větší přehlednost rozděluji požadavky do samostatných logických celků. Zároveň doplňuji každý požadavek o jednu ze tří priorit:

- **nutný** – požadavek, bez kterého by se základní verze aplikace neobešla; má nejvyšší prioritu,
- **chtěný** – požadavek, který rozšiřuje základní verzi aplikace; má střední prioritu,
- **volitelný** – požadavek, který je nad rámec základní i rozšířené verze aplikace; má nejnižší prioritu.

2.2.1 Seznam funkčních požadavků

Funkční požadavky vyjadřují, jak by se měl systém v určitých situacích chovat. Zjednodušeně řečeno tím definujeme, co by měl systém potažmo aplikace umět. [9]

a) **Aplikace jako celek**

i. **F1 – Komunikace mezi mobilní aplikací a webovou službou** – nutný
Webová služba bude odpovídat na požadavky mobilní aplikace.

ii. **F2 – Přístup ke vzdálené databázi** – nutný
Aplikace umožní přístup ke sdíleným datům z vícero zařízení a zároveň umožní sdílet stejná data mezi více uživateli.

iii. **F3 – Zobrazení notifikací** – volitelný
Aplikace bude zobrazovat notifikace, například po přidání bodu zájmu jiným uživatelem, pozvání do sdílené cesty a podobně.

b) **Přihlášení a registrace**

i. **F4 – Registrace nových uživatelů** – nutný
Registrace bude povinná pro všechny nové uživatele. Základním identifikátorem uživatele bude jeho uživatelské jméno.

ii. **F5 – Přihlášení se pomocí emailu a hesla** – nutný
Uživatelé se budou přihlašovat pomocí zadané kombinace emailu a hesla.

iii. **F6 – Přihlášení se pomocí účtu Google** – volitelný
Uživatelé budou mít možnost se přihlásit účtem Google.

iv. **F7 – Resetování hesla po jeho zapomenutí** – chtěný
Pokud uživatel v případě přihlášení pomocí emailu a hesla zapomene heslo, bude si jej moci resetovat. Aplikace odešle email s odkazem na změnu hesla.

c) **Účet**

i. **F8 – Odhlášení se z účtu** – nutný
Uživatel se bude moci odhlásit ze stávajícího zařízení. Odhlášení bude fungovat i bez připojení k internetu.

ii. **F9 – Odhlášení se ze všech účtů** – chtěný
Uživatel se bude moci odhlásit ze všech zařízeních, na kterých je aktuálně přihlášen.

iii. **F10 – Smazání účtu** – chtěný
Uživatel si bude moci smazat účet. Aplikace v takovém případě jeho účet zneplatní, nicméně uživatelem vložená data v aplikaci nadále zůstanou (například body zájmů v cestě, jíž se účastní i další uživatelé).

iv. **F11 – Změna hesla** – nutný
Uživatel si v případě přihlášení se pomocí emailu a hesla bude moci změnit heslo. Aplikace bude vyžadovat zadání aktuálního hesla.

v. **F12 – Změna zobrazovaného jména** – chtěný
Každý uživatel je identifikován uživatelským jménem. Pokud uživatel nechce, aby jeho uživatelské jméno bylo veřejné, může si nastavit zobrazované jméno, kterým se prezentuje.

d) **Cesta**

i. **F13 – Zobrazení nadcházejících cest** – nutný
Uživatel v aplikaci uvidí seznam cest, které aktuálně probíhají nebo budou probíhat.

ii. **F14 – Zobrazení proběhlých cest** – nutný
Uživatel v aplikaci uvidí seznam již proběhlých cest.

iii. **F15 – Uživatelé v konkrétní cestě mohou mít různé role** – chtěný
V každé cestě bude uživatel mít jednu ze tří rolí – administrátor, editor, divák. Každá role bude mít jiná oprávnění.

iv. **F16 – Přidání nové cesty** – nutný
Uživatel bude moci vytvořit novou cestu. U nové cesty půjde zadat název, počáteční a koncové datum a popis. Přidáním cesty se uživatel automaticky stává administrátorem dané cesty.

v. **F17 – Smazání cesty** – nutný
Uživatel bude moci smazat cestu. Smazat cestu bude smět pouze administrátor za předpokladu, že bude ve výpravě sám anebo se spolu s ním bude nacházet ve výpravě další administrátor. V opačném případě mu bude zobrazeno upozornění.

vi. **F18 – Editování cesty** – nutný
Uživatel bude moci změnit údaje dané cesty. Editovat cestu budou smět pouze administrátoři a editoři.

vii. **F19 – Přidání bodů zájmů ke konkrétní cestě** – nutný
Body zájmů budou moci přidávat pouze administrátoři a editoři.

viii. **F20 – Evidence výdajů** – chtěný
Uživatelé s patřičným oprávněním budou moci u každé cesty evidovat a rozpočítávat výdaje.

ix. **F21 – Zobrazení itineráře** – nutný
Body zájmů konkrétní cesty budou zobrazeny v chronologickém pořadí. Po kliknutí na bod zájmu budou uživateli představeny podrobnější informace o daném bodě.

x. **F22 – Přehled jednotlivých bodů zájmů** – chtěný
Každá cesta bude obsahovat seznam bodů zájmů dle kategorií. Po kliknutí na bod zájmu bude uživateli otevřena stránka s podrobnějšími informacemi o daném bodu.

xi. **F23 – Opuštění výpravy** – nutný
Uživatel může opustit výpravu, neboli opustit ostatní členy dané cesty. V takovém případě pak cesta zmizí i ze seznamu uživatelových cest. Opustit výpravu nepůjde, pokud bude uživatel administrátorem a zároveň bude výprava obsahovat další členy z nichž nikdo nebude administrátorem.

xii. **F24 – Pozvání nového člena** – nutný
Administrátoři a editoři budou moci přizvat do výpravy nové členy a přiřadit jim role. Editor bude smět nastavit novému účastníkovi roli diváka nebo editora. Administrátor bude moci nad to pozvat i nového administrátora.

xiii. **F25 – Změna role** – chtěný
Administrátor bude moci změnit účastníkovi cesty roli. Pokud administrátor nebude se členem výpravy spokojen, bude smět jej vyloučit.

xiv. **F26 – Zobrazení mapy s body zájmů** – chtěný
Uživatel si bude moci rozkliknout mapu se značkami znázorňující přidané body zájmů. Značky budou barevně rozlišeny podle jednotlivých kategorií bodů zájmů.

xv. **F27 – Přidání cesty do kalendáře** – chtěný
Uživatel si bude moci přidat cestu do externího kalendáře (např. Google Calendar, Outlook atd.).

xvi. **F28 – Přidání dokumentu** – chtěný
Uživatelé s patřičným oprávněním budou moci přidávat ke konkrétní cestě dokumenty.

e) **Bod zájmu**

i. **F29 – Přidání bodu zájmu** – nutný
Uživatelé s patřičným oprávněním budou moci přidat bod zájmu.

Bod zájmu se bude skládat z rozličných parametrů dle kategorie. S výběrem místa pomůže našeptávač od Google. Pro usnadnění budou také některé parametry předvyplněny podle výsledku z našeptávače.

ii. **F30 – Smazání bodu zájmu** – nutný
Uživatelé s patřičným oprávněním budou moci smazat bod zájmu. Pokud budou součástí bodu i dokumenty, pak budou smazány taky.

iii. **F31 – Editování bodu zájmu** – nutný
Uživatelé s patřičným oprávněním budou moci upravit údaje o bodě.

iv. **F32 – Přidání dokumentu** – chtěný
Uživatelé s patřičným oprávněním budou moci přidávat k danému bodu zájmu dokumenty.

v. **F33 – Zobrazení bodu zájmu** – nutný
Informace o bodu zájmu půjdou zobrazit na samostatné obrazovce.

vi. **F34 – Zobrazení mapy** – chtěný
Pokud bude bod zájmu obsahovat údaje o jeho umístění, pak v přehledu daného bodu bude zobrazena mapa s označením místa.

vii. **F35 – Přidání bodu zájmu do kalendáře** – chtěný
Uživatel si bude moci přidat bod zájmu do externího kalendáře (např. Google Calendar, Outlook atd.).

viii. **F36 – Zobrazení Wikipedie** – volitelný
Aplikace zobrazí článek z Wikipedie u přidaného místa.

ix. **F37 – Zobrazení počasí** – volitelný
Aplikace zobrazí u bodu zájmu krátkodobou předpověď počasí.

x. **F38 – Zobrazení webkamery** – volitelný
Aplikace zobrazí u bodu zájmu nejbližší webkameru.

xi. **F39 – Načtení letu z kódu** – volitelný
Během přidávání letu budou údaje o letu načteny z kódu.

f) **Dokument**

i. **F40 – Přidání dokumentu** – chtěný
Uživatel s patřičným oprávněním bude moci přidat dokument. U dokumentu bude smět nastavit jeho jméno a klíč nutný pro zobrazení.

ii. **F41 – Zobrazení dokumentu** – chtěný
Libovolný uživatel si bude moci zobrazit dokument. Pokud bude dokument opatřen klíčem, bude muset pro jeho zobrazení zadat platný klíč. Dokument bude zobrazován pod názvem, jenž zvolil jeho vlastník během nahrávání.

iii. **F42 – Smazání dokumentu** – chtěný
Dokument bude moci smazat jeho vlastník nebo administrátor.

g) **Evidence výdajů**

i. **F43 – Přidání místnosti pro evidenci výdajů** – chtěný
U každé cesty půjdou evidovat výdaje. Pro větší přehlednost bude evidování výdajů probíhat v místnosti k tomu určené. To znamená, že jedna cesta může obsahovat několik rozličných místností, jejichž členové budou moci sdílet náklady mezi sebou. Členové musí být unikátní a zároveň každá místnost bude vedena v konkrétní měně.

ii. **F44 – Smazání místnosti pro evidenci výdajů** – chtěný
Uživatel s patřičným oprávněním bude moci smazat místnost pro evidenci výdajů. Zároveň budou smazány veškeré výdaje spojené s místností.

iii. **F45 – Editování místnosti pro evidenci výdajů** – chtěný
Uživatelé s patřičným oprávněním budou moci upravit údaje o místnosti.

iv. **F46 – Přidání výdaje** – chtěný
Uživatel s patřičným oprávněním bude moci přidat výdaj. U výdaje bude mimo název a datum evidována také částka, plátce a seznam dlužníků.

v. **F47 – Smazání výdaje** – chtěný
Uživatel s patřičným oprávněním bude moci smazat výdaj.

vi. **F48 – Editování výdaje** – chtěný
Uživatel s patřičným oprávněním bude moci upravit výdaj.

vii. **F49 – Zobrazení výdajů** – chtěný
Aplikace zobrazí seznam všech výdajů v dané místnosti.

viii. **F50 – Navrhnutí plateb** – chtěný
Aplikace navrhne platby mezi účastníky dané místnosti tak, aby počet transakcí byl co nejmenší a nikdo nikomu nic nedlužil.

h) **Ostatní**

i. **F51 – Mobilní aplikace bude podporovat více jazyků** – volitelný
Mobilní aplikace bude dostupná v angličtině a češtině. Uživatel si bude moci změnit jazyk přímo v aplikaci.

ii. **F52 – Mobilní aplikace bude ve světlém a tmavém režimu** – volitelný
Mobilní aplikace bude dostupná ve světlém a tmavém režimu. Uživatel si bude moci změnit režim přímo v aplikaci.

2.2.2 Seznam nefunkčních požadavků

Nefunkční požadavky na rozdíl od funkčních definují vlastnosti a omezení služeb, jež jsou systémem nabízeny. [9]

a) **Mobilní aplikace**

i. **N1 – Operační systém** – nutný
Mobilní aplikace bude dostupná pro operační systém Android. Minimální verze API[6] bude 26 (viz podkapitola 4.2).

ii. **N2 – Internetové připojení** – nutný
Pro zajištění aktuálních dat bude mobilní aplikace vyžadovat připojení k internetu.

iii. **N3 – Dostupnost offline** – volitelný
Body zájmu, cesty a dokumenty budou uloženy v keš paměti zařízení.

iv. **N4 – Přístup k poloze** – chtěný
Pro zobrazení aktuální polohy bude mobilní aplikace vyžadovat přístup k poloze zařízení.

b) **Aplikace jako celek**

i. **N5 – Rozšířitelnost** – chtěný
Aplikace bude rozšířitelná o další způsoby autentizace.

ii. **N6 – Srozumitelnost pro uživatele** – chtěný
V případě chyby zobrazí aplikace adekvátní chybovou hlášku.

iii. **N7 – Bezpečnost** – nutný
Neautorizovaný přístup bude odepřen. Požadavek bez patřičného oprávnění bude odmítnut.

iv. **N8 – Sdílení dat mezi uživateli** – nutný
Aplikace bude podporovat sdílení společných dat mezi více uživateli.

2.3 Cíloví uživatelé

Cílové uživatele lze rozdělit do dvou skupin:

- nepřihlášený uživatel,
- přihlášený uživatel.

Nepřihlášený uživatel nemá v aplikaci příliš mnoho možností. Může se zaregistrovat nebo přihlásit. V případě, že je již zaregistrován, ale nedopatřením zapomněl heslo, si jej může resetovat.

[6] Z anglického Application Programming Interface, jedná se o sadu funkcí, které umožňují dvěma aplikacím spolu komunikovat.

Přihlášený uživatel může používat aplikaci bez omezení. V kontextu cest ale dále specifikujeme uživatele na:

- diváka,
- editora,
- admina.

2.3.1 Divák

Jak již z názvu vyplývá, divák se cesty účastní pouze jako pozorující člen. Nemá dostatečná oprávnění pro to, aby aktivně zasahoval do organizace výletu. Přesto si může prohlížet jednotlivé body zájmů, uložit si cestu či bod do kalendáře a zobrazit si itinerář. V neposlední řadě smí stahovat dokumenty a procházet vložené výdaje.

2.3.2 Editor

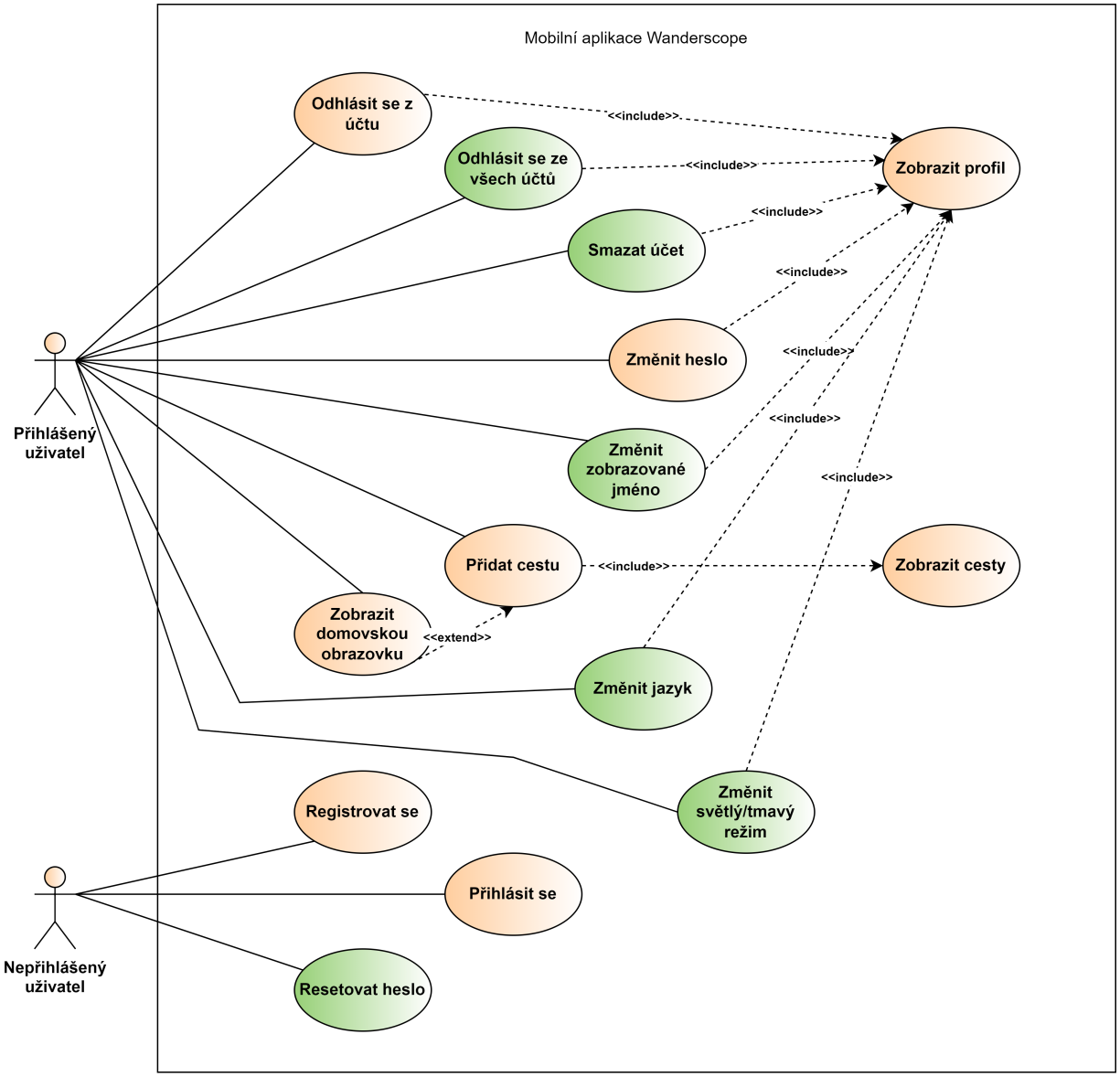
Editor se aktivně podílí na plánování cesty. Mimo pravomocí diváka smí přidávat, upravovat a mazat body zájmů. Také je mu dovoleno zakládat nové místnosti pro evidenci výdajů a výdaje zaznamenávat. Rovněž může přizvat do výpravy další editory nebo diváky.

2.3.3 Admin

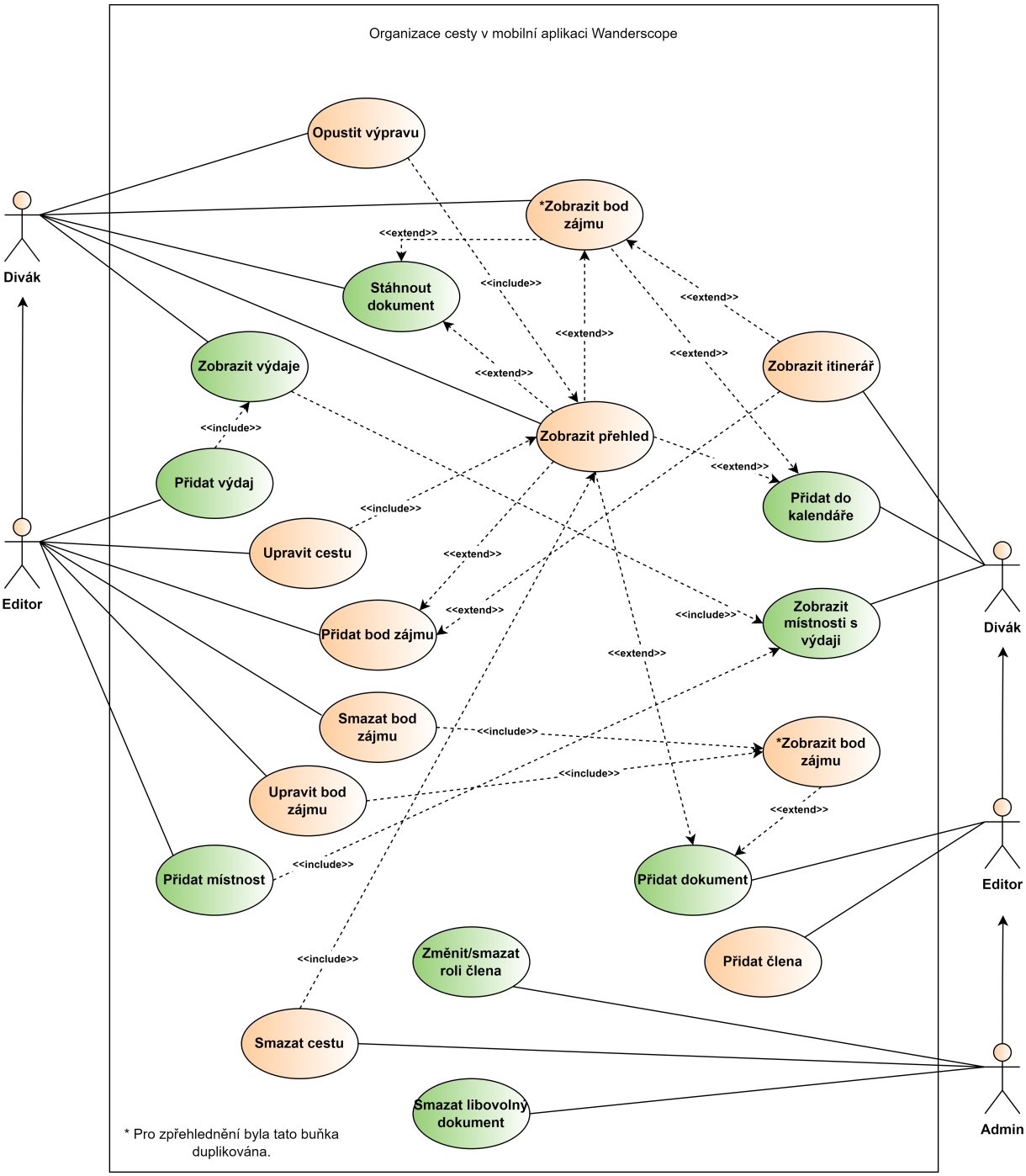
Admin je nejkompetentnější osobou výpravy. Uživatel se jim stává automaticky po vytvoření cesty. Kromě pravomocí editora je schopen přidat do výletu nové administrátory. Navíc má oprávnění ke změně role již stávajícího člena. Aby toho nebylo málo, admin smí smazat cizí dokument a v poslední řadě smazat celý výlet.

