FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Mobilní klientská aplikace systému HouseKeeper pro Android
Student:	Tomáš Rys
Vedoucí:	Ing. Zdeněk Rybola, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

HouseKeeper je webová aplikace v PHP pro správu domácnosti. Systém umožňuje mimo jiné správu inventáře a evidenci spotřeby energií.

Cílem bakalářské práce je vytvoření mobilní klientské aplikace pro systém HouseKeeper pro operační systém Android. Aplikace by měla umožnit především následující funkce systému:
- přihlášení uživatele a zobrazení jeho domácností
- správa inventáře domácnosti - zejména evidence jednotlivých kusů, jejich kategorizace, umístění a parametry

Při řešení práce postupujte dle praktik softwarového inženýrství. Práce by měla obsahovat zejména:
- analýzu stávající webové aplikace HouseKeeper, jejího rozhraní a funkcí
- analýzu existujících mobilních aplikací řešících stejnou problematiku
- analýzu konkrétních požadavků na mobilní aplikaci
- návrh implementace mobilní aplikace
- implementace požadavků vybraných společně s vedoucím práce
- patřičné otestování aplikace a její vhodná dokumentace

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 15. ledna 2024 v Praze.

Bakalářská práce

MOBILNÍ KLIENTSKÁ APLIKACE SYSTÉMU HOUSEKEEPER PRO ANDROID

Tomáš Rys

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Zdeněk Rybola, Ph.D.
16. května 2024

Odkaz na tuto práci: Tomáš Rys. *Mobilní klientská aplikace systému HouseKeeper pro Android.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat svému vedoucímu, Ing. Zdeňku Rybolovi, Ph.D., především za jeho zpětnou vazbu při vývoji aplikace. Dále bych chtěl poděkovat své rodině, která mě po celou dobu studia podporovala.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 16. května 2024

Abstrakt

Tato bakalářská práce se zabývá analýzou, návrhem a implementací mobilní aplikace pro systém HouseKeeper pro operační systém Android. Při implementaci aplikace jsem použil programovací jazyk Kotlin a Jetpack Compose framework. Aplikace komunikuje s backendem systému HouseKeeper pomocí REST API a umožňuje uživateli spravovat domácnosti, sdílet je s ostatními uživateli a spravovat jejich inventáře. V každém inventáři lze spravovat lokace, kategorie, produkty a jednotlivé kusy.

Klíčová slova mobilní aplikace, Android, Kotlin, Jetpack Compose, MVI vzor, správa domácností, správa inventářů

Abstract

This bachelor's thesis deals with the analysis, design, and implementation of a mobile application for the HouseKeeper system for the Android operating system. During the implementation of the application, I used the Kotlin programming language and the Jetpack Compose framework. The application communicates with the backend of HouseKeeper system by using REST API and allows users to manage households, share them with other users, and manage their inventories. Within each inventory, user can manage it's locations, categories, products and individual items.

Keywords mobile application, Android, Kotlin, Jetpack Compose, MVI pattern, household management, inventory management

Seznam zkratek

AAA	Arrange Act Assert
API	Application Programming Interface
APK	Android Package Kit
ARM	Advanced RISC Machine
CRUD	Create, Read, Update, Delete
DSL	Domain Specific Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MVC	Model-View-Controller
MVI	Model-View-Intent
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
OS	Operační systém
QR	Quick Response
REST	Representational State Transfer
SSE	Server-Sent Event
UI	User Interface
VPN	Virtual Private Network
XML	Extensible Markup Language

Úvod

V dnešní době nespočet rodin k jejich hlavnímu bydlení vlastní i nějaké rekreační obydlí, kam jezdí na víkendy a k tomu ještě mohou mít navíc nějaké skladové prostory, například pronajatou garáž nebo skladovací box na skladování věcí, co nejsou tak často potřeba. Udržovat si v paměti nebo někde na papíře, který lze lehce ztratit, informace o tom co je kde uloženo, může být náročné, obzvláště když je může využívat více členů rodiny a každý si to může skladovat podle sebe. Potom si vzpomenout a najít věc, kterou jsem před měsíci někde uložil, může trvat i hodiny.

Proto kolega Bc. Nicolas Stefan Maskaľ s pomocí kolegy Bc. Dávid Jenčo, který se hlavně podílel na vývoji jádra a následně na modulu pro řešení energií, vytvořil webovou aplikaci HouseKeeper pro správu inventáře v jednotlivých domácnostech, aby se těmto problémům předešlo. Ale ne každý má na chalupě nebo někde v garáži počítač po ruce, aby si informace, co právě uložil a kam, mohl hned zapsat. Pamatovat si, co jsem uložil nebo si to zapsat někam do poznámek, abych to mohl později uložit, je velmi nepraktické a uživatele to může odradit, aby HouseKeeper vůbec používal. Obzvláště když to lze lehce zapomenout, ne nadarmo se říká „Co můžeš udělat dnes, neodkládej na zítřek“.

Na druhou stranu v dnešní době má většina lidí u sebe chytrý telefon, do kterého by si mohli uložit informace hned a předejít tím nepříjemnostem spojeným se zapamatováním nebo zapsáním a následném uložení do webové aplikace. Bakalářská práce se tedy zabývá vývojem mobilní aplikace pro HouseKeeper pro operační systém Android. Jejím cílem je uživatelům usnadnit ukládání informací, pokud nemají počítač po ruce, ale chytrý telefon ano.

Cíle práce

Hlavním cílem této práce je navrhnout a implementovat mobilního klienta pro HouseKeeper pro operační systém Android, ve kterém si uživatelé budou moci vytvářet a spravovat domácnosti a jejich inventáře. Zejména evidovat jednotlivé kusy, jejich kategorizaci, umístění a parametry. HouseKeeper umožňuje spravovat a evidovat energie, to je však mimo rozsah bakalářské práce a bude rozšiřováno později.

Cílem teoretické části je analýza webové aplikace HouseKeeper, včetně jejího rozhraní a funkcí, analýza existujících mobilních aplikací řešících stejnou problematiku a dohodnutí se s vedoucím Ing. Zdeňkem Rybolou, Ph.D. na konkrétních požadavcích mobilní aplikace.

Cílem praktické části je návrh aplikace, její architektura a jaké technologie použiji, pak samotná implementace aplikace, kde splním všechny požadavky na funkcionalitu definované v praktické části a na konec aplikaci patřičně otestovat a zdokumentovat.

Kapitola 1
Analýza

V této kapitole jsem se zaměřil na analýzu stávající webové aplikace HouseKeeper. Následně jsem udělal rešerši Android aplikací řešících podobnou problematiku. Nakonec jsem udělal analýzu požadavků na aplikaci.

1.1 Analýza stávající webové aplikace HouseKeeper

HouseKeeper je webová aplikace pro správu domácností. Aplikace se skládá z jednoho hlavního modulu pro správu uživatelů a domácností, na kterém závisí ostatní moduly, mezi které momentálně patří modul pro správu inventáře a energií. V analýze se ale budu zabývat pouze hlavním modulem a modulem pro správu inventáře, jelikož cílem mobilní aplikace je implementovat tyto dva moduly. Modul pro správu a evidenci energií je mimo rozsah práce.

Hlavní modul nabízí následující funkcionalitu:

- Správa uživatelů:
  - Registrace.
  - Přihlášení.
  - Obnovení hesla.
  - Změna jména a hesla.
- Správa domácností:
  - Vytváření, mazání a úprava domácností.
  - Zobrazení přehledu mých domácností nebo domácností, kterých jsem členem.
  - Pozvání uživatelů do domácností.
  - Přijetí nebo odmítnutí pozvánky.
  - Přidělování a upravování práv členům pro přístup k jednotlivým modulům.

Každá domácnost má právě jeden inventář, o který se stará modul pro správu inventáře. Každý inventář se skládá ze čtyř základních pilířů, lokací, kategorií, produktů a jednotlivých kusů. Modul nabízí následující funkcionalitu:

- Správa lokací:
  - Vytváření a jejich hierarchické skládání.
  - Zobrazení přehledu lokací.

  - Úprava a mazaní lokací.
- Správa kategorií:
  - Vytváření a jejich hierarchické skládání.
  - Vytváření, mazání a úprava atributů jednotlivých kategorií.
  - Zobrazení přehledu kategorií.
  - Úprava a mazaní kategorií.
- Správa produktů:
  - Vytváření.
  - Zobrazení přehledu produktů s možností řazením, vyhledáváním a filtrováním.
  - Úprava a mazání produktů.
  - Vytváření, mazání a úprava atributů jednotlivých produktů.
  - Zobrazení historie manipulace kusů produktu.
- Správa jednotlivých kusů:
  - Vytváření jednotlivých kusů z produktů.
  - Úprava a mazaní kusů.

1.1.1 Systémová architektura

Aplikace je rozdělena na dvě části frontend, který se stará o zobrazování informací a interakci s uživatelem a backend, který řeší logiku funkcí popsaných v předchozí sekci.

1.1.1.1 Komunikace s backendem

Backend nabízí REST (Representational State Transfer) API (Application Programming Interface) rozhraní, pomocí kterého s ním frontend komunikuje přes HTTP (Hypertext Transfer Protocol) protokol a mobilní klient bude toto rozhraní využívat také. Rozhraní backendové části navrhl Bc. Nicolas Stefan Maskaľ v jeho práci v kapitole 4.5 [1].