2.4 Use case diagram

Pro lepší představu přikládám dva use case diagramy (diagramy případů užití), jež kombinují funkční požadavky a příslušné role do dvou kompaktních obrázků. Diagram 2.1 znázorňuje oprávnění dvou základních aktorů – nepřihlášeného a přihlášeného uživatele. Diagram 2.2 pak upřesňuje jednotlivé role v rámci cesty. Oba dva diagramy 2.1 a 2.2 obsahují kvůli přehlednosti oranžové a zelené bubliny. Bubliny se světle oranžovým pozadím označují nutné požadavky. Bubliny se zeleným pozadím pak označují chtěné, resp. volitelné požadavky.



Obrázek 2.1: Diagram případů užití pro přihlášené a nepřihlášené uživatele



Obrázek 2.2: Diagram případů užití během organizace cesty

Kapitola 3

Backend

V úvodu této kapitoly popisuji výběr některých technologií a programovacího jazyka. Dále zde rozebírám nástroje použité pro vývoj mé webové služby a zvolenou architekturu. V neposlední řadě doplňuji tento text o databázový model a návrh autentizace. V pozdější části této kapitoly pak představuji čtenáři vybrané body z implementace backendu a nasazení webové služby na cloud.

3.1 Výběr technologií a jazyk

V této podkapitole se dočtete o výběru jazyka a některých technologiích, které jsem se rozhodl použít pro implementaci vlastní webové služby.

3.1.1 Programovací jazyk

Kotlin je již několik let standardem pro programování nativních aplikací v Androidu. Není se čemu divit, Kotlin je z vlastní zkušenosti velmi expresivní jazyk, který je zároveň stručný a díky své syntaxi snižuje výskyt běžných chyb v kódu (např. *NullPointerException*). Nyní tomu budou už tři roky, kdy byl Kotlin oficiálně představen veřejnosti jako výchozí jazyk pro vývoj v Androidu. [10] A jeho představení se dočkalo úspěchu. Převedeno do řeči čísel, 80 % z 1000 nejstahovanějších aplikací pro Android obsahuje v této době alespoň část kódu v Kotlinu. [11] Osobně jsem s Kotlinem velmi spokojen, mám v plánu jej použít pro programování mobilní aplikace (viz podkapitola 4.1), a proto jsem si tento jazyk zvolil jako primární i pro implementaci backendu.

3.1.2 Framework

Skvělou vlastností Kotlinu je fakt, že je plně kompatibilní s Javou. Tedy na místě, kde se dá použít Java, je možné použít i Kotlin. Během výběru jsem nicméně cílil na ten framework, který nabízí Kotlinu plnou podporu a nejlépe

je vytvořen přímo v něm. Nakonec jsem si zvolil Ktor Server, nepříliš známý framework vyvíjený firmou JetBrains[7]. Na následujících řádcích vysvětluji proč.

Ktor je od základu postaven na korutinách (anglicky coroutines), konceptu podobnému vláknům umožňujícímu asynchronní programování. Korutiny mají ale oproti klasickým vláknům velmi malou režii a jejich použití je intuitivní. Ktor aplikace tak obsluhuje všechny požadavky asynchronně, což má za následek vyšší rychlost zpracování současně běžících požadavků.

Další výhodou frameworku je jeho velká flexibilita a jednoduchost. „Ktor dovoluje vývojáři použít pouze to, co vývojář opravdu použít chce," hlásí tvůrci Ktoru na webu. [12] Každá Ktor aplikace se totiž zpravidla skládá z několika pluginů (původně features). Ty lze chápat jako sadu funkcí, jež pomáhají programátorům s vývojem. Záleží pak pouze na vývojáři, které z nabízených pluginů si do své aplikace nainstaluje. Pro lepší představu – pokud chce vývojář například zabezpečit svojí aplikaci přidáním emailu a hesla ke každému požadavku, nainstaluje si *Authentication plugin* (viz ukázka 3.1). Tento plugin pak obsahuje metody, pomocí kterých může vývojář zkontrolovat, zdali je kombinace emailu a hesla správná. Zároveň je jenom na vývojáři, jakým architektonickým stylem bude svojí aplikaci strukturovat.

```
install(Authentication) {
    basic("auth-basic") {
        validate { credentials ->
            if (credentials.name == "jetbrains" &&
                credentials.password == "foobar"
            ) {
                UserIdPrincipal(credentials.name)
            } else {
                null
            }
        }
    }
}
```

Ukázka kódu 3.1: Příklad použití frameworku Ktor

Z ukázky zdrojového kódu 3.1 lze vidět, že je zápis v Ktoru snadno čitelný. Děje se tak zejména díky promyšlenému návrhu API, kdy autoři Ktoru naplno využili potenciál Kotlinu DSL. Kotlin DSL vychází z anglického Domain Specific Language a jedná se o chytře použité lambda a extension funkce[8], jež činí kód srozumitelný i pro Nováčky.

[7] Firma JetBrains je autorem a hlavním správcem programovacího jazyku Kotlin.

[8] Kotlin umožňuje díky speciální syntaxi rozšířit třídy o nové funkce bez nutnosti dědění nebo použití návrhového vzoru.

3.1.3 Databáze a ORM

Pro správné fungování aplikace je nutné zajistit perzistentní uložení dat na serveru. Abych se vyhnul přímé práci s databází, rozhodl jsem se použít některý z dostupných ORM frameworků. Tři písmena ve zkratce ORM vyjadřují objektově relační mapování, které je užitečné, pokud chceme propojit objektově orientovaný programovací (OOP) jazyk s relační databází. V OOP pracujeme s objekty, jež pro lepší představivost a určitou praktičnost zastupují věci ze světa kolem nás. Relační databáze naopak zachází s koncepcí řádků a tabulek. Jeden řádek reprezentuje uložený objekt. ORM framework se pak stará o to, abychom mohli jednoduše převádět objekty na řádky a obráceně.

Obrázek 3.1: Maskotem ORM frameworku Exposed je oliheň [13]

3.1.3.1 ORM framework

Pro svůj projekt jsem si vybral ORM framework s maskotem olihně – Exposed. Maskot olihně ostatně nevybrali tvůrci náhodou. „Exposed může stejně jako náš maskot napodobovat různé databázové systémy, a pomáhat tak vytvářet aplikace bez závislostí na konkrétním systému," popisují framework zaměstnanci JetBrains, kteří stojí za jeho vznikem. [14] Framework je napsaný čistě v Kotlinu a nabízí dvě úrovně přístupu k databázi. První možností je psát SQL[9]dotazy pomocí syntaxe Kotlinu díky již zmíněnému DSL. Eventualitou je pak použití objektů, jež zapouzdřují přístup k datům, tzv. DAO neboli anglicky Data Access Object(s). Více uchopitelná pro mě byla první možnost, kterou jsem si do mé práce nakonec vybral.

3.1.3.2 Databáze

Exposed podporuje mnoho populárních databázových systémů. Jak jsem uvedl v odstavci výše, Exposed eliminuje závislost vyvíjené aplikace na databázovém systému. Při výběru databáze jsem tak měl volnou ruku a má ruka se obrazně řečeno rozhodla pro PostgreSQL[10], dle StackOverflow aktuálně druhou nejoblíbenější relační databázi na trhu. [15] Mozek ovládající mou ruku přitom

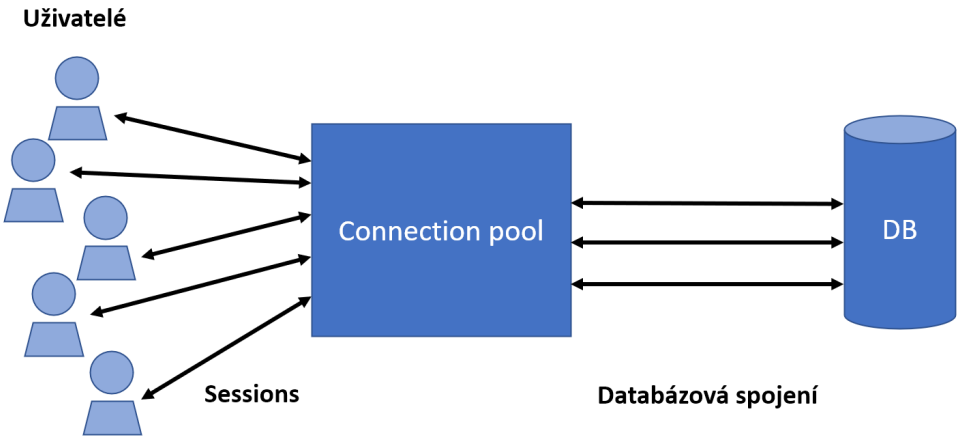
[9]Z anglického Structured Query Language, jedná se o dotazovací jazyk v relačních databázích.

myslel ještě na jednu věc. Díky vysoké popularitě Postgresu by mělo být propojení aplikace s databázovým serverem snazší, což se později během nasazení ukázalo býti pravdou.

3.1.4 Connection pooling

Uživatelé jakýchkoliv aplikací mají rádi, když aplikace běží rychle, a nemusí tak dlouho čekat na vyřízení požadavku. Vytváření spojení pro každý databázový požadavek je ale velice neefektivní a poměrně dlouhý proces. Kdybychom ale dopředu vytvořili několik databázových spojení, které budou v případě potřeby znovu použity, docílili bychom značné úspory času a vyšší spokojenosti uživatelů.

Přesně o to se naštěstí stará koncept zvaný *connection pooling*. Vlastní implementace *connection poolingu* by byla nesmírně složitá, a tak jsem si pro mojí diplomovou práci vybral knihovnu HikariCP. Ta je velmi jednoduchá, spolehlivá a zároveň si vede v hodnocení rychlosti skvěle. [16]



Obrázek 3.2: Ukázka fungování konceptu *connection pool*

3.1.5 Úložiště dokumentů

Mým původním záměrem bylo ukládat obsah souborů do databáze jakožto pole bytů (datový typ *Blob* v databázi). V průběhu implementace mi ale došlo, že velikost mé databáze by rychle stoupala a pokud chci nasadit back-end na bezplatný cloud, pak jistě bude mít nějaké omezení co se do velikosti týče. Pozměnil jsem tedy můj návrh a pro ukládání souborů zvolil uložiště od Amazonu s názvem Simple Storage Service, neboli S3.

Amazon S3 je služba pro ukládání dat, která deklaruje vysokou škálovatelnost, špičkovou dostupnost dat, zabezpečení a výkon. [17] Pro mé potřeby

[10] Též někdy označován jako Postgres.

je služba navíc zdarma. Služba rovněž dokáže zašifrovat uložené dokumenty. K obsahu dokumentů by se tak neměl dostat nikdo cizí, a to ani prostředník po síti, neboť komunikace mezi mobilní aplikací a backendem bude probíhat pomocí zabezpečeného protokolu HTTPS[11].

3.1.6 HTTP klient

V předchozí podkapitole 3.1.2 jsem se rozepsal o Ktoru – frameworku, který jsem se rozhodl použít jako základ pro implementaci backendu. Popravdě bych měl ale framework označovat celým názvem Ktor Server, neboť existuje i Ktor Client od stejných tvůrců.

Jak si lze domyslit, Ktor Client je asynchronní HTTP[12]klient, který umožňuje zadávat požadavky a zpracovávat odpovědi. Mimo to lze jeho funkčnost rozšířit pomocí pluginů jako je autentizace, serializace, logování a podobně. Do jisté míry je tedy podobný svému většímu bratrovi a opět stojí na základech Kotlinu DSL. Díky intuitivnímu API a srozumitelné dokumentaci jsem jej proto zařadil do mého technologického stacku.

```
client.post("https://apakrychle.ninja") { //POST pozadavek
   url {
       parameters.append("admin", "true") //dotaz
   }
   headers {
       append(HttpHeaders.Authorization, "heslo123") //hlavicka
       append(HttpHeaders.UserAgent, "apakrychle") //hlavicka
   }
   contentType(ContentType.Application.Json) //typ
   setBody(User(2, "Tomas", "Zacatecnik")) //telo
}
```

Ukázka kódu 3.2: Volání HTTP požadavku v Ktor Client

3.2 Nástroje pro vývoj

Mimo popsané technologie v kapitole 3.1 budou během vývoje použity tyto nástroje:

- Git – systém pro zálohování a verzování aplikace,
- IntelliJ IDEA – vývojové prostředí pro snazší psaní zdrojového kódu v Kotlinu,

[11] Z anglického Hypertext Transfer Protocol Secure, jedná se o protokol ke komunikaci na internetu, který šifruje přenos dat a je z toho důvodu bezpečnější než HTTP.

[12] Z anglického Hypertext Transfer Protocol, jedná se o standardní protokol sloužící ke komunikaci na internetu. Přenos dat není zabezpečený.

- Postman – platforma pro testování mého naimplementovaného API,
- Github Actions – nástroj pro automatické nahrání aplikace na testovací prostředí běžící na cloudu; více v kapitole 3.7,
- Heroku Pipelines – rozhraní pro oddělení testovacího a produkčního prostředí na cloudu; více v kapitole 3.7,
- Database Navigator – plugin vývojového prostředí IntelliJ pro přístup a správu databáze.

3.3 Zvolená architektura

Pro libovolný projekt máme na výběr z několika standardních druhů architektur. Jakožto vývojáři pak musíme uvážit, který druh se nejvíce hodí pro naší konkrétní aplikaci.

Velmi často používanou architekturou je třívrstvá. Ta dělí aplikaci do tří samostatných vrstev, z nichž každá má svou unikátní funkci. Jmenovitě jde o:

- prezentační vrstvu – obstarává zobrazení výsledku uživateli,
- aplikační vrstvu – zajišťuje komunikaci mezi prezentační vrstvou a modelem,
- model – obsahuje logiku aplikace a komunikuje s databází.

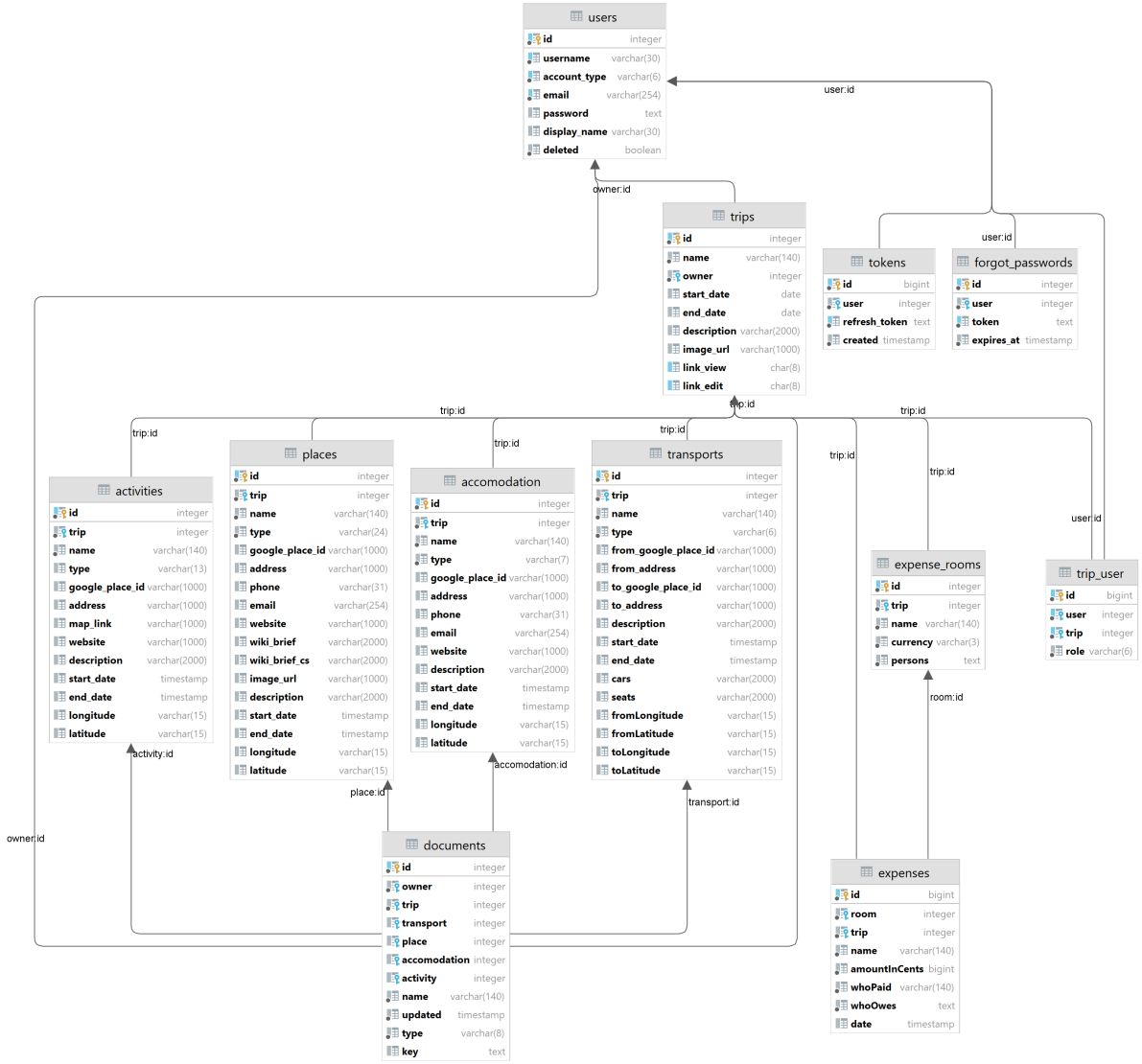
Úkolem webové služby je přijímat požadavky od uživatelů, ty odpovídajícím způsobem zpracovat, a poté na ně adekvátním způsobem odpovědět. Pokud mají požadavky jako v mém případě podobu formátu JSON[13], pak není třeba implementovat prezentační vrstvu. Pro mojí webovou službu tak postačí pouze dvouvrstvá architektura, kde bude *controller* komunikovat s objekty přistupující k databázi.

3.4 Databázový model

Databázi jsem rozvrhl do 12 tabulek. Významnou tabulkou je tabulka *trips*, která reprezentuje cestu. Každá cesta se pak skládá z mnoha bodů zájmů, jež jsou představovány tabulkami *accommodation*, *activities*, *places* a *transports*. Všechny tyto body zájmů se odkazují cizím klíčem na svou cestu. Mimo to se v databázi nachází tabulka *documents*, znázorňující dokumenty. Dokument je možné přidat jak samostatně do výletu, tak zvlášť do bodu zájmu. Proto obsahuje cizí klíče na jednotlivé body (můžou být prázdné) a zároveň cizí klíč na výlet. Poslední tabulka, která stojí za zmínku, je vazební tabulka *trip_user*.

[13] Z anglického JavaScript Object Notation, jedná se o formát určený pro přenos dat v textové podobě, který je čitelný pro člověka a snadno zpracovatelný strojem.

Jak lze z názvu odvodit, tabulka vyjadřuje vazbu mezi uživatelem a cestou a je doplněna o roli, která přísluší uživateli v dané cestě. Jelikož obrázek vydá za tisíc slov, doplňuji tuto práci o podrobný diagram 3.3.



Obrázek 3.3: Databázové schéma webové služby

3.5 Návrh autentizace

V nefunkčním požadavku N7 uvádím, že neautorizovaný přístup v aplikaci bude odepřen a požadavek bez patřičného oprávnění bude odmítnut. Abych dokázal rozlišit, který požadavek smí aplikace legálně obsloužit, a který je naopak neoprávněný, budu muset znát identitu uživatele. Podle identity pak poznám, jaká má uživatel v aplikaci práva a zdali je jeho dotaz validní.

Z toho důvodu se budou muset uživatelé v aplikaci zaregistrovat a pokud chtějí aplikaci využít smysluplně, pak i přihlásit. Pro vyřešení autentizace bych mohl jednoduše využít služby některé z třetích stran, například společnosti Firebase, a byl by to velmi racionální postup. Nicméně jsem se dopředu rozhodl, že si autentizaci napíšu sám, abych nad ní měl větší kontrolu.

Má webová služba se bude řídit architektonickým stylem REST. Tato informace je pro návrh autentizace důležitá, proto je vhodné stručně tento styl nejdříve vysvětlit.

3.5.1 Stručný úvod do RESTu

REST je v současné době velmi populární styl pro tvorbu API a stojí za ním Roy Fielding, který jej poprvé popsal ve své disertační práci v roce 2000. [18] REST umožňuje složitou komunikaci mezi klientem a serverem[14], aniž by klient potřeboval dopředu znát cokoliv o struktuře API. Ve skutečnosti by měl server poskytovat klientovi všechny potřebné informace pro to, aby byl klient schopen se serverem spolehlivě komunikovat. Pokud si server musí pamatovat údaje o předchozích spojeních klienta (například session), pak porušuje zásady RESTu. V praxi je ale někdy složité (nebo i nemožné) některá omezení neporušit.

Styl REST pracuje s tzv. zdroji (resources). Zdrojem může být jakákoliv informace, jež je natolik důležitá, že ji jednoznačně identifikujeme. Protože je koncept zdroje poněkud abstraktní, uvedu raději názorný příklad:

```
https://cvut.cz/fakulty
```

Uvedená URL adresa obsahuje zdroj `fakulty`. Zdroj můžeme jakkoliv pojmenovat, měl by ovšem dávat smysl. V tomto příkladu bychom očekávali, že nám zdroj vrátí seznam fakult na ČVUT.[15]

Nejběžnějším protokolem pro přenos dat je HTTP, ačkoliv to není nutnou podmínkou. Se zdroji pak můžeme pomocí HTTP metod pracovat. K popisu jejich chování nám lépe poslouží ukázka na další straně, jež se nepatrně liší:

[14] Klient a server jsou dvě samostatné instance. Díky jejich oddělení je můžeme vyvíjet nezávisle na sobě a lépe škálovat. Klientem může být například mobilní aplikace.