REST API je pouze jednosměrná komunikace, frontend pošle požadavek na backend a backend odpoví. Poté se spojení ukončí. Pokud by chtěl backend komunikovat s frontendem, například že uživateli přišla pozvánka do domácnosti, není to možné. Tedy pokud chce frontend zijstit, zdali uživatel nemá nové pozvánky, musí se pokaždé poslat požadavek na backend, čemuž se říká tzv. „polling“, což je technika, při které dochází k pravidelnému dotazování backendu, zdali neexistují nové informace. Přičemž dochází k zbytečnému zatěžování sítě.

Použitím SSE (Server-sent events)[2], které také komunikují jednosměrně, ale po vyřízení požadavku z frontendu na backend se spojení neukončuje a backend může dál informovat frontend o změnách v reálném čase, dokud frontend spojení neukončí. Nicméně frontend už pomocí tohoto spojení komunikovat nemůže, a pokud by chtěl jiné informace, musel by poslat nový požadavek. Tím by se polling eliminoval, jelikož backend by sám informoval frontend, pokud by došlo k nějaké změně.

Stejný efekt by mělo použití WebSocketů [3], což je oboustraná komunikace, kdy po navázání spojení mezi frontendem a backendem nedochází k uzavření spojení a frontend i backend si mezi sebou mohou vyměňovat infromace v reálném čase, dokud jeden z nich spojení nepřeruší.

Podobný problém může nastat při používaní pouze REST API, kdy dochází k sdílení dat mezi více uživateli, což HouseKeeper umožňuje. Například jeden uživatel upraví nějaký produkt a druhý uživatel, který má stále původní informace o upraveném produktu, s ním chce manipulovat, ale neví, že produkt byl změněn. Díky tomu může dostávat chybné odpovědi od backendu, protože stav lokálních dat a dat na backendu je jiný.

1.1.1.2 Návrh rozhraní

Rozhraní využívá HTTP [4] metody GET na získání dat ze serveru, POST na vytváření nových dat, PUT na modifikování dat a DELETE na mazání dat. Data jsou posílána ve formátu JSON.

Na modifikování dat by u některých metod bylo vhodnější použít metodu PATCH [5], jelikož povoluje částečné modifikování dat. Oproti metodě PUT, kde je zapotřebí vždy posílat celý model, i když jsem měnil pouze jednu hodnotu. Tedy u větších modelů, jako je například modifikace jednotlivých kusů, kdy uživatel bude měnit pouze počet zbývajících kusů, se musí posílat všechna data, i ta, která jsou nezměněna, což může zbytečně zatěžovat síť, a použitím metody PATCH by se tomuto problému předešlo, jelikož by stačilo posílat pouze modifikovaná data.

Například by šlo použít JSON Patch [6], který definuje operace, které lze provést na nějakém modelu a je vhodný pro použití s PATCH metodou.

Navíc rozhraní neposkytuje metodu pro smazání veškerých dat o uživateli, které je pro nahrání aplikace na Google Play potřeba. Google Play směrnice [7] říká, že pokud aplikace umožňuje vytvoření uživatelského účtu, potom musí aplikace také umožnit zažádat si o smazání účtu.

1.1.1.3 Autentizace

Aplikace k ověření identity uživatelů využívá Cookies, na jejich návrhu a implementaci se podílel Bc. Nicolas Stefan Maskaľ v jeho práci v kapitole 4.4 [1] a Bc. Dávid Jenčo v jeho práci v kapitole 3.8 [8]. Po úspěšném přihlášení dostane uživatel Cookie, který se pak posílá s dalšími požadavky na backend a backend tak může ověřit identitu uživatele. Cookie má od backendu nastavený datum expirace. S každým dalším požadavkem se datum expirace prodlužuje, pokud ale uživatel nepošle žádný požadavek a Cookie vyprší, uživatel nemá jinou možnost, než se znovu přihlásit.

Pro mobilní aplikace se tento přístup moc nehodí, protože je uživatelsky přívětivější se přihlásit pouze jednou a pokaždé, když aplikaci spouštím, už být přihlášen, jako to funguje u většiny mobilních aplikací.

Jedním z řešení je uložit přihlašovací údaje do lokálního úložiště zařízení. Z bezpečnostního hlediska tento přístup není vhodný, i v případě, že jsou informace zašifrovány. Protože kdyby potenciální útočník získal přístup k našemu zařízení, mohl by se dostat ke klíčům použitých pro šifrování a tím přečíst naše přihlašovací údaje.

Dalším z řešení by bylo místo Cookies použít OAuth 2.0 framework [9], kde by uživatel po úspěšném přihlášní obdržel přístupový token a obnovovací token. Oba tokeny mají také nastavený datum expirace, přístupový token má nastavenou krátkou dobu a obnovovací token má nastavenou delší dobu, klidně i týdny. Tedy v případě, že přístupový token vyprší, se automaticky, pomocí obnovovacího tokenu, zažádá o nový přístupový token a uživatel by se tak nemusel znovu přihlašovat. Tím pádem by byl při každém dalším spuštění aplikace už přihlášen.

1.2 Požadavky

V této sekci se zaměřím na funkční a nefunkční požadavky, které jsou na aplikaci kladeny. Tyto požadavky vymezují hranice systému a pomáhají si vyjasnit zadání se zadavatelem.

1.2.1 Funkční požadavky

Funkční požadavky vymezují co všechno by systém měl umět.

- FP1. Správa uživatelů

  Aplikace uživatelům umožní registraci, přihlášení, změnu hesla a uživatelského jména a resetování hesla.

- FP2. Správa domácností

  Aplikace uživatelům umožní vytvářet, upravovat a mazat domácnosti. Také umožní sdílení domácností s ostatními uživateli. Dále si uživatelé budou moci zobrazit přehled domácností, kterých jsou členem. Pokud danou domácnost vlastní, budou mít možnost členům nastavovat práva pro jednotlivé moduly.

- FP3. Správa lokací

  Aplikace uživatelům umožní vytvářet, upravovat a mazat lokace, které jsou vázané na konkrétní domácnost. Lokace bude možno hierarchicky skládat, neboli každá lokace pod sebou může mít podlokace, například lokace prosklená skříň je podlokací ložnice. Také umožní zobrazení přehledu lokací.

- FP4. Správa kategorií

  Aplikace uživatelům umožní vytvářet, upravovat a mazat kategorie, které jsou vázané na konkrétní domácnost. Kategorie bude možno hierarchicky skládat, neboli každá kategorie pod sebou může mít podkategorie, například kategorie těstoviny je podkategorií jídlo. Také umožní zobrazení přehledu kategorií. Dále bude možno přidat kategorii výchozí typ a jednotku měření.

- FP5. Správa produktů

  Aplikace uživatelům umožní vytvářet, upravovat a mazat produkty, které jsou vázané na konkrétní domácnost. Produktu bude možno nastavit výchozí lokaci, typ a jednotku měření, pokud uživatel nechce použít výchozí typ a jednotku měření kategorie, a popřípadě změnit hodnoty atributů produktu. Také umožní zobrazení přehledu produktů s možností vyhledávání, filtrování podle typu a jednotky měření, kapacity, kategorie a výchozí lokace a řazení podle názvu a kapacity. Dále bude možno si zobrazit historii kusů produktu.

- FP6. Správa jednotlivých kusů

  Aplikace uživatelům umožní vytvářet, upravovat a mazat jednotlivé kusy, které jsou vázané na konkrétní domácnost. Také umožní zobrazení přehledu kusů s možností vyhledávání, filtrování podle typu a jednotky měření, zbývajícího množství, data spotřeby, kategorie a lokace a řazení podle názvu produktu, data spotřeby a zbývajícího množství.

- FP7. Správa atributů

  Aplikace uživatelům umožní vytvářet, upravovat a mazat atributy. Uživatel bude moci na kategoriích vytvářet definice atributů, které budou vázány na produkty v dané kategorii. Příkladem je atribut značka na kategorii nápoje, každý produkt s kategorií nápoje bude pak mít atribut značka, který lze vyplnit nebo použít výchozí hodnotu z definice atributu kategorie. Podobně to bude u produktů, kde uživatel bude moci vytvářet definice atributů, které budou vázány na jednotlivé kusy, vytvořené z daného produktu.

1.2.2 Nefunkční požadavky

Nefunkční požadavky určují omezení kladena na systém. Neboli, jak by se měl systém chovat.

- NFP1. Mobilní aplikace pro OS Android

  Aplikace je určena pro operační systém Android a měla by běžet na zařízeních s verzí 11 a novější.

- NFP2. Lokalizace

  Aplikace by měla podporovat český a anglický jazyk, s možností jednoduchého rozšíření o další jazyky.

- NFP3. Napojení na HouseKeeper backend

  Aplikace by měla být napojena na backend HouseKeeper, který poskytuje REST API rozhraní.

- NFP4. Optimalizace pro používání na výšku

  Aplikace by měla být optimalizovaná pro používání na výšku. Používání aplikace na šířku by mělo být možné, ale není optimalizováno.