[15] Ve skutečnosti vrací tento zdroj Stavební fakultu, což je trochu nešťastné.

`https://cvut.cz/fakulta`

Nejčastěji používanými HTTP metodami jsou:

- GET – slouží pro čtení reprezentace zdroje, v naší druhé ukázce bychom očekávali, že nám tato metoda vrátí jednu fakultu.
- HEAD – funguje podobně jako GET, ale odpověď je nám vrácena bez těla. V praxi můžeme touto metodou ověřit existenci zdroje.
- POST – obvykle slouží k vytvoření nového zdroje, v našem příkladu bychom takto otevřeli novou fakultu (údaje o fakultě by byly uloženy v těle požadavku).
- PUT – je analogická metodě POST. Obvykle touto metodou aktualizujeme zdroj. My bychom pomocí ní mohli například upravit název fakulty nebo vyměnit děkana.
- DELETE – slouží k mazání zdroje, v našem příkladu bychom takto zrušili jednu fakultu.

3.5.2 JSON Web Token

Jak jsem popsal dříve, webová služba praktikující styl REST byl neměla nic tušit o minulých spojeních klienta. Co by ale v takovém případě měl klient odesílat na server, aby webová služba rozpoznala jeho identitu? Jednou z možných odpovědí jsou tokeny, konkrétně JWT neboli JSON Web Token.

Na JWT můžeme nahlížet jako na token se speciálními vlastnostmi, který umožňuje bezpečnou výměnu dat mezi dvěma stranami. Skládá se ze tří částí:

- Hlavička (header) – informuje příjemce o použitém hashovacím algoritmu a dalších metadatech, je ve formátu JSON.
- Data (payload) – tato část obsahuje seznam tzv. prohlášení (claims), neboli dvojic klíč-hodnota a je určena pro uložení dat v tokenu. Klíčem může být jakýkoliv text, nicméně autoři JWT několik klíčů zarezervovali pro obecné užití. Mezi takovými klíči kupříkladu najdeme *sub*, *exp* nebo *iat*. Popořadě tyto klíče definují subjekt (např. ID uživatele), dobu platnosti a čas, kdy byl token podepsán. Má formát JSON.
- Podpis (signature) – podpis závisí na typu hashovacího algoritmu, v případě *HMAC-SHA256* je zkonstruován takto:

```
HMAC_SHA256(
  "nejake silne tajne heslo",
  base64(hlavicka) + '.' +
  base64(data)
)
```

Celý token je vytvořen tak, že se všechny tři části zakódují do Base64[16]a spojí tečkami. Ověřování tokenu pak funguje obdobně, jenom v obráceném pořadí. Tvůrce tokenu (např. webová služba) si rozdělí token zpět na tři části a první dvě (hlavičku a datovou část) použije pro výpočet podpisu. Pokud se spočítaný podpis shoduje s původním a tokenu nevypršela platnost, potom je token validní.

```
//hlavicka (header)
{
  "alg": "HS256",
  "typ": "JWT"
}

//data (payload)
{
  "sub": "123",
  "name": "John Doe",
  "iat": 1516239022
}
```

Ukázka kódu 3.3: Hlavička a datová část v JWT

Kupříkladu token po zakódování hlavičky a dat z ukázky 3.3 heslem „nejake silne tajne heslo“ vypadá následovně:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFt
ZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.Uw2xtfqbKyBDy0RWZB2hlbynPT
syngV_v77NtgtiF5o
```

Je důležité zmínit, že dekódování lze provést i bez znalosti tajného klíče, proto by token neměl obsahovat citlivé informace nebo by měly být nejprve zašifrovány. Výhodou je fakt, že obdržená data jsou důvěryhodná. Kdyby útočník data změnil, pak by se změnil i vypočítaný podpis a ten by se tak neshodoval s originálním.

JWT obsahuje všechna potřebná data pro ověření identity a platnosti tokenu, což je ale žádoucí jenom v některých situacích. V případě, že bychom nikdy nepotřebovali libovolný token zneplatnit, si server žádný token nemusí ukládat, neboť veškeré informace načte z tokenu. Takový stav je ideální. Webová služba dodržuje zásady RESTu a v mnoha okolnostech není ani třeba tolikrát přistupovat do databáze. Potíž nastává, pokud potřebujeme znevalidnit JWT před jeho vypršením. Abychom toho dosáhli, musíme token stejně uložit (např. mezi zneplatněné tokeny), čímž zanášíme do webové služby stav.

[16]Kódování, které převádí jakákoliv binární data na alfanumerické znaky a znaky plus, lomítko a rovnítko.

3.5.3 Nesnáze s použitím tokenů

Na použití JWT se mi líbí to, že obecně snižuje nutnost přístupu do databáze. Nicméně je třeba vyřešit jednu nesnáz. Pokud má každý klient u sebe uložen token, a ten zároveň posílá s každým požadavkem na server, pak je zde značná šance, že některému uživateli bude token ukraden. To se může stát například během používání aplikace na veřejné Wifi. V případě krádeže si pak útočník může token donekonečna obnovovat a používat aplikaci pod cizím účtem.

Tento problém se dá vyřešit přidáním druhého tokenu, tzv. refresh tokenu (původně zmiňovaný token se označuje jako access token). Jakmile klientovi vyprchá platnost access tokenu, během jeho obnovy zašle na server refresh token. Server refresh token zkontroluje a pokud je vše v pořádku, vygeneruje nový access token a zašle jej zpět klientovi. Klient si pak nový access token uloží.

Navržený postup se ale stále dá malinko vylepšit. Ke vší smůle totiž může být uživatelovi ukraden i refresh token. Útočník s cizím refresh tokenem se opět může svévolně vydávat za druhého uživatele a vesele si obnovovat access tokeny. Pokud ale během obnovování access tokenu vytvoříme i nový refresh token a starý smažeme, útočníkovi zůstane uložen neplatný refresh token, čímž zamezíme jeho přístupu do aplikace.

Existují různé způsoby, jak odhalit útočníkovu činnost. V každém případě se útočníka zbavíme tak, že ze serveru smažeme (pro jistotu) všechny refresh tokeny poškozeného uživatele.

S výše zmíněným postupem obnovování refresh a access tokenů jsem se spokojil a budu jej používat pro zajištění autentizace a autorizace na mém serveru.

3.6 Implementace

V této podkapitole představuji čtenáři ukázku zpracování standardního požadavku mnou implementované webové služby. Dále se v tomto oddíle zaobírám řešením autentizace a podrobněji popisuji některé realizované části aplikace. Součástí vybraných podkapitol je i příklad práce s ORM frameworkem Exposed. Na konci této sekce pak rozebírám způsob nahrávání dokumentů.

3.6.1 Zpracování požadavku

Úkolem webové služby je přijímat požadavky, ty zpracovat a poté na ně odpovědět. V Ktoru se o přijímaní požadavků stará plugin zvaný *Routing*. Jak jsem již popsal v podkapitole 3.1.2, plugin musíme nejprve v aplikaci nainstalovat.

Celá logika se pak odehrává ve funkci s příhodným názvem *routing*, jež je ve skutečnosti funkce vyššího řádu (anglicky high order function). To znamená, že alespoň jeden z jejích argumentů je jiná funkce. Díky Kotlinu DSL,

který jsem zmínil v podkapitole 3.1.2, nabízí *routing* funkce ve svém těle další metody[17]užitečné pro zpracování požadavku. Takovými volanými metodami jsou v praxi nejčastěji metody korespondující s HTTP.

Nejlepší bude ukázat zjednodušený příklad, jak vypadá v mé webové službě tvorba nové cesty. Principiálně jsou pak podobně řešeny i další požadavky pokrývající zbylé CRUD[18]operace.

Zdroj pro vytvoření nové cesty je identifikován takto: `/trip`. Protože výsledkem úspěšného dotazu je nový zdroj (tedy v ukázce nová cesta), použil jsem HTTP metodu POST. Celý dotaz tak vypadá následovně:

```
POST https://wanderscope.herokuapp.com/trip

Headers
    Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciO...
    Content-Type: application/json

Body
{
    "name": "Cesta na vychod",
    "duration": {
        "startDate": "2022-06-10T00:00:00.00Z",
        "endDate": "2022-06-15T00:00:00.00Z"
    },
    "description": "Bude to moc hezka cesta."
}
```

Ukázka kódu 3.4: Příklad HTTP požadavku pro vytvoření nové cesty

Mezi hlavičkami požadavku si můžete povšimnout hlavičky *Authorization*. Ta obsahuje token identifikující uživatele, jenž si chce cestu přidat. O způsobu řešení autorizace píšu později v podkapitole 3.6.4. Momentálně postačí vědět, že je token typu JWT a zahrnuje ID uživatele. Vytvoření nové cesty pak vypadá následovně:

```
routing {
   val tripController: TripController by inject() //1
   createTrip(tripController) //2
}
```

Ukázka kódu 3.5: Implementace přidání nového výletu v Ktoru

kde tělo funkce `createTrip` v sobě ukrývá:

```
private fun Route.createTrip(tripController: TripController) {
   post("/trip") { //3
       val trip = receive<TripRequest>(MISSING_PARAM).toDto() //4
```

[17]V tomto textu nerozlišuji rozdíl mezi metodou a funkcí.

[18]Create, Read, Update a Delete jsou operace sloužící ke tvorbě, čtení, úpravě a mazání zdrojů.

```
        val result = tripController.createTrip(getUserId(), trip) //5
        respond(result) //6
}}
```

Ukázka kódu 3.6: Příklad zpracování POST požadavku v Ktoru

Pojďme tyto dvě ukázky 3.5 a 3.6 rozklíčovat řádek po řádku. Každý řádek je v ukázce označen unikátním číslem, jež popisuje jeho funkci:

- 1 – vytvoření instance *controlleru* pomocí návrhového vzoru Dependency injection (česky pod méně známým termínem Vkládání závislostí, viz podkapitola 4.7).
- 2 – volání funkce `createTrip`, která je i názorným příkladem použití extension funkce v Kotlinu.
- 3 – metoda z Ktor API. Je zavolána v případě POST dotazu na zdroj `/trip`.
- 4 – funkcí metody `receive` je převést tělo dotazu na objekt (v kontextu OOP), se kterým můžeme v aplikaci lépe manipulovat. Metoda `receive` je součástí Ktor API.
- 5 – *controller* vytvoří novou cestu a výsledek uloží do proměnné.
- 6 – metoda `respond` vrátí odpověď klientovi. Jedná se o poslední metodu z Ktor API v ukázce.

Výše uvedený příklad funguje naprosto v pořádku, nicméně Ktor nabízí experimentální plugin v předběžném přístupu – *Locations*. Ten umožňuje způsob vytváření *routes*[19]odlišně. Prvně si zadefinujeme třídu Trip, jež identifikuje zdroj. Třída může rovněž obsahovat parametry, za které pak Ktor automaticky dosadí parametry z HTTP požadavku. Místo `post("/trip")` na prvním řádku těla funkce `createTrip` tak můžeme použít `post<Trip>`, čímž nám framework u parametrů zajistí správné datové typy.

```
@Location("/trip/{id?}")
data class Trip(val id: Int? = null)
```

Ukázka kódu 3.7: Třída *Trip* identifikuje zdroj. Otazník za *id* značí nepovinný parametr

[19]Slovem *route* označují autoři Ktoru místo pro práci se zdrojem.

3.6.2 Přihlášení a registrace

Implementace v současné verzi mé aplikace podporuje pouze přihlášení a registraci pomocí kombinace emailu a hesla. Na druhou stranu již od začátku implementace backendu jsem počítal s myšlenkou možného rozšíření o jiné metody, zejména přihlášení přes Google účet. Do databáze tak u uživatele spolu s jeho osobními informacemi ukládám i údaj o způsobu přihlášení. Mimo to je heslo v databázi nepovinné, proto půjde později přidat uživatele i bez hesla. Pokud se už bavíme o hesle, pak znalé čtenáře nepřekvapím faktem, že heslo je v databázi zahešované[20]. K tomu používám algoritmus BCrypt, jenž doporučuje organizace OWASP kladoucí si za cíl zvýšení povědomí o zabezpečení softwaru. [19]

Přihlášením i registrací se v mé implementaci zabývá *controller* `EmailPasswordController`. Pro snazší rozšíření o jiné třídy dědí z generického rozhraní `AuthController`, která nabízí dvě metody – `register` a `login`.

```
interface AuthController<T : Login> {
   suspend fun register(credentials: Credentials<T>): Response
   suspend fun login(login: T): Response
}
```

Ukázka kódu 3.8: Rozhraní pro autentizaci uživatelů

Během registračního procesu využije *controller* služby třídy `Validator`, jejíž úkolem je ověřit správnost a omezení zadaného emailu, hesla a uživatelského jména. Pokud `Validator` nenalezl žádnou chybu, pak je uživatel i s zahešovaným heslem uložen do databáze. Uložení má na starost instance rozhraní `UserDao`. Ta implementujte funkci pro přidání nově zaregistrovaného člena do databáze pomocí frameworku Exposed. Funkce vypadá následovně:

```
override suspend fun addUser(
   username: Username,
   accountType: AccountType,
   email: String,
   password: String?
) = transaction {
   UserTable.insertAndGetIdOrNull {
       it[this.username] = username.it
       it[this.accountType] = accountType
       it[this.email] = email
       it[this.password] = password
   }?.value ?: throw
       BadRequestException(FailureMessages.ADD_USER_FAILURE)
```

[20]Hešování je převod libovolných vstupních dat na výstupní data o přesné délce. Z výstupních dat bychom správně neměli rozpoznat vstupní data. Hešování tvoří základ kryptografie.

```
}
```

Ukázka kódu 3.9: Přidání uživatele do databáze v Exposed

V ukázce 3.9 si můžete povšimnout metody `transaction`. Ta je klíčovou metodou v Exposed a umožňuje provést příkazy v jedné databázové transakci, která zajistí, že databáze bude ve stále konzistentním stavu. Transakce se buď provede celá, nebo vůbec, a je odstíněna od jiných probíhajících transakcí. V důsledku si tak současně přistupující klienti k databázi laicky řečeno „nepolezou do zelí".

Metoda `transaction` je v Exposed blokující, naštěstí ale autoři frameworku později vydali novou metodu postavenou na korutinách. `Transaction` v ukázce 3.9 je tak ve skutečnosti má vlastní naimplementovaná funkce, která na pozadí ukrývá právě neblokující verzi (viz ukázka 3.10).

```
suspend inline fun <T> transaction(
   dispatcher: CoroutineDispatcher = Dispatchers.IO,
   crossinline query: suspend () -> T
): T {
   return newSuspendedTransaction(dispatcher) {
       query.invoke()
   }
}
```

Ukázka kódu 3.10: Metoda *transaction* vytváří neblokující databázovou transakci

V podkapitole o výběru databáze 3.1.3.1 jsem napsal, že při práci s Exposed budu používat syntaxi zdánlivě podobnou SQL. Z první ukázky 3.9 to příliš patrné není, nicméně další ukázka z mého zdrojového kódu 3.11 je již mnohem přesvědčivější:

```
private suspend fun getTrips(
   selectWhere: Op<Boolean>,
   column: Expression<*>,
   orderBy: SortOrder
): List<TripOverviewDto> {
   val query = TripTable.innerJoin(TripUserTable)
       .join(UserTable, JoinType.INNER, TripUserTable.user,
           UserTable.id)
       .slice(TripTable.columns + TripUserTable.columns)
       .select(selectWhere)
       .orderBy(column, orderBy)
       .withDistinct()
//...
}
```

Ukázka kódu 3.11: Spojení dvou tabulek v Exposed pomocí syntaxe podobné SQL

Po úspěšném uložení uživatele vygeneruje webová služba nový refresh a access token. K vytvoření tokenů mi posloužila knihovna JWT, jež je součástí pluginu pro autentizaci. Refresh token popravdě nemusel být typu JWT, ale přišlo mi snazší jej vygenerovat s použitím již přítomné knihovny. Během jeho generování vkládá má aplikace do datové části náhodný řetězec, aby tak minimalizovala riziko, že pro jednoho uživatele vygeneruje v jeden moment více stejných tokenů (například během obnovování access tokenů na více zařízeních uživatele přesně ve stejný čas).

Refresh tokenu jsem nastavil platnost na 90 dnů. To znamená, že uživatel který nepoužije aplikaci déle jak tři měsíce, se bude muset poté znovu přihlásit. U access tokenu jsem pak nastavil platnost na 8 minut. Proč ale právě toto specifické (a pro některé národy speciální) číslo?

Jak jsem se rozepsal v podkapitole o JWT, viz 3.5.2, zneplatnit platný JWT token bez uložení do databáze není možné. Jeho uložením ale token ztrácí výhodu bezestavového způsobu autorizace. Proto jsem se rozhodl, že jej ukládat nebudu a v krajním případě bude 8 minut platný. Osm minut mi zároveň připadala adekvátní doba na to, aby byl token obnoven během občasného užívání aplikace jenom jednou[21](vlastně tím předpokládám, že uživatel stráví v aplikaci maximálně kolem osmi minut v kuse).

Jakmile jsou oba dva tokeny úspěšně vytvořeny, uloží webová služba request token do databáze. Během obnovy access tokenu musí klient zaslat serveru i refresh token a právě ten je pak porovnán s uloženým tokenem. Pokud se token v databázi nenachází, pak musel být uživatel někdy odhlášen nebo mu bylo změněno heslo. Jinými slovy nemá patřičné oprávnění pro zobrazení zdroje a jeho požadavek je odmítnut.

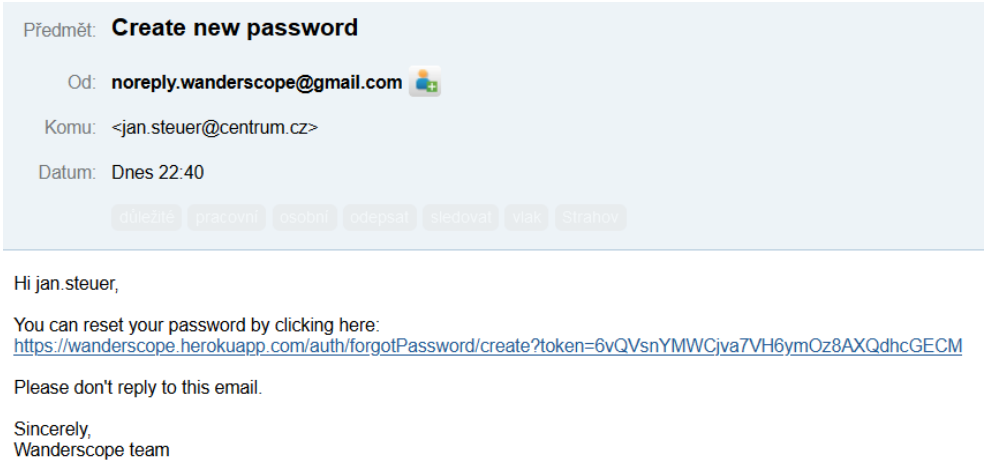
Po uložení refresh tokenu je celý požadavek přihlášení nebo registrace hotov a služba klientovi zašle oba dva vygenerované tokeny.

3.6.3 Zapomenutí hesla

Pokud uživatel nedopatřením zapomene heslo, může si jej jednoduše obnovit. V pozadí aplikace se ale děje poměrně složitý proces. Celý proces začíná tak, že webová služba zkontroluje, zdali uživatelem zadaný email v aplikaci existuje. Pokud ano, vygeneruje speciální token o délce 32 znaků. Ten zahešuje algoritmem HmacSHA256 a uloží do tabulky zapomenutých hesel v databázi. Mimo uložený token obsahuje tabulka cizí klíč na uživatele a čas, kdy token expiruje. Dobu platnosti jsem nastavil z bezpečnostních důvodů na 30 minut, po této době přestane token platit a uživatel si musí nechat zaslat nový email.

Aplikace po uložení informací do databáze odešle email, jehož součástí je URL adresa pro resetování hesla.

[21]Naštěstí je obnova tokenu v produkční verzi aplikace rychlá a zabere v průměru jenom 100 ms.



Obrázek 3.4: Ukázka příchozího emailu po zapomenutí hesla

Pokud uživatel otevře odkaz uvedený v emailu (viz obrázek 3.4), webový server vygeneruje velmi základní HTML[22] formulář pro zadání a potvrzení nového hesla (viz 3.5). Formulář jsem naprogramoval čistě v Kotlinu díky, ano, patrně jste uhodli, DSL (konkrétně HTML DSL). Po kliknutí na tlačítko „resetovat heslo“ je zavolán třetí, tentokrát poslední endpoint[23].

Webová služba nejprve z URL adresy extrahuje token (pomocí pluginu *Locations*) a voláním funkce `receiveParameters()` získá z formuláře heslo. Poté token zahešuje a tento zahešovaný token vyhledá v databázi. Pokud token nalezne, pak zkontroluje dobu expirace a v případě úspěchu změní uživatelovi heslo. Následně služba smaže z databáze záznam o změně hesla a rovněž uživatelovi odstraní všechny uložené refresh tokeny. Tím docílíme odhlášení uživatele ze všech zařízeních.

Nabízí se dobrá otázka, proč je token v databázi zahešovaný? Není překvapivé, že kdokoliv, kdo má platný token, může změnit uživateli heslo. Velký bezpečnostní problém nastává, pokud útočník získá přístup do naší databáze. Takový útočník by si totiž mohl snadno generovat nové tokeny (zapomenutím hesla uživatelů) a poté měnit nešťastníkům aplikace hesla. Pokud ale do databáze uložíme zahešovaný token, útočník se nikdy nedozví, jak vypadá token ve skutečnosti a nikdy nebude moci komukoliv heslo upravit.

3.6.4 Přístup k zabezpečenému zdroji

Veškerý složitý proces registrace, přihlášení, zapomenutí hesla, změny hesla, odhlášení se ze všech zařízeních a podobně by byl zbytečný, kdybychom za-

[22] Z anglického Hypertext Markup Language, jedná se o značkovací jazyk pro tvorbu webových stránek.

[23] Endpoint lze chápat jako nadmnožinu zdroje. Na rozdíl od zdroje není vázán na styl REST a identifikuje místo, kde je služba dostupná, například cvut.cz/fakulty.

Obrázek 3.5: Webová stránka pro resetování hesla je velmi jednoduchá

bezpečené zdroje neochránili. K tomu v Ktoru slouží *Authentication* plugin (viz ukázka kódu 3.12). V rámci takového pluginu pak můžeme instalovat další konkrétní mechanismy pro zajištění autentizace. Pokud není autentizace úspěšná, server vrátí klientovi stavový kód HTTP *401 – Unauthorized*.