- NFP5. Distribuce aplikace přes APK (Android Package Kit)

  Uživatel aplikace by měl mít možnost nainstalovat aplikaci pomocí APK souboru, který si může stáhnout.

- NFP6. Použití GitLab pipeline a SonarQube

  Při vývoji by měla být použita GitLab pipeline pro sestavení APK souboru, spuštění jednotkových testů a nahrání výsledků statické analýzy na SonarQube server.

1.3 Existující řešení

V této sekci se podívám na existující aplikace řešící stejnou problematiku. Zaměřím se pouze na aplikace pro OS (operační systém) Android, které nabízí podobnou funkcionalitu a porovnám, zdali uspokojují všechny funkční požadavky z předchozí sekce. Nebudu brát v potaz funkcionalitu mimo funkční požadavky. Aplikace jsem zkoušel na Samsung Galaxy Note 9 s vlastní ROM LineageOS 20 s verzí Android 13.

1.3.1 Home Inventory, Food Shopping

Home Inventory [10] je velmi hezky zpracovaná aplikace, má moderní design s intuitivním ovládáním, který je možný vidět na obrázcích 1.1 a 1.2.

Uživatel má možnost vytvářet domácnosti (v aplikaci se používá termín listy) a sdílet je s uživateli, ale bohužel jsou obě funkce ve verzi zdarma omezené a uživatel může mít pouze jednu domácnost a nelze ji sdílet s ostatními uživateli. Dále má uživatel možnost vytvářet a spravovat lokace, kategorie, a produkty pro jednotlivé domácnosti. Lokace lze hierarchicky skládat, ale u kategorií je to omezené pouze na jedno zanoření, tedy lze mít podkategorii, ale už nelze mít podkategorii podkategorie. Bohužel ani kategorie ani produkty nepodporují přidávání atributů.

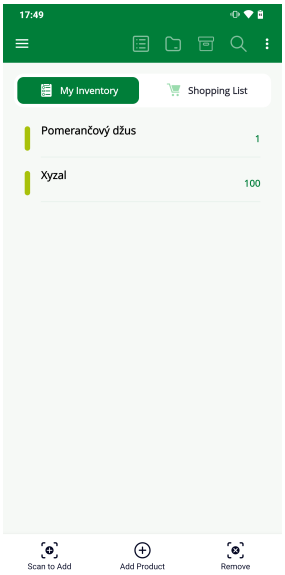
Na druhou stranu aplikace umožňuje vytvoření vlastního typu měření pro produkty a neomezuje uživatele na předem určené, které musí používat, s možností přidat produktu čárový kód, pomocí kterého lze produkty vyhledávat. Čárový kód lze naskenovat fotoaparátem telefonu nebo ručně zadat.

Aplikace nabízí premium verzi, kterou si lze pořídit buď za 24,99 Kč měsíční poplatek, za 199,99 Kč roční poplatek nebo za 489,99 Kč si lze aplikaci koupit se všemi výhodami napořád. V premium verzi dostane uživatel přístup k novým funkcím, které ve verzi zdarma neměl. Především jde o sdílení domácností s ostatními uživateli, vytváření nových domácností a přidávání obrázků k produktům.

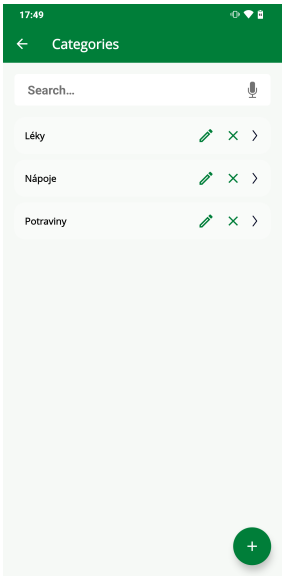
1.3.2 My Stuff Organizer

My Stuff Organizer [11] je také velmi hezky zpracovaná aplikace s moderním designem a intuitivním ovládáním, což je možné vidět na obrázcích 1.3 a 1.4.

V aplikaci lze vytvářet lokace a kategorie, ale nelze je hierarchicky skládat. Bohužel aplikace neumožňuje vytvářet domácnosti a sdílet je s ostatními uživateli a navíc s internetovým připojením se v aplikaci každou chvíli objevují reklamy, které velmi negativně ovlivňují zážitek



Obrázek 1.1 Ukázka jednotlivých kusů z aplikace Home Inventory



Obrázek 1.2 Ukázka kategorií z aplikace Home Inventory

z používání aplikace. Reklamy lze vidět na obrázcích 1.3 a 1.4, navíc po kliknutí na většinu tlačítek se ještě objeví vyskakovací reklama blokující celou obrazovku, která musí být manuálně zavřena.