```
authentication { //instalace pluginu
   jwt { //autentizace pomoci JWT
       verifier(jwtController.getVerifier()) //overuje format a
           podpis tokenu
       validate { //validace datove casti JWT
           val userId = it.payload.subject //sub claim
           if (userId != null) {
               JWTPrincipal(it.payload) //JWT prihlaseny uzivatel
           } else {
               null
           }
       }
   }
}
```

Ukázka kódu 3.12: Instalace a použití JWT v Ktoru

Zabezpečení zdroje je pak už snadné:

```
routing {
   authenticate {
       //...
   }
}
```

Ukázka kódu 3.13: Zabezpečení zdroje v Ktoru

3.6.5 Nahrávání dokumentů

Uživatelé mohou v aplikaci nahrávat a stahovat dokumenty. Faktem je, že nahrávání dokumentů se dělí na dva požadavky. Pokud je první požadavek zdařilý, jsou výsledkem uložená metadata v databázi. Součástí metadat dokumentu je i klíč. Klíč je důležitý, neboť každý dokument může uživatel před nahráním zaheslovat. K souboru pak mají přístup pouze ti výletníci, kteří daný klíč znají.

Volání v pořadí druhého požadavku závisí na klientovi, ale správná implementace by měla být taková, že jej klient provede okamžitě po získání odpovědi na první požadavek. V případě úspěchu prvního dotazu totiž server vrátil klientovi ID nového souboru, které nyní poslouží pro nahrání skutečných dat.

Obecný postup, jak pomocí HTTP poslat libovolná data nebo soubory na server, je ukryt v názvu *multipart/form-data*. Jedná se o POST požadavek, jehož tělo má ale speciální formát. Tělo se skládá z kombinace bajtů a dalších údajů o souborech a je zpracovatelné strojem, nicméně pro čtenáře moc smysl nedává. Výhodou je, že *multipart* zvládne zkombinovat několik souborů do jednoho požadavku. [20]

Popsaný postup jsem aplikoval i do mé webové služby. Po ověření velikosti (maximální velikost souboru jsem nastavil z praktických důvodů na 10 MB) a přípony souboru pak webová služba nahraje dokument pod svým identifikačním číslem na cloudové úložiště Amazon S3. Amazon vydal sadu metod (SDK[24]) pro snadnější práci s jejich cloudem, nicméně nabízené funkce balíčku nejsou pro tento text příliš zajímavé.

Během stahování dokumentu provede webová služba kontrolu klíče. Pokud je dokument zaheslován, musí klient v GET požadavku zaslat mnou nadefinovanou HTTP hlavičku s validním klíčem (hlavičku jsem pojmenoval *Wanderscope-Document-Key*). Zaslaný klíč je poté na serveru zahešován a porovnán oproti uloženému klíči v databázi. Pokud se jejich heše shodují, webová služba zavolá adekvátní metodu z SDK a lokálně k sobě stáhne obsah dokumentu. Následně získaný obsah přepošle jako pole bajtů klientovi.

3.7 Nasazení aplikace

Během implementace jsem aplikaci lokálně testoval u sebe. Nicméně, aby byla webová služba v praxi použitelná, je třeba ji zpřístupnit všem.

Líbí se mi myšlenka nasazení aplikace na cloud. Hostování v cloudu je metoda používání online virtuálních serverů, které lze podle potřeby vytvářet, modifikovat a mazat. Zdroje, jako jsou CPU[25]jádra a paměť, alokuje serverům na cloudu fyzický server, který je hostí. Server může být zároveň nakonfigu-

[24]Z anglického Software Development Kit, jedná se o sadu nástrojů pro vytváření aplikací a práci s nimi.

rován podle toho, jaký zvolí vlastník operační systém a doprovodný software. [21]

Dalším evolučním stupněm ve vývoji v cloudu je tzv. PaaS, anglicky Platform as a Service (platforma jako služba). PaaS poskytuje uživatelům architekturu, která usnadňuje vývoj cloudových aplikací. [22] To znamená, že vývojáři mají během vývoje k dispozici řadu předpřipravených databází, API, aplikací a dalších nástrojů. Tyto nástroje pak vývojářům usnadňují a zrychlují vývoj jejich vlastních produktů. Mimo to se uživatelé nemusí zabývat správou sítí, serverů ani jiných systémů, neboť veškerou infrastrukturu řeší právě poskytovatel platformy. Nevýhodou ale je, že použití PaaS je zpravidla zpoplatněno.

Obrázek 3.6: Rozdíl mezi IaaS (Infrastructure as a Service), PaaS (Platform as a Service) a SaaS (Software as a Service). Zleva doprava: hostované aplikace; nástroje pro vývoj, správa databází, obchodní analýzy; operační systémy; servery a úložiště; zabezpečení a brány sítí; fyzický provoz/budova datového centra [22]

Pro můj projekt jsem se rozhodl využít platformu Heroku. Heroku je jedním z prvních poskytovatelů PaaS na trhu. [23] Svým uživatelům nabízí mnoho funkcí, z nichž za zmínku stojí integrace s GitHubem, automatická správa databáze Postgres a různé doplňky (anglicky add-ons) pro vylepšení vývoje a údržby aplikace. Rozhodujícím faktorem pro mě byla cena. Abych byl přesný, pro velikost mého projektu je použití platformy zcela zdarma. To s sebou ale nese řadu menších či větších omezení. Například databázi není možné zálohovat, má limitovanou velikost a nepodporuje šifrování.

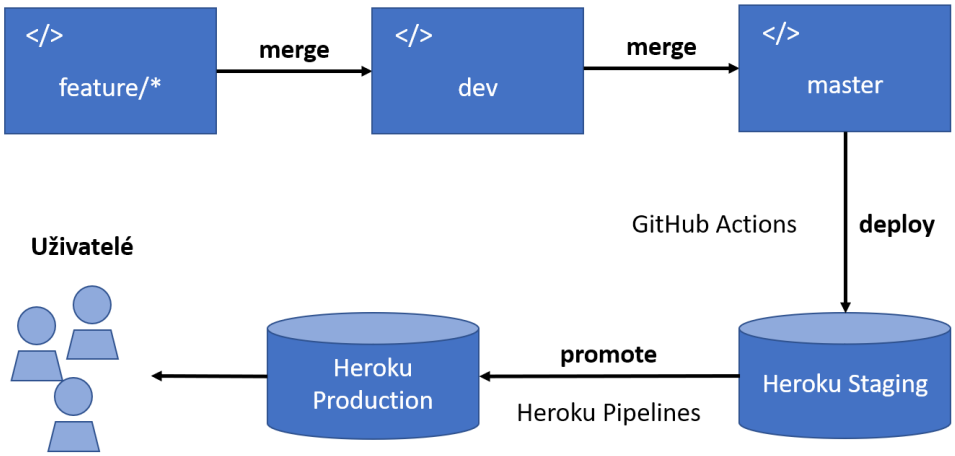
Největším nedostatkem je ovšem fakt, že se nasazená aplikace po 20 minutách neaktivity uspí. Probuzení pak trvá až kolem půl minuty, což není uživatelsky příliš přívětivé. Naštěstí jsem se ale dokázal tohoto omezení zbavit. Paradoxně mi k tomu pomohlo samotné Heroku. Platforma totiž jako jeden z doplňků nabízí tzv. *scheduler* – plánovač procesů. Doplněk jsem si tedy nainstaloval a nastavil tak, aby proces každých 10 minut zaslal dotaz na

[25] Z anglického Central Processing Unit, jedná se o součástku v počítači, která vykonává strojové instrukce.

mojí webovou službu. Služba je díky tomu stále aktivní a Heroku proto server neuspí.

Abych mohl vyvíjet aplikace i v budoucnu bez omezení aktivních uživatelů, nahrál jsem na Heroku ve skutečnosti dvě aplikace – testovací a produkční. Každou aplikaci tvoří jiné endpointy, mají vlastní databáze a používají oddělené úložiště na Amazonu. Prováděné změny v testovací verzi (tzv. *staging*) aplikace tak nijak neovlivní uživatele produkční verze. Zároveň jsem díky užití Heroku Pipelines schopen změny v kódu testovací verze snadno zpropagovat do verze produkční.

Pro rychlejší a pohodlnější nasazení používám ve svém GitHub repozitáři GitHub Actions. GitHub Actions pomáhají s automatickým sestavováním (build) aplikací, testováním (test) a nasazením (deploy). [24] V mém případě GitHub Actions kontrolují, zdali nebyly provedeny změny na mé hlavní (master) větvi v repozitáři a pokud ano, sestaví aplikaci a nasadí ji na testovací prostředí na Heroku. Novou funkcionalitu jsem tak schopen rychle a bezpečně doručit uživatelům.



Obrázek 3.7: Diagram znázorňující proces doručení nové funkce uživatelům

Kapitola 4

Mobilní aplikace

V této kapitole seznamuji čtenáře s vyvíjenou mobilní aplikací. V úvodu se rozepisuji o vybraném programovacím jazyce a minimální verzi Android API. V následující části textu rozebírám návrh uživatelského rozhraní s přiloženými snímky finální podoby aplikace. Po uživatelském rozhraní pak vysvětluji čtenáři doporučené rozvržení Android aplikace s konkrétními příklady použití nejen v mé práci. V neposlední řadě představuji v této kapitole vybrané technologie a principy, jež jsem aplikoval do mého řešení. V samotném závěru pak na konkrétní ukázce z kódu vysvětluji fungování mé aplikace.

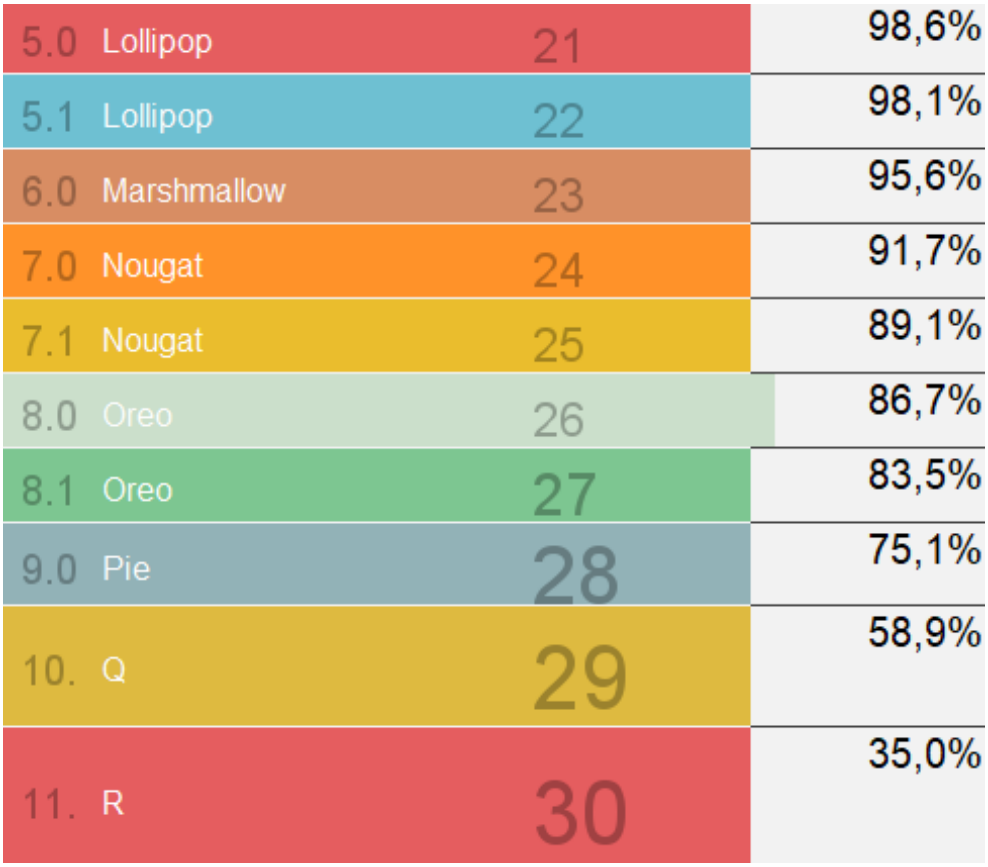
4.1 Programovací jazyk

V současné době je Kotlin doporučovaným jazykem pro vývoj v Androidu. Na oficiálních stránkách pro Android vývojáře se můžeme dočíst: „Kotlin je expresivní a stručný programovací jazyk, který snižuje běžné chyby v kódu. Pokud chcete vytvořit aplikaci pro Android, doporučujeme začít s Kotlinem.“ [10] Dalším velmi významným argumentem je fakt, že projekty v Jetpack Compose – revolučním způsobu vývoje Android aplikací – lze programovat již pouze v Kotlinu. [10] Z výše zmíněných důvodů a mé osobní inklinaci k tomuto jazyku jsem si vybral také Kotlin.

4.2 Minimální verze

Operační systém Android je na trhu od roku 2008 a za tu dobu už autoři představili mnoho verzí. [25] Každá verze přidává do systému nové funkce a upravuje ty stávající. Všichni vývojáři si tak musí před vývojem aplikace rozmyslit, jakou minimální verzi bude jejich aplikace podporovat. Čím vyšší zvolí verzi, tím zpravidla snazší budou mít vývoj, neboť mnoho funkcí bylo přidáno až v pozdějších verzích API. Na druhou stranu, ne všichni uživatelé

mají na svých zařízeních nainstalovanou aktuální verzi Androidu, a tak se vyvíjená aplikace dostane k menšímu množství potenciálních zákazníků.



Obrázek 4.1: Rozdělení jednotlivých verzí API k 9. květnu 2022 [26]

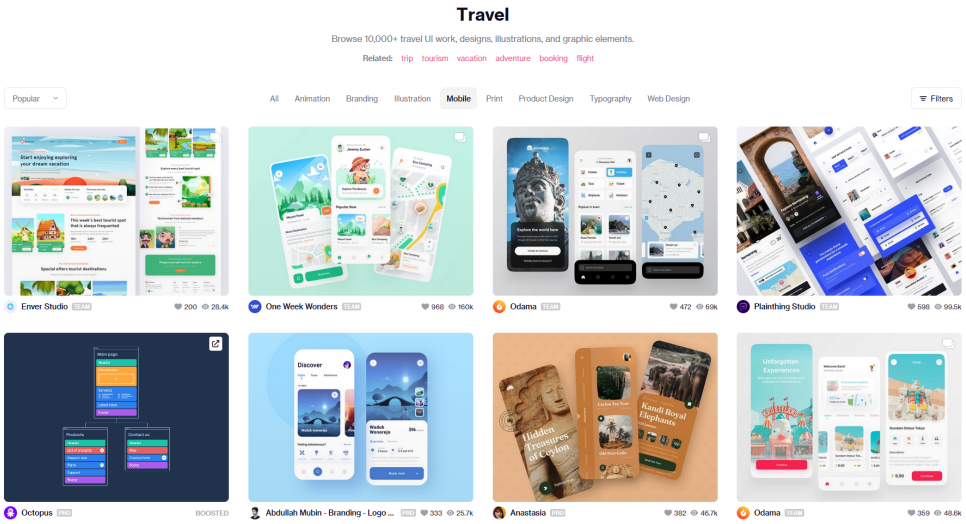
Mojí mobilní aplikaci půjde nainstalovat na telefony s minimální verzí API 26 (Android 8.0 Oreo) a výše. Dle posledních dostupných dat 4.1 používá verzi Oreo a novější necelých 87 % uživatelů Androidu. Jedná se tak o přijatelný kompromis mezi dostupností pro uživatele a nabízenými funkcemi.

4.3 Návrh uživatelského rozhraní

Mým cílem je vytvořit mobilní aplikaci, která bude pro uživatele jednoduchá, přehledná a intuitivní. Pro inspiraci jsem navštívil webovou stránku Dribbble.com, která nabízí nepřeberné množství ilustrací, skic, wireframů a návrhů UI[26]. Webová stránka mě pak silně ovlivnila při designování mé aplikace.

[26] Z anglického User Interface – uživatelské rozhraní.

Protože uživatelé vlastní různé typy mobilů a tabletů s různě velkými obrazovkami, navrhl jsem mobilní aplikaci s použitím responzivních prvků. Při návrhu jsem rovněž počítal s tím, aby aplikace zobrazovala uživatelům pouze adekvátní grafické elementy. Například pokud uživatel přidá hotel bez uvedené lokality, pak se v přehledu onoho ubytování neobjeví žádná mapa (namísto prázdné mapy).



Obrázek 4.2: Ukázka webové stránky Dribbble.com

Aplikace bude také muset umět rozlišovat jednotlivé uživatelské role v rámci cest. To znamená, že kupříkladu uživatel s rolí diváka neuvidí v aplikaci ty UI prvky, kterými by mohl ovlivnit průběh výletu. Mimo to jsem cílil i na přehledné a jasné chybové hlášky, jež se zobrazí na adekvátních místech. V neposlední řadě jsem myslel na to, aby uživatel mohl kliknout na akční tlačítko (např. odeslat) až po správném vyplnění konkrétních položek ve formuláři.

4.3.1 Název aplikace

Název je neoddělitelnou součástí identity aplikace. Ideální by bylo, aby byl výstižný, ne moc dlouhý, a pokud možno odlišitelný od ostatních[27]. Po intenzivním brainstormingu a popsané straně ne příliš zdařilých nápadů mi zůstaly na výběr dva adepti – Lembas a Wanderscope. Pro neznalé Pána prstenů je Lembas výživný elfský chléb, který s sebou dva protagonisté, Frodo a Sam, nesli během cesty do Mordoru. Chléb jim na výpravě dodával potřebnou sílu pro pokračování a nebýt jej, možná by nedošli, ale pošli. Název by tak v přeneseném významu vystihoval fakt, že aplikace pomáhá poutníkům na jejich cestě. Ke vší smůle již ovšem existuje aplikace s podobným názvem – Lambus (viz podkapitola 1.4). Proto jsem se rozhodl pro druhou možnost – Wanderscope. *Wander* znamená v angličtině *toulat se*, spojením se *scope* –

Název	Kód	Vizualizace
colorPrimary	#2b54ef	
colorSecondary	#f4f8fb	
colorBackground	#ffffff	
colorText	#000000	

Tabulka 4.1: Vybrané barvy aplikace

v češtině *prostor*, *příležitost* (pro činnost ap.) – vznikne příležitost k toulání se – *Wanderscope*. Zároveň zní název podobně jako například *gyroscope*, *telescope* – tedy nástroje, jež dříve užívali námořníci a dobrodruzi. Sečteno podtrženo, má aplikace se bude jmenovat Wanderscope.

4.3.2 Barevná paleta a font

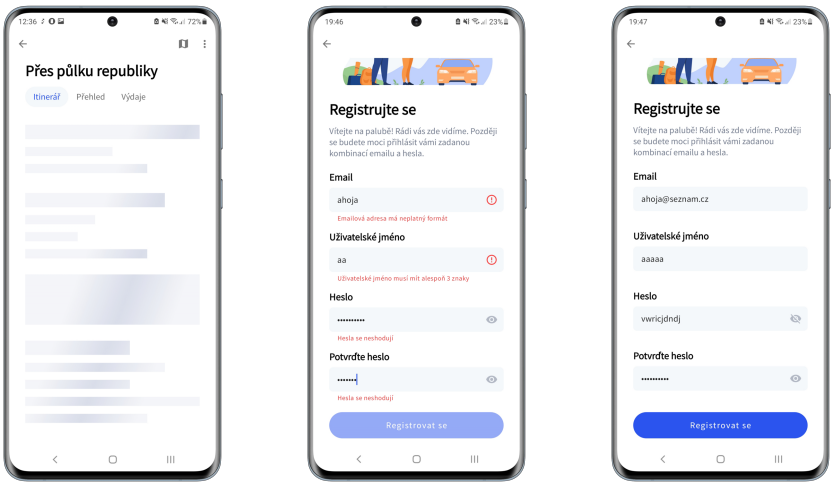
Předpokládám, že uživatelé budou často používat aplikaci ve venkovním prostředí. Pro lepší čitelnost na přímém světle tak bude uživatelské rozhraní vedeno primárně ve světlých barvách s kontrastními prvky vyvolávajícími nějakou akci (například tlačítka). To v praxi znamená, že pozadí aplikace bude bílé, významné a zpravidla klikatelné prvky modré, text primárně černý a méně důležité grafické komponenty budou zbarveny do jemného odstínu pomněnkové (viz tabulka 4.1). Jako font jsem se rozhodl použít *Source Sans Pro*, který je dostupný na webové stránce Google Fonts pod open source licencí. Tento font vypadá moderně, je dobře čitelný, a skvěle tak funguje při návrhu uživatelského rozhraní.

4.3.3 Základní rozložení

K navigaci, tedy k přecházení mezi jednotlivými obrazovkami v aplikaci, poslouží tzv. *bottom navigation bar* – navigační UI komponenta umístěná na spodku obrazovky. Bude se skládat ze tří ikonek – *Domů*, *Cesty* a *Účet*. Po kliknutí na ikonku pak bude uživatel přesměrován na patřičnou obrazovku. Jinými slovy, pokud uživatel klikne například na *Účet*, zobrazí se mu v aplikaci obrazovka s jeho účtem. Tento způsob navigace je v dnešní době mezi aplikacemi běžný a pro uživatele intuitivní.

Jakmile uživatel opustí jednu ze tří výše zmiňovaných obrazovek, navigační řádek se skryje a pro pohyb mezi obrazovkami poslouží *toolbar* – další z navigačních komponent nacházející se pod řádkem s notifikacemi a časem. Kliknutím na interaktivní prvek (například bod zájmu) se uživateli zpravidla otevře nová obrazovka s podrobnějšími informacemi. Pro návrat zpět pak bude moci využít ikonu šipky v *toolbaru*. Výhodou *toolbaru* je i fakt, že může obsahovat rozbalovací menu, další ikonky, název a podobně. Právě rozbalovací

[27] Berte prosím na vědomí, že toto jsou pouze mé subjektivní domněnky.



Obrázek 4.3: Ukázka výsledné aplikace: obrázek vlevo zachycuje načítání; prostřední obrázek ukazuje způsob zobrazení chybových hlášek; obrázek vpravo znázorňuje stav, kdy uživatel vyplnil vše správně