Podobně jako Home Inventory aplikace umožňuje přidání čárového kódu k produktům, plus lze přidat i QR (Quick Response) kód, pomocí kterých lze vyhledávat. Navíc lze k produktu přidat obrázek, buď z galerie nebo ho lze vyfotit. Bohužel ani kategorie, ani produkty nepodporují přidávání atributů a nejsou rozlišeny jednotlivé kusy a produkty, navíc ve verzi zdarma je uživatel omezen pouze na 30 produktů, u kterých nelze specifikovat jiný typ měření, než je množství.

Aplikace nabízí premium verzi, kterou si lze pořídit buď za 64,99 Kč měsíční poplatek, za 99,99 Kč půlroční poplatek nebo za 179,99 Kč roční poplatek. V premium verzi jsou odstraněny reklamy a navíc si uživatel může vytvořit, kolik produktů potřebuje, a není omezen.

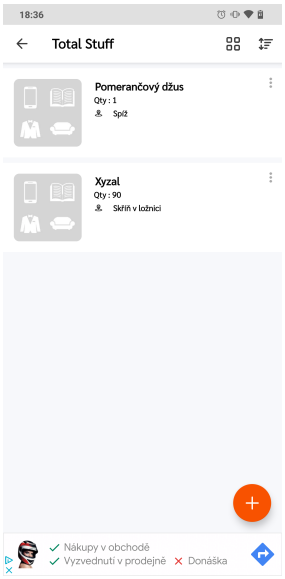
1.3.3 HouseBook – Home Inventory

HouseBook [12] je nově aktualizován a používá Material Design 3, aplikace má tedy moderní design a intuitivní ovládání, což je možné vídět na obrázcích 1.5 a 1.6. Bohužel například v detailu pokoje obsahuje aplikace grafickou chybu, kterou lze vidět na obrázku 1.6, kdy ikona pokoje zasahuje do horní lišty.

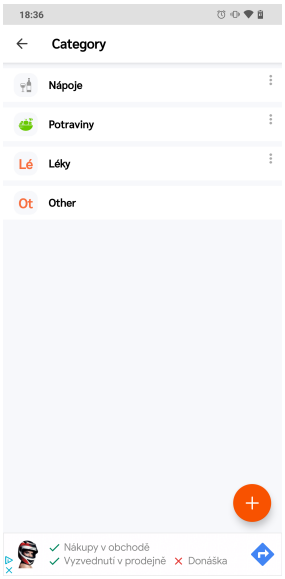
Každý uživatel má ve verzi zdarma přidělen jeden dům, ve kterém lze vytvářet pokoje, ve kterých lze vytvářet hierarchie kontejnerů, které se v daném pokoji nacházejí. Bohužel v aplikaci není možné vytvářet ani kategorie, ani atributy a nelze sdílet dům s ostatními uživateli. Atributy částečně nahrazují štítky, které lze kusům přidávat a pomocí kterých jde následně vyhledávat. Navíc aplikace nerozlišuje mezi jednotlivými kusy a produktem.

K produktům lze přidat QR kód, pomocí kterého lze vyhledávat, a obrázek, ale ve verzi zdarma jsem omezen pouze na 200 produktů. Produkty navíc nepodporují jiný typ a jednotku měření, než je množství.

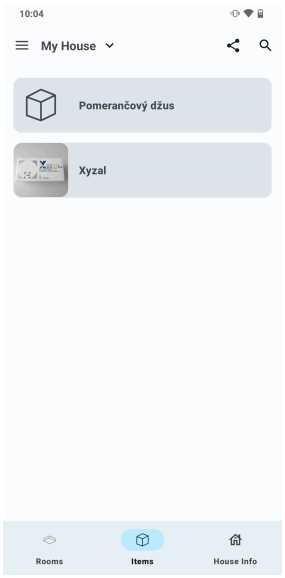
Aplikace nabízí premium verzi, kterou si lze pořídit za jednorázový poplatek 799,99 Kč. V premium verzi může uživatel vytvořit až 500 produktů, přidávat více než jeden obrázek k produktům a navíc dostane druhý dům. Pokud by uživateli nestačilo 500 produktů, lze si za další jednorázový poplatek koupit další.