menu v podobě tří teček bude nedílnou součástí mnoha obrazovek. Po jeho rozkliknutí nabídne uživateli doplňující funkce v kontextu dané obrazovky (například editaci či smazání cesty). Pro pohyb mezi itinerářem, přehledem a výdaji během organizování cesty pak poslouží záložky – *tabs*.

4.3.4 Přihlášení a registrace

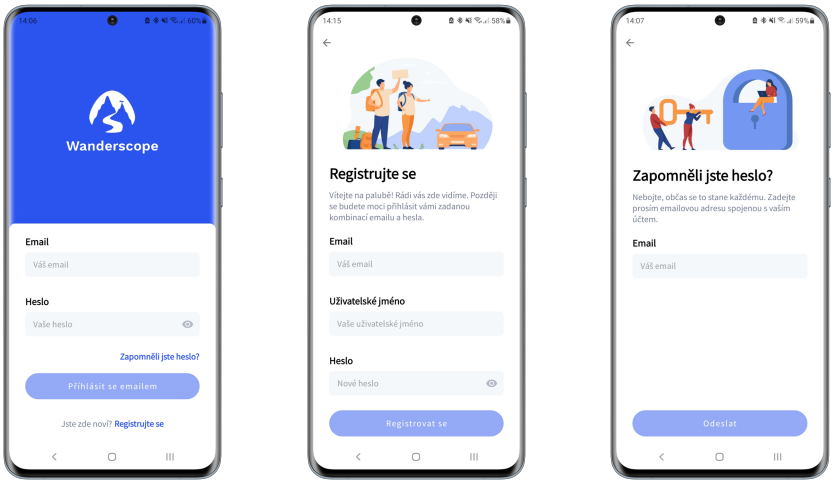
Nepřihlášený uživatel bude v aplikaci přesměrován na úvodní přihlašovací obrazovku. Na ní se bude smět přihlásit, potažmo zaregistrovat. V případě zapomenutí hesla si zde bude moci nechat zaslat email pro resetování hesla. Pro lepší představu vizte ukázku 4.4.

4.3.5 Domů

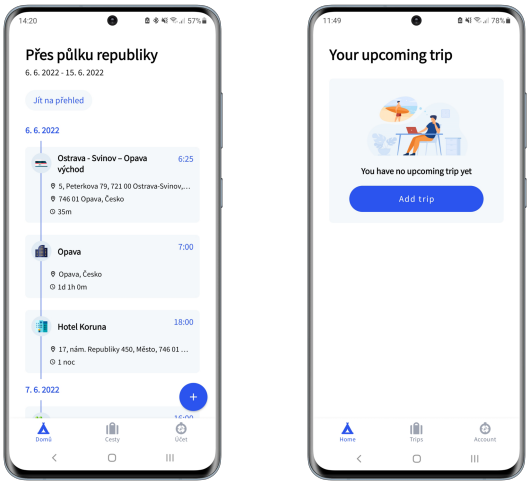
Obrazovka *Domů* nabídne přihlášenému uživateli itinerář s posledně přidanou nadcházející cestou. Pokud bude mít uživatel v dané cestě adekvátní oprávnění, bude moci z *Domů* přidávat i nové body zájmů. Z domovské obrazovky rovněž půjde přímo rozkliknout přehled vybrané cesty. Zároveň počítám s tím, že v případě rozšíření aplikace o nové funkce provedu na této obrazovce poměrně velké grafické změny. Pro lepší představu vizte ukázku 4.5.

4.3.6 Cesty

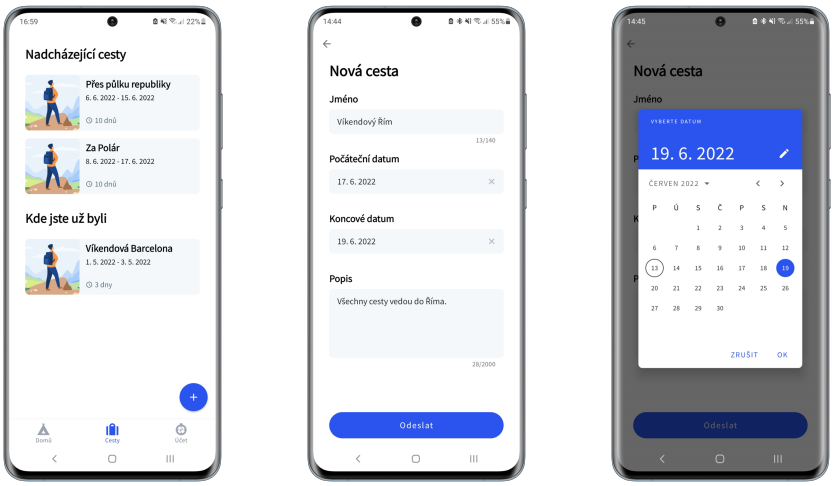
Obrazovka *Cesty* poslouží k jednoduchému účelu, a to zobrazení seznamu uživatelových nadcházejících a uskutečněných cest. Uživatel bude také moci



Obrázek 4.4: Ukázka výsledné aplikace: zleva doprava je zobrazena obrazovka přihlášení, registrace a zapomenutí hesla



Obrázek 4.5: Ukázka výsledné aplikace: vlevo je zachycena domovská obrazovka; obrázek vpravo znázorňuje prázdnou domovskou obrazovku, jak si lze povšimnout, aplikace je v češtině i angličtině



Obrázek 4.6: Ukázka výsledné aplikace: obrázek vlevo zobrazuje nadcházející a proběhlé cesty; prostřední obrázek ukazuje formulář pro přidání nové cesty; obrázek vpravo zachycuje kalendář při výběru data návratu

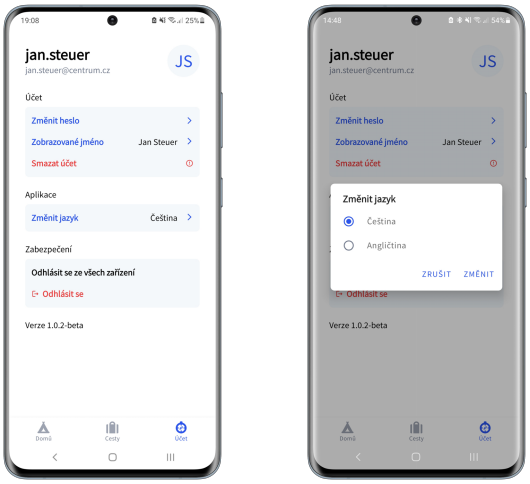
kliknutím na plovoucí tlačítko (*floating action button*) vytvořit novou cestu. Pro lepší představu vizte ukázku 4.6.

4.3.7 Účet

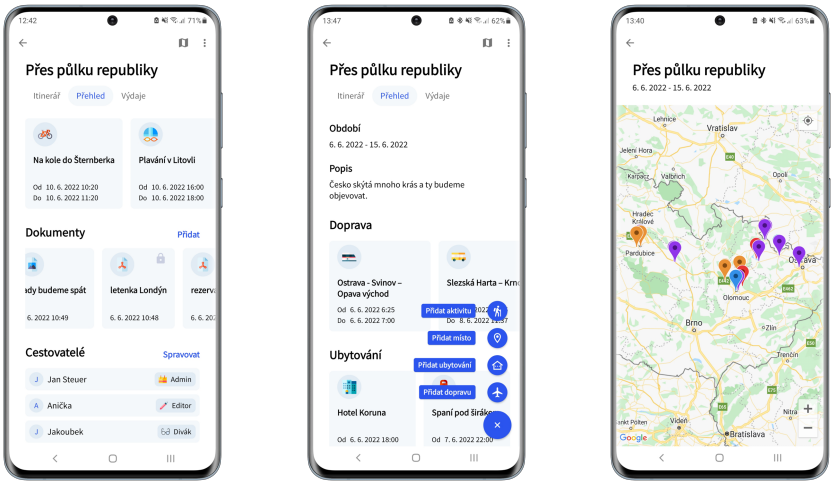
Na obrazovce *Účet* bude smět uživatel provádět změny týkající se jeho účtu. Jmenovitě půjde o změnu hesla, nastavení zobrazovaného jména a smazání účtu. Tato obrazovka také poskytne uživateli možnost odhlášení se z účtu a všech zařízení. Dále si zde uživatel bude moci přizpůsobit aplikaci, ať už například změnou jazyka či v pozdější verzi nastavením světlého/tmavého režimu. Pro lepší představu vizte ukázku 4.7.

4.3.8 Přehled cesty

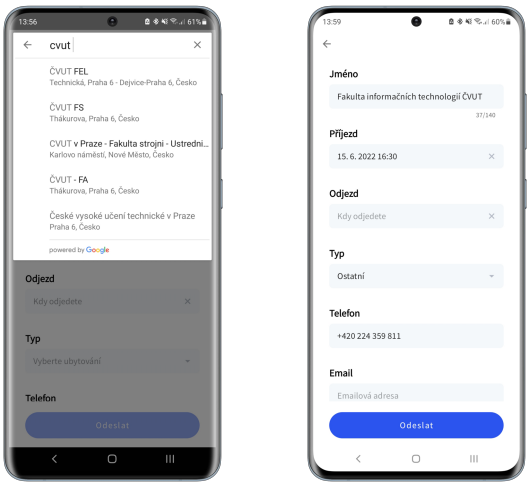
Přehled cesty nabídne podrobnější informace o naplánované cestě. Mimo to poskytne členům výletu seznam všech bodů zájmů podle kategorie, všechny přidané dokumenty k cestě (i když je dokument vázán pouze k bodu zájmu) a seznam dalších spoluúčastníků výpravy. Kterýkoliv člen si také bude moci rozkliknout mapu se všemi přidanými body. Admini a editoři navíc uvidí na této obrazovce plovoucí tlačítko pro přidání bodu zájmu. Aby tlačítko na obrazovce nezakrývalo některé informace, při tažení (*scroll*) obrazovky dolů zmizí. Pohybem nahoru se opět objeví. Po kliknutí na tlačítko se následně rozbalí nabídka s přidáním konkrétního typu aktivity (tj. přidání aktivity, místa, ubytování nebo dopravy). Pro lepší představu vizte ukázku 4.8.



Obrázek 4.7: Ukázka výsledné aplikace: obrázek vlevo představuje obrazovku s účtem; obrázek vpravo zachycuje dialog pro změnu jazyka v aplikaci



Obrázek 4.8: Ukázka výsledné aplikace: obrázek vlevo zobrazuje část přehledu konkrétní cesty; prostřední obrázek zachycuje rozbalovací nabídku pro přidání bodu zájmu; obrázek vpravo znázorňuje mapu se všemi přidanými body



Obrázek 4.9: Ukázka výsledné aplikace: obrázek vlevo ukazuje Google našeptávač během přidání bodu zájmu; obrázek vpravo zachycuje některé položky vyplněné našeptávačem

4.3.9 Přidání bodu zájmu

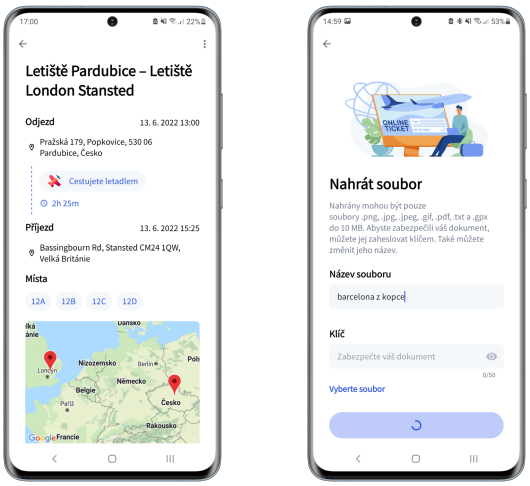
Pro přidání bodu zájmu poslouží formulář s rozličnými položkami podle typu bodu. Společným jmenovatelem ale bude políčko pro vyplnění adresy místa. Po kliknutí na políčko se otevře kontextová nabídka s vyhledáváním na Google. Jakmile si uživatel vybere místo z nabídky, Google pak předvyplní některé zbylé položky formuláře sám (například telefon, webovou stránku a podobně). Pro lepší představu vizte ukázku 4.9.

4.3.10 Přehled bodu zájmu

V přehledu o bodu zájmu se uživatel dozví veškeré podrobnosti o daném bodu. K dispozici bude také mapka s vyznačeným místem. Navíc půjde ke každému bodu zájmu nahrávat dokumenty zvlášť. Zobrazená textová políčka budou vesměs klikatelná a přesměrují uživatele do jiných aplikací. Pro ilustraci, pokud například uživatel v ukázce 4.10 klikne na „Cestujete letadlem“, pak ho aplikace přesměruje do Google Map a rovnou zobrazí spoje mezi Pardubicemi a Londýnem. Pro lepší představu vizte ukázku 4.10.

4.4 Rozvržení aplikace

Android aplikace se obvykle skládá z několika komponent – stavebních bloků – jež zajišťují její správné fungování a komunikaci s operačním systémem a jinými aplikacemi. Jednotlivé komponenty musí vývojář deklarovat ve speciál-



Obrázek 4.10: Ukázka výsledné aplikace: obrázek vlevo zobrazuje obrazovku bodu zájmu, zde konkrétně dopravy; obrázek vpravo zachycuje proces nahrávání dokumentu

ním souboru, tzv. *manifestu*. Operační systém (Android) z tohoto souboru vyčte základní informace o aplikaci a rozhodne se, jak vyvíjenou aplikaci integruje do zařízení uživatele. [27]

Každé mobilní zařízení má omezenou paměť, kterou musí operační systém mezi běžícími procesy a procesy na pozadí vhodně přerozdělovat. Z toho důvodu se může snadno stát, že Android naší aplikaci (zpravidla na pozadí) odebere paměťové prostředky. To má za následek smazání neuložených dat. Takovou událost nemůžeme jako vývojáři ovlivnit, nicméně vhodně zvolenou architekturou a doporučenými postupy lze zajistit, aby uživatelé byli s používáním aplikace za všech okolností spokojeni.

4.4.1 Oddělení zodpovědností

Oddělení zodpovědností (anglicky separation of concerns) je v programování přístup, jak aplikaci vhodně rozdělit na samostatné části tak, aby každá část vykonávala pouze své funkce. Jinými slovy by se měly různé části aplikace z hlediska funkcionality co nejméně překrývat. [28]

Základními stavebními jednotkami v Androidu jsou aktivity a fragmenty – třídy poskytující UI. V rámci oddělení zodpovědností by se tyto třídy měly zabývat pouze vykreslováním grafického rozhraní, interakci s uživatelem a operačním systémem. Ve skutečnosti totiž tvoří aktivity a fragmenty jakési lepidlo mezi naší aplikací a OS[28]. Nesmíme ovšem zapomenout, že operační systém může tyto komponenty na základě interakce uživatele a systémových

prostředků kdykoliv zničit. V praxi bychom tak měli omezit závislost na těchto třídách na minimum.

4.4.2 Doporučená architektura

Tvůrci Androidu z výše popsaných důvodů doporučují rozdělit architekturu na alespoň dvě vrstvy [29]:

- UI vrstvu (též prezentační) – obstarává zobrazení dat aplikace na obrazovku,
- datovou vrstvu – obsahuje businessovou logiku a vystavuje data zbylým vrstvám.

Mezi oběma vrstvami se může nacházet ještě třetí, doménová vrstva. Ta zapouzdřuje složitou businessovou logiku, kterou je třeba použít na více místech v aplikaci. Každá třída doménové vrstvy optimálně představuje unikátní use case, případ užití, a je zodpovědná právě za jedinou věc. Vhodným poskládáním doménových tříd jsme schopni psát lépe udržitelné a testovatelné aplikace. Na druhou stranu, pokud nepotřebujeme sdílet složitou businessovou logiku mezi třídami, není potřeba tuto vrstvu implementovat.

4.4.2.1 UI vrstva

Prezentační neboli UI vrstva má na starost zobrazení dat. Jestliže se v aplikaci změní jakákoliv data, musí na ně umět zareagovat a obnovit uživatelské rozhraní. Pokud například uživatel klikne na tlačítko nebo aplikace obdrží aktualizovaná data z backendu, je UI vrstva povinna zajistit správné vykreslení nových změn. Prezentační vrstva se dále skládá ze dvou druhů komponent [30]:

- UI prvků, které vykreslují data na obrazovku. Donedávna to byly jenom instance třídy *View*, dnes to mohou být i funkce v Jetpack Compose,
- tříd reprezentující stav (tzv. *state holders*), například instance třídy *ViewModel*. Takové třídy uchovávají data, vystavují je UI a řídí logiku jiných částí.

View – objekty třídy *View* se nazývají widgety a v praxi jimi jsou například tlačítka, obrázky, textová pole a podobně. Pokud pominu Jetpack Compose, který mění způsob implementování aplikací od základu, pak běžným způsobem tvorby uživatelského rozhraní je nadefinování layoutů[29] pomocí XML[30].

[28] Operační systém

[29] Layout si můžeme představit jako HTML stránku. Místo HTML ale používáme v Androidu XML. Layout tedy definuje rozvržení UI.

[30] Z anglického Extensible Markup Language, jedná se o strojově zpracovatelný a pro člověka čitelný značkovací jazyk.

ViewModel je speciálně navržena třída za účelem uchování a správy UI-orientovaných dat. Řídí se životním cyklem aktivity nebo fragmentu. [31] Zjednodušeně řečeno, instance *ViewModelu* nám zajistí, že v případě konfiguračních změn (například otočení obrazovky nebo změna jazyka v zařízení) zůstanou aktuální UI data uložena. *ViewModel* na rozdíl od aktivit a fragmentů přežije konfigurační změny, kdeže aktivity a fragmenty jsou zničeny a opětovně vytvořeny, čímž přijdou o vložené informace. *ViewModel* je zároveň vázán na jejich životní cyklus, a tak když systém vyhodnotí, že by aktivita nebo fragment měly být definitivně odstraněny, pak odstraní i *ViewModel*.

Mimo uchování dat slouží *ViewModel* jako prostředník mezi komunikací aktivit či fragmentů s komponentami, jež nezávisí na uživatelském rozhraní. V praxi tedy *ViewModel* nejčastěji přijme požadavek od fragmentu, ten zpracuje, a přerozdělí úkoly jiným komponentám. Fragment mezitím čeká na odpověď sledováním instance třídy *LiveData*[31]. Jakmile *ViewModel* dokončí požadavek, aktualizuje *LiveData* a fragment obdrží nová data.

4.4.2.2 Datová vrstva

Datová vrstva obsahuje businessovou logiku vyvíjené aplikace. Jejím účelem je spravovat data, která dohromady tvoří celou aplikaci. Datová vrstva se tak snaží vyhodnotit, jaká data a jakým způsobem by měla být vytvořena, které informace uložit a jak se zachovat během jejich změn. Datová vrstva se skládá z tzv. repozitářů (anglicky *repository*, dále budu v práci používat tento anglický pojem).

Repository je třída, která zapouzdřuje přístup k libovolnému počtu zdrojů dat (tzv. *data sources*). Jak již název napovídá, každý zdroj nabízí *repository* data. Liší se ale v tom, odkud data čerpají. Zdrojem může být například lokální databáze (např. pomocí knihovny Room), *sharedPreferences*, Jetpack DataStore[32]nebo spojení s webovou službou (např. knihovnou Retrofit). Z hlediska oddělení zodpovědností je důležité, aby jedna instance zdroje dat pracovala pouze s jediným skutečným zdrojem. *Repository* se následně snaží rozhodnout, z jakého zdroje a ve které situaci data získat. Přesněji je *repository* odpovědné za [32]:

- vystavení dat zbytku aplikace,
- centralizaci změn dat,
- řešení konfliktů mezi více zdroji dat,
- abstrahování zdrojů dat od zbytku aplikace,

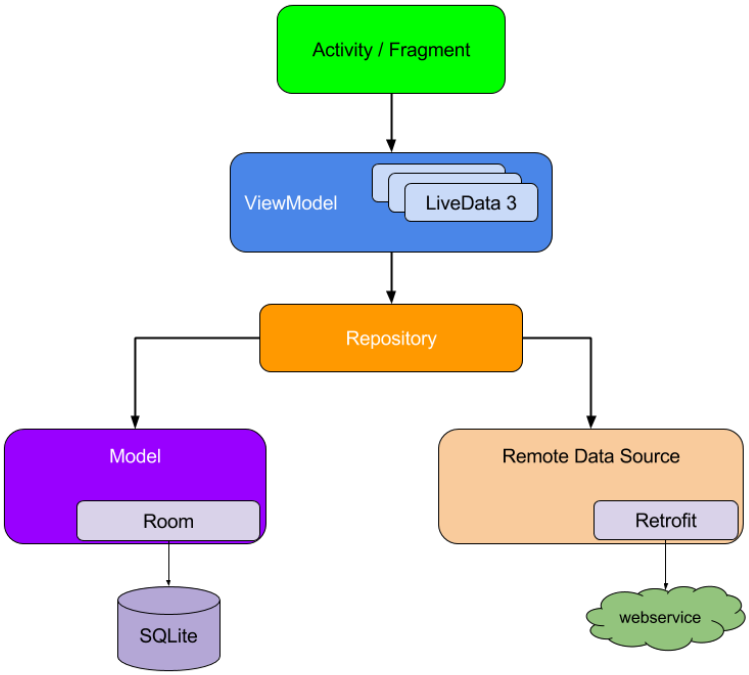
[31] LiveData implementují návrhový vzor Observer, jenž umožňuje sledovat změny hodnot.

[32] Jednoduché úložiště pro Android aplikace umožňující uchovávat data typu klíč-hodnota. Jetpack DataStore je modernizovaný nástupce *sharedPreferences*.

- businessovou logiku aplikace.

4.4.2.3 Použití v aplikaci

Při návrhu architektury jsem vycházel z doporučeného způsobu rozvržení zodpovědností. Každý fragment tak v mnou vyvíjené aplikaci spolupracuje s vlastním *ViewModelem*, který pak dále komunikuje s *repository*. Výsledky ale *ViewModel* většinou nevrací fragmentu, nýbrž přímo UI komponentám díky knihovně Data Binding z balíčku Android Jetpack.



Obrázek 4.11: Ukázka doporučené architektury Android aplikace [33]

4.5 Data Binding

Jak jsem v uvedl v odstavci výše, Data Binding je knihovna pocházející z balíčku Android Jetpack. Od roku 2018 pomáhá Android Jetpack vývojářům s dodržováním osvědčených postupů při vývoji aplikací (tzv. *best practices*). Balíček se skládá ze značného počtu oficiálních knihoven, které podle autorů redukují množství napsaného kódu. Zároveň balíček pomáhá s programováním takového kódu, který funguje konzistentně na různých verzích Androidu a mobilních zařízení. [34] Použití knihoven z tohoto balíčku také v praxi snižuje chybovost aplikací. Společnost Monzo, vyvíjející bankovní aplikaci, dodává:

„V naší aplikaci jsme původně pro fotografování dokumentů a nahrávání selfie používali *camera2* API. Bohužel se stávalo, že až na 25 % zařízení nám aplikace padala. Jakmile vyšla knihovna CameraX z Android Jetpack, rozhodli jsme se naše tehdejší řešení upravit. Díky úspěšné integraci knihovny jsme pak mohli z projektu smazat 9000 řádků kódů a výsledkem je, že aplikace nyní padá uživatelům mnohem méně, jen v pěti procentech případů.“ [35]

Data Binding umožňuje propojit uživatelské rozhraní s třídami zapouzdřující UI data, nejčastěji *ViewModely*. Widgety pomocí speciální syntaxe pozorují data přiřazené třídy a pokud dojde k změně, aktualizují svůj stav (a tím vlastně aktualizují i UI). Pro snazší představu uvádím názornou ukázku převzatou z oficiální dokumentace (viz 4.1, 4.2, 4.3). [36]

V aktivitách nebo fragmentech často definujeme widgety z layoutu. Kód níže (viz 4.1) volá metodu `findViewById`, která podle ID dokáže nalézt konkrétní widget, zde `TextView`. Widget `TextView` představuje v našem layoutu textové pole. V ukázce 4.1 pak `TextView` nastavíme text, jenž je obsažen v proměnné `userName`. Jak můžete vidět, tato proměnná je součástí *ViewModelu*.

```
findViewById<TextView>(R.id.sample_text).apply {
text = viewModel.userName
}
```

Ukázka kódu 4.1: Nastavení textu *TextView* bez knihovny Data Binding

Pokud bychom chtěli docílit stejného efektu jenom s použitím Data Bindingu, pak bude `TextView` v našem XML layoutu vypadat následovně:

```
<TextView
    android:text="@{viewmodel.userName}" />
```

Ukázka kódu 4.2: Nastavení textu *TextView* s knihovnou Data Binding

Všimněte si, že se celá logika skrývá pouze za syntaxí `@{}`.

Do layoutu poté umístíme párový tag `<data>`, kterým propojíme *ViewModel* s layoutem. Knihovna Data Binding pak na pozadí vygeneruje třídu, jež zajistí funkčnost tohoto propojení:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto">
    <data>
        <variable
            name="viewmodel"
            type="com.myapp.data.ViewModel" />
    </data>
    <ConstraintLayout... />
</layout>
```

Ukázka kódu 4.3: Propojení *ViewModelu* s XML layoutem

Do zdrojového kódu fragmentu nebo aktivity už tedy nic psát nemusíme, nicméně stále z nich můžeme k widgetům přistupovat pomocí zmíněné vygenerované třídy. Data Binding knihovna ovšem nabízí další užitečné funkce, z nichž dvě popisuji na dalších řádcích.

4.5.1 Binding adapter

Pro každý výraz v layoutu existuje tzv. *binding adapter*, který přiřadí výrazu skutečnou metodu třídy *View* (nebo jeho potomka). Jak jsem již uvedl dříve, widget je ve skutečnosti instance třídy *View*. To znamená, že například u widgetu `TextView` a jeho výrazu `android:text` zavolá *binding adapter* metodu `setText()`. Co když ale máme nadefinovanou vlastní třídu dědící z *View*? Nebo potřebujeme rozšířit widget o nové funkce? Pro tyto případy vytvořili autoři knihovny způsob, jak si nadefinovat vlastní adaptér:

```
@BindingAdapter("visibleOrGone")
fun View.visibleOrGone(visible: Boolean) {
   visibility = if (visible) View.VISIBLE else View.GONE
}
```

Ukázka kódu 4.4: Vytvoření vlastního binding adaptéru

Nyní můžeme v XML layoutu použít výraz „visibleOrGone“ (nebo „app:visibleOrGone“) stejně jako zmíněný `android:text` a knihovna správně zavolá námi implementovanou metodu `visibleOrGone()`.

Z mé osobní zkušenosti můžu říct, že je tento způsob rozšíření widgetů o nové metody velmi užitečný a během vývoje aplikace jsem jej použil mnohokrát.

4.5.2 Two-way data binding

Poslední funkcí knihovny, kterou zde zmíním, je tzv. *two-way binding*. *Two-way binding* umožňuje widgetům nejenom pozorovat změny, ale i aktualizovat obsah proměnné, kterou pozorují. Tím máme bez jakýkoliv změn v kódu třídy zaručeno, že proměnná obsahuje aktuální UI data. Užití této funkcionality je navíc poměrně jednoduché. K speciální syntaxi `@{}` totiž stačí přidat pouze rovnítko: `@={}`. Na druhou stranu, tuto funkci podporují implicitně jenom některé předem definované binding adaptéry. Zbytek adaptérů si pak musí vývojář naprogramovat sám.

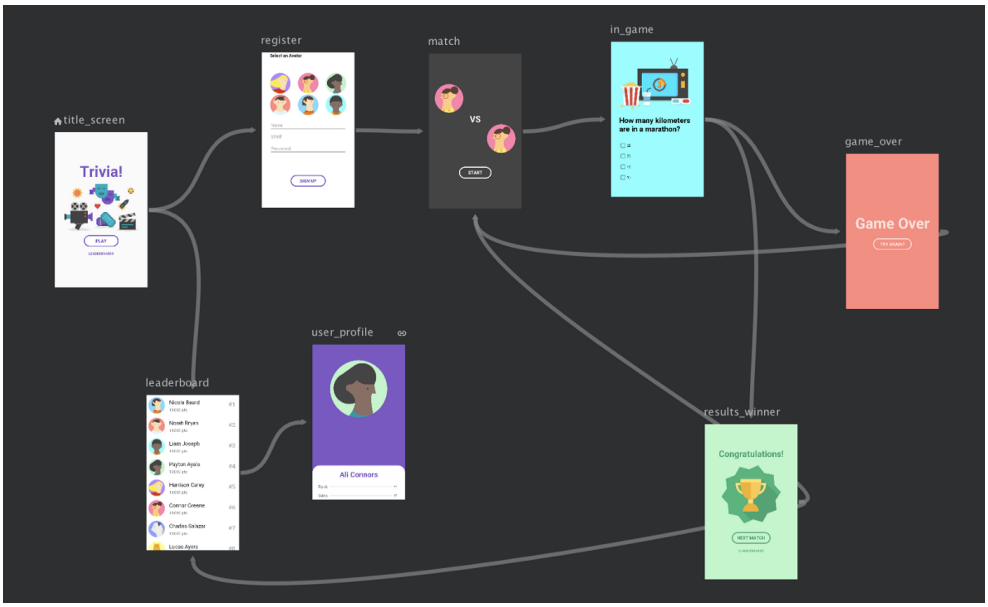
4.5.3 Použití v aplikaci

Použitím knihovny Data Binding jsme schopni z aktivit a fragmentů odstranit přímou závislost na UI, čímž aktivity a fragmenty zjednodušujeme a jejich kód se lépe udržuje. Data Binding navíc vylepšuje výkon celé aplikace a

pomáhá s prevencí úniků paměti[33] (anglicky memory leak) a pádů aplikace (nejčastěji kvůli *NullPointerException*). [36] Data Binding knihovna proto tvoří významnou součást zdrojového kódu mojí aplikace a díky ní komunikuje většina instancí *ViewModelu* s UI přímo, bez užití fragmentu. Zpětně také můžu dodat, že mi práce s knihovnou urychlila vývoj a snížila množství tzv. *boilerplate*[34] kódu, jenž jsem musel napsat.

4.6 Navigation

Možná jste si všimli, že v této práci uvádím častěji fragmenty – základní jednotky Android aplikací, nežli aktivity. Opravdu tomu není náhoda, neboť má aplikace obsahuje ve skutečnosti pouze jednu aktivitu. Zbylé třídy poskytující UI jsou pak v mém kódu právě fragmenty.



Obrázek 4.12: Ukázka navigačního grafu v Android Studiu [37]

Aplikace, která přijala tento architektonický styl, se nazývá *single-activity app*. Ve stručnosti, *single-activity* aplikace je tvořena nejlépe jenom jednou aktivitou. Aktivitu pak zde můžeme chápat jako jakýsi kontejner pro fragmenty. To jinými slovy znamená, že systém umístí vždy aktuálně zobrazovaný fragment do těla aktivity. Možná vás teď napadne, jaký to má ale praktický význam?

[33] Únik paměti nastává tehdy, když systém není schopen uvolnit programu paměť, kterou dříve používal a nyní ji už nepotřebuje.

[34] Kód, který je nutný napsat, abychom zprovoznili část programu. Vetšinou jej nejde příliš zkrátit ani optimalizovat, takže se často opakuje.

Ačkoliv jsou aktivity součástí Androidu již od jeho počátku, ve sdílení dat s dalšími aktivitami a v přesunech z jedné aktivity do druhé nejsou příliš flexibilní. Pokud naopak pracujeme s fragmenty, které jsou součástí společné aktivity, mohou tyto fragmenty sdílet data pomocí sdíleného *ViewModelu* poměrně jednoduše. Na druhou stranu, vlastní implementace navigování mezi fragmenty, neboli zjednodušeně řečeno procházení mezi jednotlivými obrazovkami, je poměrně komplexní záležitost a vede k mnohým chybám. Naštěstí Google vyslechl stížnosti vývojářů a v roce 2018 představil Navigation komponentu, jež napravuje výše zmíněné úskalí. [38]

Navigation komponenta z balíčku Android Jetpack zjednodušuje implementaci navigování v aplikaci a k tomu nabízí množství pokročilých funkcí. Pro vývojáře je příjemné, že veškerou složitou práci s *FragmentManagerem* a transakcemi řeší knihovna automaticky. Navíc dokáže úzce spolupracovat s dalšími UI komponentami pro navigaci – například *bottom navigation bar*, *toolbar* nebo *navigation drawer*[35]. Jak se lze dočíst v dokumentaci: „Komponenta zaručuje uživatelům konzistentní a předpokládané chování během užívání aplikací díky dodržování sady doporučených postupů.“ Abychom ale mohli Navigation komponentu použít ve vyvíjené aplikaci, je třeba porozumět třem základním pojmům [39]:

- **Navigation graph** (navigační graf) je XML dokument, který obsahuje veškeré potřebné informace pro zajištění navigace. Mimo jiné zahrnuje popis všech obrazovek, jež může uživatel v aplikaci navštívit. V kontextu Navigation komponenty jsou tyto obrazovky pojmenovány jako destinace – *destinations*. V neposlední řadě definujeme v navigačním grafu spojení mezi destinacemi. Skvělé je, že vývojové prostředí Android Studio dokáže nadefinovaná spojení a destinace vizualizovat (viz 4.12).

- **NavHost** je kontejner, který je schopen zobrazit destinace z navigačního grafu. Pro snazší práci můžeme použít speciální kontejner *NavHostFragment*, v rámci kterého jsou vyměňovány jednotlivé fragmenty, mezi nimiž se uživatel pohybuje. [40]

- **NavController** je třída, jež zajišťuje správné fungování navigace. Můžeme říci, že její činností je výměna fragmentů v kontejneru *NavHost* podle toho, jak se uživatel přesouvá v aplikaci mezi destinacemi.

Jak jsem již zmínil, v mém řešení naleznete z výše popsaných důvodů pouze jednu aktivitu. O navigaci se tak stará samotná komponenta. Výsledný kód je potom jednodušší, udržitelnější a nabídne méně prostoru pro chyby. Komponenta navíc obsahuje další skvělé funkce, jako je například zaručení datových typů během přeposílání argumentů mezi destinacemi. Nadto plánuji

[35]Rozbalující boční menu, které umožňuje uživateli procházet mezi obrazovkami v aplikaci.

do budoucna přidat notifikace a možnost sdílet odkaz k cestě s dalšími lidmi. I k tomu mi pomůže Navigation komponenta, jež pro tyto účely ulehčuje práci s tzv. *deep linky*.

4.7 Dependency injection

Jak jsem uvedl v podkapitole 4.4.2.3, mnou vyvíjená aplikace se skládá z mnoha fragmentů, kde každý fragment komunikuje se svým *ViewModelem*. A co víc, *ViewModel* deleguje práci jiným částem aplikace, zejména libovolnému počtu instancí *repository* (v závislosti na konkrétním *ViewModelu*), se kterými úzce spolupracuje. Každé *repository* se poté musí rozhodnout, z jakého zdroje data získat. I malá aplikace se tak rázem stává docela složitá, pokud se bavíme o zajištění závislostí mezi třídami. Právě pro tyto účely slouží návrhový vzor Dependency injection (česky Vkládání závislostí, známý také pod zkratkou DI), který se snaží centralizovaně vyřešit závislosti mezi jednotlivými komponentami. Jak takové řešení obecně vypadá?

DI framework zpravidla nabízí speciální metody nebo anotace, kterými jsme schopni vytvořit instance vyvíjených tříd. Konkrétní vytváření instancí se většinou příliš neliší od klasického zavolání konstruktoru, jak je obvyklé ve většině OOP jazycích. Důležité ve smyslu DI jsou ale právě ty speciální metody či anotace, jež DI framework nabízí. Pomocí nich totiž framework rozpozná, jak námi nadefinovanou instanci zkonstruovat a dále s ní pracovat. Síla DI frameworku tkví právě v tom, že je framework schopen dodat instance na patřičná místa v kódu automaticky.

Pro jistotu uvedu zjednodušený příklad přímo z mého zdrojového kódu. Ukázka je naimplementována pomocí frameworku Koin.
Mějme *ViewModel* zvaný `HomeFragmentVM`:

```
class HomeFragmentVM(
   private val authRepository: AuthRepository,
   private val tripsRepository: TripsRepository,
   private val itineraryRepository: ItineraryRepository,
   private val sessionManager: SessionManager
) : BaseViewModel()
```

Ukázka kódu 4.5: Příklad konstruktoru třídy *HomeFragmentVM*

`HomeFragmentVM` závisí na třech *repository* a jedné instanci `SessionManageru`. Uvažujme první *repository*.

```
class AuthRepository(
   private val sessionManager: SessionManager,
   private val authApi: AuthApi,
   private val accountApi: AccountApi
)
```

Ukázka kódu 4.6: Příklad konstruktoru třídy *AuthRepository*

Toto *repository* dále závisí na dalších třech komponentách. Jinými slovy tím můžeme říci, že abychom mohli vytvořit instanci `AuthRepository`, musíme být schopni prvně vytvořit instance `SessionManageru`, `AuthApi` a `AccountApi`. Začněme od konce:

```
module { //misto pro definovani zavislosti
   single { provideApi<AuthApi>(get()) } //vytvori instanci AuthApi
   single { provideApi<AccountApi>(get()) } //vytvori instanci
       AccountApi
}
```

Ukázka kódu 4.7: Definování *singletonu* v Koinu

Funkce `provideApi()` vytvoří instanci tak, jak bychom ji vytvořili kdekoliv jinde v kódu bez *dependency injection*. Důležitá je ale metoda `single()`. Koin díky ní pochopí, jakým způsobem dosadit tuto instanci tam, kam potřebujeme. Dosazení (neboli anglicky inject) do kódu je pak jednoduché, například výraz:

```
val authApi by inject<AuthApi>() //inject je metoda z Koinu
```

Ukázka kódu 4.8: Vytvoření instance pomocí Koinu

zkonstruuje instanci `AuthApi`.

Jakmile framework ví, jak vytvořit `AccountApi`, `AuthApi` a `SessionManager` (principiálně stejně), už mu nechybí žádná informace k tomu, aby nedokázal zkonstruovat i instanci `AuthRepository`. V Koinu k takovému účelu slouží elegantní metoda `singleOf()`:

```
module { //misto pro definovani zavislosti
   singleOf(::AuthRepository) //vytvori instanci AuthRepository
}
```

Ukázka kódu 4.9: Definování *singletonu* pomocí Koin Extended DSL

Obdobně jsme schopni zformovat i zbylé komponenty, na nichž závisí `HomeFragmentVM` a na konec i samotný *ViewModel*:

```
module {
   viewModelOf(::HomeFragmentVM)
}
```

Ukázka kódu 4.10: Definování *ViewModelu* pomocí Koin Extended DSL

4.8 Propojení s backendem

Už od začátku jsem koncipoval mobilní aplikaci a backend tak, aby veškerou businessovou logiku měl na starost backend a mobilní aplikace byla tak „hlou-

pá“, jak jen to jde. Proto je připojení mobilní aplikace k mé webové službě klíčové.

V prostředí Androidu slouží ke komunikaci mobilní aplikace se vzdáleným API například populární knihovna Retrofit. Právě tu jsem použil i já. S pomocí Retrofitu vytváříme HTTP požadavky jako metody rozhraní (interface). Na základě námi dodaných anotací pak knihovna naimplementuje naše metody. Pokud se podíváme metodám pod pokličku, pak zjistíme, že ve skutečnosti volá Retrofit metody knihovny OkHttp – efektivního HTTP klienta od společnosti Square[36].

```
@POST("/trip/{id}/place")
suspend fun createPlace(
   @Path("id") id: Int,
   @Body request: PlaceRequest,
   @Query("wiki") wikiSearch: String?
): Response<CreatedResponse>
```

Ukázka kódu 4.11: Ukázka vytvoření POST požadavku v knihovně Retrofit. Klíčové slovo *suspend* značí, že implementace metody používá na pozadí korutiny (coroutines)

4.8.1 Přidání autorizační hlavičky

Většina zdrojů mé webové služby je zabezpečená, čili vyžaduje autorizační hlavičku. Abych nemusel psát hlavičku ručně u všech požadavků zvlášť, nabízí Retrofit vytvoření tzv. *interceptoru* (pro tento výraz nemáme žádný český ekvivalent). Třída implementující rozhraní `Interceptor` dokáže pozorovat a upravovat HTTP požadavky a jejich odpovědi.

V mém kódu jsem tak naprogramoval třídu `AuthInterceptor` implementující zmíněné rozhraní, jež nalezne v paměti aplikace uložený access token a přidá jej do hlavičky. Retrofit poté provede požadavek s již přiloženým tokenem.

```
override fun intercept(chain: Interceptor.Chain): Response {
   val authRequest = chain.request().newBuilder().apply { //uprava
       odchoziho pozadavku
       sessionManager.getAccessToken()?.let { //nalezeni access tokenu
           header(AUTH_HEADER, bearerToken(it)) //pridani hlavicky s
               tokenem
       }
   }.build()
   return chain.proceed(authRequest) //aktualizujeme pozadavek
}
```

Ukázka kódu 4.12: Ukázka metody *intercept* pro přidání hlavičky do požadavku

[36]Společnost Square je autorem knihovny Retrofit.

4.8.2 Zajištění autorizace

V podkapitole 3.5 jsem se rozepsal o způsobu zajištění autorizace mé webové služby. Mobilní aplikace je povinna zajistit, aby autorizace správně fungovala i z klientského pohledu. To znamená, že k sobě musí uložit access i refresh token, zvládnout odesílat access token spolu s požadavkem a v případě odmítnutí aktualizovat oba dva tokeny a provést požadavek znovu.

Aplikace po úspěšném přihlášení uloží k sobě access i refresh token. K uchování obou tokenů mi posloužilo uložiště *sharedPreferences*, ke kterému nemají přístup ostatní aplikace. Protože má refresh token nastavenou dlouhou platnost, uložil jsem jej z bezpečnostních důvodů v zašifrované podobě. K tomuto činu mi posloužila speciální implementace *EncryptedSharedPreferences*, jež je součástí balíčku AndroidX Security a byla představena teprve nedávno.