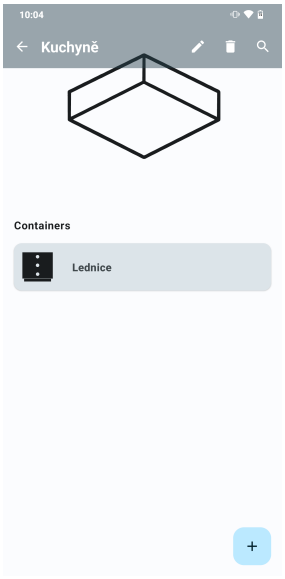
Obrázek 1.3 Ukázka jednotlivých kusů z aplikace My Stuff Organizer



Obrázek 1.4 Ukázka kategorií z aplikace My Stuff Organizer



Obrázek 1.5 Ukázka jednotlivých kusů z aplikace HouseBook



Obrázek 1.6 Ukázka pokoje z aplikace HouseBook

1.3.4 Shrnutí

V tabulce 1.1 jsem shrnul, které funkční a některé relevantní nefunkční požadavky jednotlivé aplikace splňují ve srovnání s HouseKeeper. Fajfkou je označen požadavek, který aplikace splňuje. Symbolem plus je označen požadavek, který aplikace splňuje s určitým omezením, například pouze v premium verzi. Symbolem x je označen požadavek, který aplikace nesplňuje.

Z tabulky je zřejmé, že žádná z porovnávaných aplikací nesplňuje všechnu funkcionalitu jako HouseKeeper. Především žádná z porovnávaných aplikací nenabízí komplexní správu atributů

jako HouseKeeper. Navíc žádná z aplikací nemá plnou podporu pro český jazyk, pouze Home Inventory je částečně přeložený, tedy většina je v češtině, ale najdou se slova, která zůstaly v angličtině. HouseKeeper má tedy šanci konkurovat ostatním aplikacím v této kategorii a přinést něco nového.

	Home Inventory	My Stuff Organizer	HouseBook	HouseKeeper
FP1 správa uživatelů	✓	✓	✓	✓
FP2 správa domácností	+	x	+	✓
FP3 správa lokací	✓	+	✓	✓
FP4 správa kategorií	+	+	x	✓
FP5 správa produktů	+	+	+	✓
FP6 správa jednotlivých kusů	+	x	x	✓
FP7 správa atributů	x	x	x	✓
NFP1 podpora OS Android	✓	✓	✓	✓
NFP2 Lokalizace	+	x	x	✓
NFP4 optimalizace pro používání na výšku	✓	+	✓	✓

Tabulka 1.1 Porovnání splnění jednotlivých funkčních požadavků

1.4 Případy užití

V této sekci jsem z jednotlivých funkčních požadavků vytvořil případy užití [13], které detailněji popisují požadavky na aplikaci nebo přesněji, co by měla aplikace dělat. Klíčovými koncepty jsou aktéři, případy užití a subjekty. Subjekt reprezentuje systém, do kterého případ spadá. Uživatelé nebo jiné systémy, které mohou interagovat se subjekty jsou aktéři.

V mém případě budou aktéři:

- Uživatel.
- Přihlášený uživatel.
- Vlastník konkrétní domácnosti.
- Uživatel s právem číst konkrétní domácnost.
- Uživatel s právem editovat konkrétní domácnost.

Vztahy mezi aktéry lze vidět na obrázku v příloze A.1.

V příloze na diagramech A.2, A.3, A.4, A.5, A.6, A.7 lze pak vidět jednotlivé případy užití, a který aktér je může provést.

Kapitola 2
Návrh

V této kapitole jsem se zaměřil na návrh aplikace. Jakou technologii jsem pro vývoj aplikace zvolil a proč, použitou architekturu a návrh obrazovek aplikace.

2.1 Výběr technologie

Výběr technologie pro tvorbu aplikace je klíčový aspekt, protože od toho se bude odvíjet, v jakém programovacím jazyce bude aplikace napsaná, zdali bude nativní, tedy pouze pro konkrétní platformu nebo multiplatformní, neboli jak už název napovídá pro více platforem.

Porovnám, jaké jsem měl při výběru technologie pro vývoj aplikace možnosti, jejich výhody a nevýhody a na závěr vysvětlím, kterou technologii jsem zvolil a proč.

2.1.1 Jetpack Compose

Jetpack compose [14] je nativní, moderní deklarativní nástroj pro vývoj Android aplikací. Zároveň je oficiální technologií pro vývoj aplikací pro Android, doporučovaný samotným Googlem, který je vývojářem OS Android.

Mezi jeho výhody patří nativní vývoj, který umožňuje přímý přístup k Android API, bez nutnosti další abstrakce nebo pluginů, které nabízí multiplatformí technologie. Ruku v ruce s tím, že je Jetpack Compose nativní, jde i výkon, který bude oproti multiplatformním technologiím lepší, díky přímému přístupu k Android API a optimalizacím, které se soustředí pouze na platformu Android. Další výhodou je, že UI (User Interface) vrstva i logika aplikace se píše v jednom jazyce a to sice v Kotlinu. Celá technologie je postavena na tzv. „Composable“ funkcích. „Composable“ funkce jsou malé znovupoužitelné komponenty, které přijímají data a tvoří UI. Na ukázce 1 je jednoduchá Composable funkce, která pouze zobrazí Hello a jméno, se kterým bude daná funkce volaná.