Odeslání access tokenu jsem popsal v podkapitole výše, a tak nyní již zbývá vyřešit „jen“ rotaci tokenů a znovuzavolání posledního požadavku.

Obnovování tokenů není ale popravdě natolik těžké, jak by se na první pohled mohlo zdát. Retrofit totiž nabízí rozhraní `Authenticator`, které v případě HTTP odpovědi *401 – Unauthorized* zavolá námi implementovanou metodu. Pokud je metoda zdařilá, Retrofit opětovně provede poslední neúspěšný požadavek.

Mnou implementovaná metoda tak potají zavolá požadavek na obnovení access tokenu. Pokud webová služba vydá nový access a refresh token, metoda oba dva tokeny uloží (refresh token v zašifrované podobě) a aktualizuje autorizační hlavičku původního neúspěšného požadavku. V případě neúspěchu, tedy když webová služba odmítla z bezpečnostních důvodů vydat nové tokeny, implementovaná metoda pošle signál zbytku aplikace k odhlášení uživatele. Co si ale pod pojmem „odešle signál“ představit?

Signál je poslán tak, že metoda uloží do Jetpack DataStore výzvu k odhlášení. Jetpack DataStore je modernizovaná verze *sharedPreferences*, která stejně jako *sharedPreferences* ukládá data ve formátu klíč-hodnota. Navíc ale dovoluje asynchronně sledovat změny konkrétních hodnot podle klíče – užitím Flow, relativně mladé knihovny programovacího jazyka Kotlin. Flow přichází na scénu tam, kde potřebujeme asynchronně odebírat hodnoty, jež produkuje nějaká *suspend* metoda[37].

Zpět již ale k mému programu. Uloženou výzvu k odhlášení zaregistruje má jediná aktivita v kódu (viz podkapitola 4.6), která pozoruje změny v úložišti. Po obdržení výzvy smaže z paměti zařízení tokeny a přesměruje uživatele na přihlašovací obrazovku.

[37]Suspend metoda je jádrem dění korutin v Kotlinu. Slovo *suspend* značí, že metoda může být během provádění instrukcí na nějaký čas pozastavena.

4.9 Ukázková implementace jednoho požadavku

Pro ukázku implementace mnou vyvíjené mobilní aplikace poslouží případ, kdy si uživatel zapomene heslo. Samotný fragment a jeho *ViewModel* totiž mají dohromady 60 řádků kódu, přesto nejsou tyto třídy o nic ošizené a obdobně funguje víceméně i zbytek aplikace.

```
class ForgotPasswordFragment :
    MvvmFragment<FragmentForgotPasswordBinding,
    ForgotPasswordFragmentVM>(
   R.layout.fragment_forgot_password,
   ForgotPasswordFragmentVM::class
) {
   override val hasBottomNavigation = false
   override val hasTitle = false
}
```

Ukázka kódu 4.13: Implementace fragmentu *ForgotPasswordFragment* má přesně 9 řádků

V podkapitole 4.5.3 jsem se zmínil, že v mém zdrojovém kódu komunikuje většina *ViewModelů* s UI přímo, bez použití fragmentu. Ukázka 4.13 výše je toho názorným příkladem. Fragment je zde opravdu jen zástupce jedné destinace reprezentován layoutem `fragment_forgot_password`. O správné propojení vygenerované třídy z knihovny Data Binding s *ViewModelem* se v příkladu stará abstraktní třída `MvvmFragment`. Samotná logika zapomenutí hesla se pak ukrývá ve spolupracujícím *ViewModelu* – konkrétně `ForgotPasswordFragmentVM`.

Obrazovka se zapomenutím hesla obsahuje pouze jedno textové políčko, kam uživatel musí zadat svůj email. Před zasláním požadavku na webovou službu tak musí *ViewModel* zkontrolovat, zdali vložený text je validní emailová adresa:

```
val email = MutableLiveData<String>()

val validateEmail = email.switchMapSuspend {
   validator.validateEmail(it)
}
```

Ukázka kódu 4.14: Ověření platnosti zadaného emailu

Proměnná `email` je propojena díky Data Binding knihovně přímo s textovým políčkem v layoutu. Propojení je navíc obousměrné (neboli *two-way binding*, viz podkapitola 4.5.2), a tak se v proměnné nachází vždy aktuální hodnota, kterou uživatel zadal. Metoda `switchMapSuspend()` je pak mé rozšíření metody `switchMap()` z LiveData API, která pokaždé, když se změní hodnota v proměnné email, zavolá *suspend* metodu `validateEmail()` pro zkontrolování emailu. Protože ověření správnosti emailové adresy není na první pohled

tak jednoduché, jak by se mohlo zdát, přenechal jsem raději tuto záludnost externí knihovně.

Pro zlepšení UX[38]nebude uživatel moci kliknout na tlačítko „odeslat“ dříve, než dokud správně nevyplní email. O to se stará třída `ValidationMediator`:

```
val enableSubmit = ValidationMediator(validateEmail)
```

Ukázka kódu 4.15: Příklad použití třídy *ValidationMediator*

Tato třída umí přijmout libovolný počet argumentů typu *LiveData* (v ukázce přijímá pouze jeden argument). Pro připomínku, třída *LiveData* umožňuje sledovat změny hodnot implementací návrhového vzoru *Observer*. Třída `ValidationMediator` tak obsahuje seznam parametrů typu *LiveData* a pokaždé, když se změní hodnota některého z parametrů, vyhodnotí, zdali již bude uživateli dovoleno kliknout na tlačítko. Jinými slovy zkontroluje, že uživatel vyplnil vše správně.

Hodnotu v proměnné `enableSubmit` pak sleduje tlačítko přímo v XML layoutu:

```
//Pro ukazku zkraceno, dulezity pro nas je ted vyraz app:isEnabled.
//app:isEnabled je vlastni nadefinovany binding adapter
<com.google.android.material.button.MaterialButton
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:onClick="@{() -> vm.forgotPassword()}"
   android:text="@string/submit"
   app:isEnabled="@{vm.enableSubmit}" />
```

Ukázka kódu 4.16: Ukázka spolupráce widgetu a *ViewModelu* v XML layoutu

Po kliknutí na tlačítko (viz výraz `android:onClick`) se zavolá ve *ViewModelu* metoda `forgotPassword()`. Ta má na starost odeslání požadavku webové službě a uživateli zobrazit jeho stav.

```
fun forgotPassword() {
   viewModelScope.launch {
       val email = email.value ?: return@launchIO
       val request = ForgotPasswordRequest(email)
       authRepository.forgotPassword(request).safeCollect(this) {
           when (it) {
               is Result.Cache -> TODO()
               is Result.Failure -> forgotPasswordFailure()
               is Result.Loading -> loading.value = true
               is Result.Success -> forgotPasswordSuccess()
           }
}}}
```

Ukázka kódu 4.17: Zpracování požadavku ve *ViewModelu*

[38]Z anglického User Experience, jedná se o uživatelský zážitek z používání programu či aplikace.

Konkrétně tedy připraví požadavek – `ForgotPasswordRequest(email)` – a zavolá vhodnou metodu z *repository* – `forgotPassword(request)`. Dále pak již jen sleduje, v jaké fázi zpracování se požadavek nachází:

- Cache – požadavek odeslán ke zpracování, momentálně jsou dostupná data z keše,
- Failure – požadavek skončil chybou, návratová hodnota obsahuje chybovou hlášku,
- Loading – začátek zpracování požadavku, zobraz načítání,
- Success – požadavek skončil úspěchem, návratová hodnota obsahuje odpověď REST API.

Je nutné zmínit, že metoda z *repository* `forgotPassword(request)` je *suspend* metoda, a tak může být zavolána pouze z další *suspend* metody nebo být součástí korutiny. Jelikož původní metoda `forgotPassword()` není *suspend*, je volána právě z korutiny pomocí funkce `launch()`.

Věnujte prosím v ukázce 4.17 pozornost `viewModelScope.launch {...}`. Každá spuštěná korutina závisí na rozhraní *CoroutineScope*. Toto rozhraní zjednodušeně řečeno definuje životní cyklus korutiny a zodpovídá za její zrušení. Pro snazší práci s korutinami ve *ViewModelu* tak autoři Android SDK vytvořili extension proměnnou *ViewModelScope*. Ta nedělá nic jiného, než že vrací „přizpůsobenou" instanci *CoroutineScope*, která dokáže během odstranění *ViewModelu* z paměti zrušit všechny probíhající korutiny[39]. Funkce `launch()` je pak extension funkce na *CoroutineScope* a spouští novou korutinu.

Mechanismus sledování jednotlivých hodnot z *repository*, které přicházejí postupně, jsem naimplementoval právě díky Kotlin Flow (viz podkapitola 4.8.2). Uživatel tak v každý moment ví, co se děje na pozadí aplikace (načítání, doručení z keše, chyba, apod.). Má aplikace zatím kešování nepodporuje, nicméně v budoucích verzích je s tím počítáno.

Metoda `forgotPassword(request)` z *repository* pak vypadá následovně:

```
suspend fun forgotPassword(forgotPasswordRequest:
    ForgotPasswordRequest): Flow<Result<Unit>> {
   return performCall {
      authApi.forgotPassword(forgotPasswordRequest)
   }
}
```

Ukázka kódu 4.18: *Repository* volá kvůli zapomenutí hesla požadavek na vzdálené API

[39]Samotné téma korutin je v praxi mnohem složitější, nicméně podrobnější vysvětlení fungování korutin je již mimo rozsah této práce.

Mozkem celé metody 4.18 je lambda funkce `performCall()`, která odešle požadavek na vzdálené API a zároveň informuje o aktuálním stavu zpracování. Pod pokličkou ukrývá volání další lambda funkce `safeApiCall()`, jež provede samotné spojení s webovou službou a odpověď převede na srozumitelný objekt (v kontextu OOP) značící buď úspěch, nebo neúspěch.

`ForgotPasswordFragmentVM` pak v případě neúspěchu zobrazí chybovou hlášku. Nicméně pokud vše dopadlo dobře, vrátí uživatele (díky Navigation Component, viz podkapitola 4.6) na přihlašovací obrazovku a zobrazí Snackbar[40]. Ten informuje uživatele, že by měl zkontrolovat svojí emailovou schránku, a celý proces je tímto dokončen.

[40]Dialog s krátkou zprávou, který se (na chvíli) objeví na spodku obrazovky.

Kapitola 5

Uživatelské testování

V průběhu vývoje jsem si testoval mojí aplikaci sám a lokálně. Klíčovou funkcí aplikace je ovšem organizace společné cesty ve skupině. Abych ověřil, že jsem mé řešení naimplementoval správně a uživatelsky přívětivě, uspořádal jsem uživatelské testování. V této kapitole tak píšu o přípravě testovacích scénářů, samotném průběhu testování a vyhodnocení.

5.1 Testovací scénář

Mým cílem bylo otestovat reálné plánování společné cesty ve skupině (vesměs) přátel. V průběhu několika dnů se mi k testování přihlásilo 12 účastníků. Ty jsem rozdělil do tří testovacích skupin po čtyřech členech. Každá skupina dostala stejné zadání, nicméně všichni členové jedné skupiny obdrželi jiné úkoly. Ty na sebe ale částečným způsobem navazovaly.

V aplikaci mají všichni uživatelé během plánování cesty přiřazenou jednu roli. Vlastník cesty je automaticky administrátorem (roli si ale může později změnit), zbytek členů pak může býti editory, diváky i adminy. Z toho plyne fakt, že každý cestovatel má v konkrétní cestě jiné možnosti a pravomoci.

Abych pokryl všechny možné případy, rozvrhl jsem účastníkům testování ve skupině přesně tyto role – 1x admin, 2x editor, 1x divák.

Účastníky jsem dopředu seznámil s informací, že mobilní aplikace neumožňuje konverzaci mezi uživateli, a tak slouží primárně jako doplněk pro kolektivní plánování cest k jiným komunikačním aplikacím (např. Messengeru nebo WhatsAppu). Pro každou testovací skupinu jsem proto vytvořil společnou místnost v chatovací aplikaci, kde se mohli účastníci spolu domlouvat. Na vypracování úkolů pak dostali účastníci jeden týden.

Před bližším představením scénářů v tomto textu musím ovšem uvést jednu poznámku. Testovací scénáře obsahují v této práci místa se třemi hvězdičkami (***). Ve skutečných úkolech byl na těchto místech uveden další ze členů skupiny. Kupříkladu typickým úkolem bylo takto přidat do výpravy nového člena. Během testování se ovšem ukázalo, že formulace tohoto typu úkolu

nebyla příliš šťastná. Někteří účastníci mi totiž později psali, že nemohou například Honzu v aplikaci najít. Ve skutečnosti měli ale od Honzy získat uživatelské jméno a až poté jej pod jeho uživatelským jménem přidat.

Nutno taky podotknout, že mobilní aplikace obsahuje obrazovku s výdaji. Pravdou ovšem je, že jsem evidenci výdajů nestihl v mobilní aplikaci doimplementovat[41], a tak se v rámci testování na tuto funkcionalitu vůbec neptám.

5.1.1 Scénář pro administrátora

1. Zaregistruj se a přihlas se.
2. Plánuješ cestu za polární kruh. Cesta se uskuteční někdy během letošních letních prázdnin.
3. Přidej do této cesty *** jako editora.
4. Vyjedete autem z Prahy-Zličín v dopoledních hodinách. Očekáváte, že kolem poledne dorazíte do Drážďan. Parkovat budete přímo na letišti.
5. Večer v 18:30 ještě ten samý den vám letí z Drážďan do Osla letadlo. Pravidelný přílet je v 19:50.
6. Pokud si všimneš, že ve výletu přibyl *Jan Steuer*, řekni ne ne ne, a odeber ho z výletu.
7. Ulož si aktivitu, kterou přidal někdo z editorů, do kalendáře (Google Calendar, Outlook apod.).
8. Vzpomněl/a sis, že nikdo neví, jaké máš auto, a tak doplň cestu autem z Prahy do Drážďan o popis tvého auta.

5.1.2 Scénář pro prvního editora

1. Zaregistruj se a přihlas se.
2. Navrhni adminovi, že ti datum odjezdu nevyhovuje. Jakmile se domluvíte na jiném termínu, změň datum odjezdu naplánované cesty.
3. Přidej do výletu *** jako editora.
4. V Oslu jsi našla/našel skvělé ubytování v místním hotelu Verdandi Hotel. Přijedete večer, ale zatím netušíte, kdy odjedete.
5. Nahraj potvrzení rezervace (jakýkoliv dokument) a nech jej přístupný pro všechny.

[41] Backend evidenci a přerozdělování výdajů ve skupině umí.

6. Navrhni aktivitu v národním parku Lomsdal-Visten v druhé polovině vaší cesty. Doplň webové stránky národního parku.

7. Omylem jsi nahrál/a špatný dokument. Smaž jej a nahraď novým.

5.1.3 Scénář pro druhého editora

1. Zaregistruj se a přihlas se.

2. Z Osla Central Station máte v plánu jet vlakem do Bergenu. Odjíždíte v 10:25. Sedíte ve vagonu 332, na místech 42, 43, 44, 45.

3. V okolí Bergenu navrhni místo, které bys chtěl/a navštívit.

4. Navrhni, že hotel Verdandi (až bude přidaný) je příliš drahý a že jsi našla/našel levnější ubytování přes Airbnb v Oslu. Smaž Hotel Verdandi a místo něj přidej tvůj nový objev.

5. Přidej do výletu uživatele jan.steuer jako diváka.

6. Nahraj k letu letenku (jakýkoliv dokument). Protože nechceš, aby byla dostupná i pro ostatní, zahesluj ji. Pojmenuj ji jako *moje_letenka* a heslo vzkaž ***.

7. Ulož si cestu do kalendáře (Google Calendar, Outlook apod.).

5.1.4 Scénář pro diváka

1. Zaregistruj se a přihlas se.

2. Rád/a by ses taky nějak zapojil/a, a tak tě napadne, že bys mohl/a mezitím nastudovat trasu z Prahy do Drážďan, abys mohl/a později navigovat řidiče.

3. Jakmile ti někdo ze skupiny sdělí heslo a název dokumentu, stáhni si jej k sobě do telefonu.

4. Protože tě baví mapy, zobraz si mapu s již přidanými body vaší cesty za polární kruh.

5. Změň si jméno v aplikaci.

6. V průběhu testování tě uživatel *jan.steuer* přidal do již probíhající cesty. Zobraz aktuálně probíhající bod zájmu (neboli místo, ubytování, dopravu nebo aktivitu).

7. V právě probíhající cestě nalezni dnešní den.

8. Nakonec se cesty za polární kruh nezúčastníš. Odejdi z výpravy.

5.2 Průběh testování

V této podkapitole popisuji metodu rozšíření aplikace mezi účastníky testování. Dále zde uvádím způsob, jakým jsem vzdáleně sledoval pády aplikace na zařízeních testerů.

5.2.1 Distribuce aplikace

Účastníkům testování jsem zaslal odkaz ke stažení aplikace pomocí služby Firebase App Distribution. Služba umožňuje vývojáři spravovat jednotlivé verze aplikace a ty distribuovat různým skupinám uživatelů. Každému testerovi v takovém případě přijde email s odkazem na stažení poslední verze. Služba je navíc přehledná a jednoduchá – pro rozšíření aplikace mezi účastníky stačí do webové aplikace nahrát *.apk*[42] nebo *.aab*[43] soubor a uvést email adresáta. O zbytek se pak postará Firebase. Skvělou vlastností je fakt, že služba dokáže zobrazit, v jakém stavu distribuce se uživatel nachází. Jako vývojáři tak vidíme, kdo si email alespoň zobrazil a kdo si už aplikaci reálně nainstaloval.

5.2.2 Sledování pádů aplikace

Pro monitorování pádů aplikace na cizích zařízeních jsem použil nástroj Firebase Crashlytics. Ten dokáže reportovat pády aplikace a zobrazit je s dalšími užitečnými informacemi, jimiž je například *stack trace*[44], záznam z logu[45] nebo některé základní údaje o zařízení. Přehled všech pádů pak vidíme ve webové aplikaci zvané Firebase Console.

Abychom Crashlytics zprovoznili, musíme nejprve přidat do zdrojového kódu požadovanou závislost. Závislosti na cizí knihovny se v Androidu přidávají do souborů Gradle – nástroje pro automatizované sestavování (build) programů. [41] Tento nástroj toho ale umí ve skutečnosti mnohem více. Zvládne spouštět testy, generovat dokumentaci nebo dodávat externí knihovny. [42] Můžeme s ním i konfigurovat program či exportovat balíčky. Jakmile Gradle stáhne Firebase Crashlytics SDK, nemusíme v kódu pak již nic deklarovat a knihovna od té doby funguje automaticky. Na druhou stranu, Crashlytics SDK obsahuje metody, kterými můžeme případně monitoring upravit nebo zpřesnit o další informace. SDK jsem proto propojil s populární logovací[46]knihovnou Timber.

[42]Instalační formát Android aplikací. Jedná se o zkratku pro Android Application Package.

[43]Balíček pro publikování Android aplikací. Google Play dokáže z tohoto balíčku vytvořit optimalizované *.apk* soubory. Jedná se o zkratku pro Android App Bundle.

[44]Seznam vybraných metod v průběhu pádu programu.

[45]Seznam užitečných informací, které pomáhají s vývojem a debugováním programu.

[46]Logování je zaznamenávání informací, které pomáhají s vývojem a debugováním programu.

Obrázek 5.1: Gradle umožňuje vytvořit rozdílné verze aplikace. V mém případě jsem vytvořil produkční a testovací verzi, které se liší připojením ke vzdálenému API a úrovní logování. Testovací verze zároveň obsahuje knihovnu pro monitoring HTTP požadavků a odpovědí.

Součástí Timberu je i možnost „zasadit vlastní strom“ (neboli uzpůsobit log vlastním potřebám). Do mého zdrojového kódu jsem tak „zasadil“ speciální strom, který v případě produkční verze aplikace nahraje na Firebase zalogovanou chybovou hlášku:

```
private class CrashReportingTree : Timber.Tree() {
   override fun log(priority: Int, tag: String?, message: String, t:
       Throwable?) {
       if (priority in listOf(VERBOSE, DEBUG, WARN)) {
           return
       }

       t?.let {
           Firebase.crashlytics.log(message)
           Firebase.crashlytics.recordException(it)
       }
   }
}
```

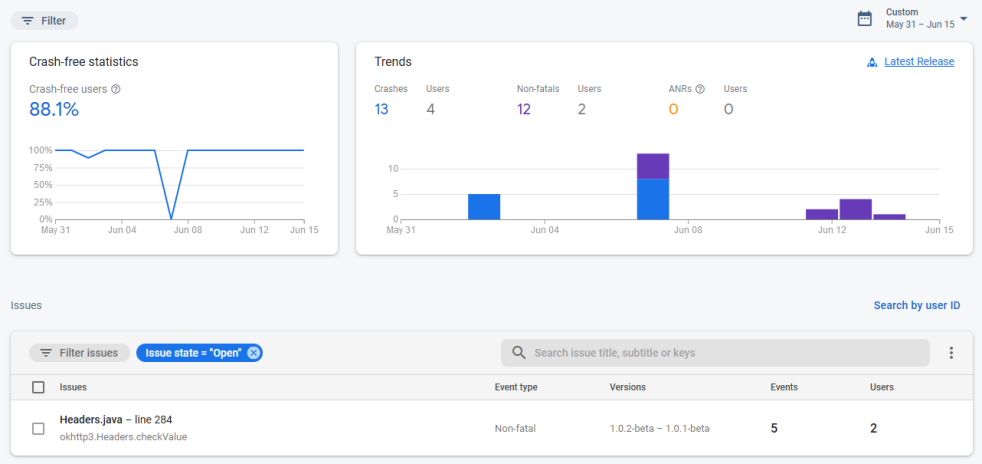
Ukázka kódu 5.1: Přesměrování logování knihovny Timber do Firebase

Hned první den testování jsem díky Firebase odhalil chybu v aplikaci, která způsobovala pád aplikace na tabletech. Tu jsem nakonec stihl opravit ještě předtím, než si špatnou verzi stáhla většina zbylých účastníků. Po opravě této chyby se pak již po celou dobu průběhu testování neobjevil žádný další pád. Díky přesměrování chybových hlášek z Timberu do Firebase jsem nicméně přišel na jiný zákeřný bug[47], kvůli kterému v některých případech nefungovalo stahování zabezpečených dokumentů.

Pokud si totiž uživatel chce stáhnout zaheslovaný dokument, musí zadat klíč. Klíč poté mobilní aplikace odešle webové službě pomocí HTTP hlavičky. Problém je ale ten, že hlavička podporuje zpravidla pouze ASCII[48]znaky. [43]

[47]Nepředpokládaná chyba v programu.

Proto v situaci, kdy někdo zadal jako klíč slovo obsahující jiné znaky než ASCII, například slovo „řeřicha“, zkolaboval požadavek z důvodu neplatné hlavičky ještě u klienta. Tuto pro mě nečekanou chybu jsem nakonec opravil po ukončení testování kódováním hlavičky do formátu Base64. Stejně tak jsem po konci testování opravil i poslední, v pořadí třetí bug – chybné zobrazení vozů a míst během editování dopravy.



Obrázek 5.2: Ukázka webové aplikace Firebase Crashlytics během uživatelského testování

5.3 Vyhodnocení uživatelského testování

V závěru uživatelského testování jsem zaslal účastníkům k vyplnění webový dotazník Google Forms. Otázky jsem rozdělil do několika sekcí podle role. Každý účastník tak správně měl odpovídat pouze na otázky související s jeho úkoly. U většiny otázek v dotazníku měl respondent na výběr z pěti stejných odpovědí. Pro lepší představu uvádím jednu konkrétní otázku:

Přidal/a jsem let z Drážďan do Osla bez problému.

- Souhlasím.
- Spíše souhlasím.
- Spíše nesouhlasím.
- Nesouhlasím.
- Tento úkol jsem nedělal/a.

[48] Z anglického American Standard Code for Information Interchange, jedná se o tabulku zakódovaných znaků anglické abecedy a dalších znaků.

Pokud účastník (spíše) nesouhlasil, mohl svůj nesouhlas odůvodnit. Odpovědi byly anonymní.

5.3.1 Přihlášení a registrace

Nutno podotknout, že mi tato sekce udělala radost. Téměř všem účastníkům se podařilo zaregistrovat i přihlásit do aplikace zcela bez problému. Se změnou hesla také neměl nikdo potíže. U odhlášení se pak našel jeden respondent, který během odhlášení z aplikace lehce zaváhal. Žádný účastník se ale nevyjádřil, že by si způsob přihlášení nebo registrace představoval jinak.

5.3.2 Sekce určená pro administrátory

Sekci pro administrátory vyplnilo pět lidí, což je překvapivé, neboť účastníci s rolí administrátora byli jenom tři. S přidáním nové cesty neměl žádný dotazovaný větší problém, pravdou ovšem je, že tři administrátoři s přidáním cesty bez problému spíše jen souhlasili. Jeden z nich dodal, že je podle něj UI nekonzistentní.

Hůře ovšem dopadla otázka „Přidal/a jsem nového editora do mé cesty bez problému.“ Zde se našel respondent, který spíše nesouhlasil. Svojí odpověď odůvodnil tím, že tuto možnost našel v aplikaci až na několikátý pokus. Další dva dotázaní by pak přidali odkazy známých uživatelů nebo našeptávač. Dále se během přidávání dopravy nepozdávaly jednomu respondentovi popisky atributů a navrhl přidat nápovědu. Zbylé úkoly již ale splnili všichni administrátoři bez obtíží.

5.3.3 Sekce určená pro editory

Úkoly editorů, patrně nejtypičtější role v aplikaci, dopadly dle dotazníku překvapivě dobře. Většina respondentů, kteří vyplnili tuto sekci, zvládli jednotlivé úkoly buď naprosto bez výhrad, nebo s jistým zaváháním. Pokud již zaváhali, pak ne ale natolik, aby se o důvodu rozepsali. Přesto celá tato sekce není tak optimistická, jak by se mohlo zdát. Největší problém opět nastal s pozváním dalšího člena do výpravy. Jeden dotázaný se vyjádřil tak, že tuto možnost hledal dlouho. Na druhou stranu, s přidáním nového člena jako diváka neměla druhá skupina editorů větší obtíž.

5.3.4 Sekce určená pro diváky

Největší neshoda mezi odpověďmi respondentů s rolí diváka nastala hned u první otázky. Otázka zněla: „Podařilo se mi rozkliknout trasu z Prahy do Drážďan a zobrazit si ji na Google Mapách.“ U této otázky jsem zaznamenal i první úplný nesouhlas s otázkou, neboť jeden respondent tuto možnost v aplikaci vůbec nenašel. Druhému pak způsob rozkliknutí trasy nepřišel intuitivní.

Další výtka směřovala k zobrazení mapy s již přidanými body zájmu. V *toolbaru* u přehledu cesty se nachází ikonka mapy (ikonka pochází z Material Icons – standardních ikon pro Android aplikace). Po kliknutí na ikonku se otevře mapa s přidanými body zájmu. Jeden dotázaný si ovšem stěžoval, že tuto ikonku nemohl nalézt. Navrhl, že by se mapa mohla nacházet mezi záložkami spolu s itinerářem, přehledem a výdaji.

Rovněž měla jedna respondentka obtíže se změnou jména. Ve své odpovědi vysvětluje: „Byla jsem na stránce s cestou a klikla jsem na tři tečky vpravo nahoře (kontextové menu, pozn. autor). Tam bych prvotně nastavení uživatele očekávala. Bez zásahu (autora této práce – aplikaci jsme testovali spolu vzdáleně, pozn. autor) bych ale asi opustila výpravu. Myslela jsem si totiž, že je to jen přesměrování na domovskou stránku. Poté, co jsem se dostala zpět na stránku s přehledem cest pomocí šipky vlevo nahoře, jsem už se změnou jména neměla problém."

Zbylé úkoly splnili testeři bez většího zaváhání, stojícího za doplňujícím komentářem.

5.3.5 Obecné

Aplikace se zdála přehledná, nebo přehledná s mírnými nedostatky, všem účastníkům testování kromě jednoho. Ten uvedl, že je aplikace ještě přehledná, ale s velkými nedostatky. Konkrétně se 54,5 % respondentů vyjádřilo, že je aplikace přehledná bez výhrad a 36,4 % uvedlo, že je aplikace přehledná s mírnými nedostatky. S používáním aplikace mimo testování pak souhlasilo s, nebo bez výhrad, 91 % dotázaných. Jeden účastník odpověděl, že mu přijde rychlejší domluva ve skupině na Facebooku.

Poté jsem nechal účastníky sepsat, co by v aplikaci upravili nebo přidali. Několik respondentů se shodlo, že by do aplikace přidali notifikace a historii, tedy zobrazení toho, kdo co změnil. Jeden účastník by také rád našel v aplikaci odkaz do jiného komunikačního prostředku (např. Messengeru nebo WhatsAppu). Další dotázaný by pro změnu uvítal, kdyby na mapě s přidanými body viděl i trasu mezi nimi.

Mimo to se jedna účastnice vyjádřila, že by k výběru místa z Google našeptávače přidala i možnost vybrat bod ručně z mapy. V neposlední řadě by jiný člen testování upravil implicitně zvolené datum během přidávání bodu zájmu. Podle něj by bylo užitečné, kdyby aplikace sama nabízela den dle posledního přidaného bodu.

Účastník, který vytkl aplikaci velké nedostatky co se týče přehlednosti, uvedl: „Upravil bych konzistenci při přidávání atributů v bodě zájmu. Mátlo mě přidávání místa pomocí ikony lupy, i když můžu na políčko normálně kliknout. Tři tečky (kontextové menu, pozn. autor) a ikonku mapy na jedné obrazovce jsem objevil náhodou a neseděly mi se zbytkem aplikace. A taky nemám rád velké plus na rohu aplikace, co dělá zázraky. Chtěl bych mít tlačítka v meníčkách."

Poslední tester uvedl, že by rád viděl po rozkliknutí cesty přehled místo itineráře.

Na úplném konci dotazníku jsem dal účastníkům prostor pro napsání toho, co měli zrovna „na srdíčku“. Několik respondentů se vyjádřilo, že aplikace vypadá graficky dobře a má potenciál. Jiný účastník zmínil radost a poslední, toho mi by bylo popravdě líto, měl na srdíčku šelest.

5.4 Shrnutí

Uživatelské testování dopadlo pozitivně. Během testování nespadla aplikace ani na jednom zařízení (po úvodní opravě chyby, viz podkapitola 5.2.2) a sloužila dobře svému účelu. Na druhou stranu, účastníci testování přišli s několika velmi zajímavými připomínkami.

Systém přidávání nových členů do výpravy není z hlediska UX naimplementován dobře. Aktuální způsob a umístění si tak zaslouží úpravu, které se budu věnovat jako první. V hlavě mám ale již nyní myšlenku na možnost sdílení cesty s dalšími lidmi pomocí odkazu. Za úvahu také stojí implementace našeptávače, nicméně ten budu muset dále zvážit z bezpečnostního hlediska.

Dle získaných odpovědí by pak u některých funkcí či atributů stálo za to je doplnit o nápovědu. Tím se budu zabývat následně. Stejně tak budu přemýšlet, jak vylepšit současné grafické rozhraní. Mám na mysli umístění ikonky mapy, zobrazení přehledu cesty jako první záložky, spojení bodů v mapě či možnou úpravu obrazovky pro přidání bodu zájmu. Jak uvedli někteří respondenti, aplikace by si rovněž zasloužila přidání notifikací a evidenci úprav.

I přes výše zmíněné nedostatky je ale aplikace již v tomto stádiu vývoje pro 91 % uživatelů přehledná a stejné množství uživatelů si dokáže představit, že by aplikaci v budoucnu používalo. Uživatelské testování tak ukázalo, že aplikace míří dobrým směrem a dalo podnět pro mnoho užitečných úprav a vylepšení.

Závěr

V této diplomové práci jsem se zabýval vývojem mobilní aplikace pro Android, jež poskytuje uživatelům rozhraní pro společné plánování cest. O zajištění funkcí pro organizaci a správu výletů mezi účastníky se stará webová služba, která komunikuje s mobilní aplikací architektonickým stylem REST.

V první kapitole jsem seznámil čtenáře s již existujícími mobilními aplikacemi na trhu, jež poskytují svým uživatelům funkce srovnatelné s tématem této diplomové práce.

V další kapitole jsem se věnoval analýze požadavků, v rámci které jsem podrobně definoval jednotlivé funkční a nefunkční požadavky. V této kapitole jsem také popsal cílové uživatele aplikace a pro lepší ilustraci ji doplnil o diagram případů užití.

Po analýze požadavků a cílových uživatelů jsem naimplementoval webovou službu, která zajišťuje funkcionalitu mobilní aplikaci. Službu jsem poté úspěšně nasadil na cloud. V pořadí třetí kapitole jsem tak čtenářům představil zvolenou architekturu a popsal vybrané technologie tvořící základ backendu. Mimo to jsem se podrobně rozepsal o návrhu autentizace a některých bodech implementace.

Čtvrtá kapitola pojednává o vývoji mobilní aplikace. V ní jsem prvně rozebral návrh uživatelského rozhraní a vysvětlil doporučené rozvržení Android aplikací. Po samotné implementaci mobilní aplikace jsem v této kapitole srozuměl čtenáře s vybranými technologiemi a principy užitými v praktické části.

V konečné fázi vývoje jsem nechal otestovat mobilní aplikaci uživateli. Během uživatelského testování proběhla simulace reálného užití aplikace v praxi. Testování jsem následně popsal a vyhodnotil v závěrečné kapitole.

Naplnění cílů

Mobilní aplikace poskytuje uživatelům možnost evidovat budoucí i minulé cesty a zároveň je sdílet s ostatními uživateli. Účastníci společné cesty tak mohou dohromady přidávat dopravu, ubytování, místa nebo aktivity, které je

na cestě čekají. Každý účastník má zároveň přidělenou roli. To znamená, že někteří členové zájezdu mohou bez omezení editovat přidané body, zatímco jiní si je pouze zobrazit. Aplikace rovněž podporuje nahrávání dokumentů, jež jdou navíc zaheslovat. Uživatelské testování pak ukázalo, že aplikace je v současném stádiu vývoje pro naprostou většinu uživatelů přehledná a dokáže si představit, že by ji používalo i v budoucnu.

Budoucnost aplikace

Aplikace je v současné fázi plně funkční a zajišťuje jednoduché rozhraní pro společnou organizaci cest a výletů. Nutno ale dodat, že prostoru pro vylepšení a úprav se v aplikaci nachází mnoho. V první řadě mobilní aplikace nepodporuje možnost evidence společných výdajů. Touto funkcionalitou se tedy budu v budoucím vývoji aplikace zabývat jako první. Aplikace by si rovněž zasloužila přidání notifikací a historie úprav.

Během uživatelského testování mimo jiné vyplynulo, že bych měl pouvažovat nad změnou aktuálního způsobu přidávání nových členů do výpravy a zlepšit rozložení některých prvků grafického rozhraní.

Mobilní aplikaci bych také rád v budoucnu doplnil o další užitečné funkce – zobrazení předpovědi počasí, zobrazení nejbližší webkamery, seznam restaurací, hotelů v okolí, a podobně. Jak je vidno, s odevzdáním mé diplomové práce tak má činnost na vývoji mobilní aplikace nekončí.

Literatura

[1] Worldee: *Naše mise*. [cit. 2022-06-15]. Dostupné z: https://www.worldee.com/homepage/about

[2] Worldee: *Naše mise*. [cit. 2022-06-15]. Dostupné z: https://www.worldee.com/homepage/about

[3] Google Play: *Wanderlog - Trip Planner App*. [cit. 2022-06-15]. Dostupné z: https://web.archive.org/web/20220425085811/https://play.google.com/store/apps/details?id=com.wanderlog.android

[4] Google Play: *Polarsteps - Travel Planner Tracker*. [cit. 2022-06-15]. Dostupné z: https://play.google.com/store/apps/details?id=com.polarsteps

[5] Perez, S.: *Indie travel app Lambus makes group trip planning easier*. TechCrunch, [cit. 2022-06-15]. Dostupné z: https://techcrunch.com/2019/05/28/indie-travel-app-lambus-makes-group-trip-planning-easier

[6] Google Play: *Lambus — Travel Planner*. [cit. 2022-06-15]. Dostupné z: https://play.google.com/store/apps/details?id=io.lambus.app

[7] Google Play: *TripIt: Travel Planner*. [cit. 2022-06-15]. Dostupné z: https://play.google.com/store/apps/details?id=com.tripit

[8] Mlejnek, J.: *Analýza a sběr požadavků*. Praha: ČVUT v Praze, [cit. 2022-06-07]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/506241/mod_resource/content/7/03.prednaska.pdf

[9] FiMuny: *PB007 – SWING výpisky z přednášek*. Brno, [cit. 2022-06-07]. Dostupné z: http://fi.muny.cz/data/PB007/PB007-pb007-zapisky.docx

[10] Google: *Android's Kotlin-first approach*. [cit. 2022-06-09]. Dostupné z: https://developer.android.com/kotlin/first

[11] Google: *Build Better Apps with Kotlin*. [cit. 2022-06-09]. Dostupné z: https://developer.android.com/kotlin/build-better-apps

[12] Ktor: *Lightweight and Flexible*. JetBrains, [cit. 2022-06-09]. Dostupné z: https://ktor.io

[13] Shpota: *Exposed logo*. JetBrains, [cit. 2022-06-09]. Dostupné z: https://github.com/JetBrains/Exposed/blob/master/docs/logo.png

[14] JetBrains: *Exposed Github*. [cit. 2022-06-09]. Dostupné z: https://github.com/JetBrains/Exposed

[15] StackOverflow: *StackOverflow Survey: Databases*. [cit. 2022-06-09]. Dostupné z: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-databases

[16] Wooldridge, B.: *HikariCP Analyses*. [cit. 2022-06-09]. Dostupné z: https://github.com/brettwooldridge/HikariCP#microscope-analyses

[17] Amazon: *Amazon S3*. [cit. 2022-06-09]. Dostupné z: https://aws.amazon.com/s3

[18] Fielding, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000, [cit. 2022-06-10]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[19] OWASP: *Who is the OWASP® Foundation?* [cit. 2022-06-11]. Dostupné z: https://owasp.org/

[20] Santilli, C.: *What does enctype='multipart/form-data' mean?* StackOverflow, [cit. 2022-06-11]. Dostupné z: https://stackoverflow.com/a/28380690/9723204

[21] ZonerCloud: *Co je to Cloud a proč ho využívat*. [cit. 2022-06-12]. Dostupné z: https://www.zonercloud.cz/napoveda/cloud-server-linux/co-je-to-cloud-a-proc-ho-vyuzivat

[22] Microsoft: *Co je PaaS?* [cit. 2022-06-12]. Dostupné z: https://azure.microsoft.com/cs-cz/overview/what-is-paas/

[23] Wikipedia: *Heroku*. [cit. 2022-06-12]. Dostupné z: https://en.wikipedia.org/wiki/Heroku

[24] GitHub: *Understanding GitHub Actions*. [cit. 2022-06-12]. Dostupné z: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

[25] Wikipedia: *Android version history*. [cit. 2022-06-14]. Dostupné z: https://en.wikipedia.org/wiki/Android_version_history

[26] AndroidStudio: *Android Platform/API Version Distribution*. Snímek z vývojového prostředí [cit. 2022-06-14].

[27] Google: *Mobile app user experiences*. [cit. 2022-06-14]. Dostupné z: https://developer.android.com/topic/architecture#mobile-app-ux

[28] Wikipedia: *Oddělení zodpovědností (informatika)*. [cit. 2022-06-14]. Dostupné z: https://cs.wikipedia.org/wiki/Odd%C4%9Blen%C3%AD_zodpov%C4%9Bdnost%C3%AD_(informatika)

[29] Google: *Recommended app architecture*. [cit. 2022-06-14]. Dostupné z: https://developer.android.com/topic/architecture#recommended-app-arch

[30] Google: *UI layer*. [cit. 2022-06-14]. Dostupné z: https://developer.android.com/topic/architecture#ui-layer

[31] Google: *ViewModel Overview*. [cit. 2022-06-14]. Dostupné z: https://developer.android.com/topic/libraries/architecture/viewmodel

[32] Google: *Data layer*. [cit. 2022-06-14]. Dostupné z: https://developer.android.com/topic/architecture#data-layer

[33] Medium: *MVVM Pattern using Architecture components*. [cit. 2022-06-14]. Dostupné z: https://miro.medium.com/max/1400/0*PKo4mQsOOGUqPlVp.png

[34] Wong, D.: *11 Weeks of Android: Jetpack*. Google, [cit. 2022-06-14]. Dostupné z: https://android-developers.googleblog.com/2020/07/11-weeks-of-android-jetpack.html

[35] Google: *Monzo reduced over 9,000 lines of code and improved registration dropout by 5x with CameraX*. [cit. 2022-06-14]. Dostupné z: https://developer.android.com/stories/apps/monzo-camerax

[36] Google: *Data Binding Library*. [cit. 2022-06-14]. Dostupné z: https://developer.android.com/topic/libraries/data-binding

[37] Medium: *Navigation Component Starting Guide 1-Navigation Between Fragments*. [cit. 2022-06-15]. Dostupné z: https://miro.medium.com/max/875/1*YIQHQmS_wneHSl3Ur055OA.png

[38] Sells, C.; Poiesz, B.; Ng, K.: *Use Android Jetpack to Accelerate Your App Development*. Google, [cit. 2022-06-15]. Dostupné z: https://android-developers.googleblog.com/2018/05/use-android-jetpack-to-accelerate-your.html

[39] Google: *Navigation*. [cit. 2022-06-15]. Dostupné z: https://developer.android.com/guide/navigation

[40] Kučerová, M.: *Mobilní aplikace Seznamovák*. Praha: ČVUT v Praze, [cit. 2022-06-15].

[41] Gradle: *What is Gradle?* [cit. 2022-06-18]. Dostupné z: https://docs.gradle.org/current/userguide/what_is_gradle.html

[42] Google: *Configure your build*. [cit. 2022-06-18]. Dostupné z: https://developer.android.com/studio/build

[43] Kopseng, C. E.: *Possible types of a HTTP header value*. StackOverflow, [cit. 2022-06-18]. Dostupné z: https://stackoverflow.com/a/50676124

Příloha A

Seznam použitých zkratek

AAB Android App Bundle

API Application Programming Interface

APK Android Application Package

ASCII American Standard Code for Information Interchange

CPU Central Processing Unit

CRUD Cread, Read, Update, Delete

DAO Data Access Object

DI Dependency Injection

DSL Domain Specific Language

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JSON JavaScript Object Notation

JWT JSON Web Token

OOP Objektově orientované programování

ORM Objektově relační mapování

OS Operační systém

PaaS Platform as a Service

PDF Portable Document Format

REST Representational State Transfer

SDK Software Development Kit

SQL Structured Query Language

UI User Interface

URL Uniform Resource Locator

UX User Experience

XML Extensible Markup Language

Příloha **B**

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `apk` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `testing` ..... uživatelské testování
  - `answers.pdf` ..... odpovědi z uživatelského testování ve formátu PDF
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
- `ui` ..... adresář se snímky výsledné aplikace