Na druhou stranu může být jeho hlavní výhoda i nevýhodou, a to, že podporuje pouze platformu Android.

2.1.2 Java + XML

Další možností pro nativní vývoj Android aplikací je Java a XML (Extensible Markup Language). Java pro logiku aplikace a XML pro vzhled. Tyto technologie se používají od počátku Androidu pro vývoj aplikací a jsou stále podporované.

```
@Composable
fun Hello(name: String) {
    Text(text = "Hello, $name")
}
```

Výpis kódu 1 Ukázka tvorby UI pomocí Composable funkce

Podobně jako Jetpack Compose, je výhodou nativní vývoj, přímý přístup k Android API a díky tomu, že je to tradiční způsob pro vývoj aplikací, bude existovat spousta knihoven a návodů, které ulehčí a urychlí vývoj aplikace.

Nevýhodou je, že je to zastaralý a momentálně už nedoporučovaný způsob pro vývoj aplikací. Další nevýhodou je vývoj UI, který díky XML, jak je vidět na ukázce 2, může být zdlouhavý a nepřehledný a jelikož UI a logika používají jiné, technologie jejich integrace může být složitější. Například stejná aplikace napsaná v Jetpack Compose by byla mnohem kratší.

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, $name" />
```

Výpis kódu 2 Ukázka tvorby UI pomocí XML

2.1.3 Compose Multiplatform

Compose Multiplatform [15] je multiplatformní technologie pro vývoj aplikací pro Android, iOS, web, Windows, Linux a MacOS. Compose Multiplatform je postavený na Jetpack Compose a Kotlin Multiplatform, tedy je to moderní a deklarativní nástroj pro vývoj mobilních aplikací.

Výhodou je, podobně jako u Jetpack Compose, že UI i logika se píše v jednom jazyce a to v Kotlinu. Technologie také využívá tzv. „Composable“ funkce pro tvorbu UI. Pokud by uživatel chtěl, tak díky tomu, že je techologie postavena na Kotlin Multiplatform lze sdílet, mezi jednotlivými platformami, například pouze logiku aplikace a UI napsat pro každou platformu pomocí nativní technologie.

Compose Multiplatform je velmi nová technologie, první verze vyšla až na konci roku 2021. Momentálně plně podporuje pouze Android, Windows, Linux a MacOS platformy. Platfoma iOS má podporu pouze v Alpha verzi a web je zatím v experimentální verzi. Tedy oproti ostatním multiplatformním technologiím, které mají oficiální podporu pro všechny tyto zmíněné platformy, je Compose Multiplatform lehce pozadu. Což může být při výběru správné technologie důležitý aspekt.

2.1.4 Flutter

Flutter [16] je multiplatformní technologie pro vývoj aplikací pro Android, iOS, web, Windows, Linux a MacOS. Nabízí moderní a deklarativní tvorbu UI pro všechny platformy. A pro každou platformu je nativně kompilovaný, tedy například pro platformu iOS se použije Swift nebo pro web se použije JavaScript.

Jak už jsem popsal v předešlém odstavci, tak mezi výhody patří podpora většiny platforem, deklarativní tvorba UI a díky tomu, že je nativně kompilovaný, nabízí skoro stejný výkon jako nativní technologie.

Hlavní nevýhodou je, že Flutter je postaven na programovacím jazyce Dart, který, i když je už přes 10 starý, není tak populární a tedy nenabízí tolik knihoven pro ulehčení psaní kódu. Kolikrát si člověk musí napsat vlastní knihovnu. A podobné je to s vyhledávaním rad nebo informací na internetu, kdy se může stát, že podobný problém ještě nikdo neřešil.

Flutter využíva tzv. „Widgety“ pro tvorbu UI. Podobně jako „Composable“ funkce u Jetpack Compose to jsou malé znovupoužitelné komponenty, ze kterých lze stavět UI. Na ukázce 3 je jednoduchý Widget, který zobrazí Hello a jméno, se kterým bude daný Widget vytvořen.

```
class HelloWidget extends StatelessWidget {
    final String name;

    HelloWidget({required this.name});

    @override
    Widget build(BuildContext context) {
        return Text('Hello, $name');
    }
}
```

Výpis kódu 3 Ukázka tvorby UI pomocí Flutter

2.1.5 Shrnutí

Existuje více technologií pro vývoj mobilních aplikací pro Android, převážně těch multiplatformních, například React Native nebo .NET MAUI. Nezahrnul jsem je do výběru, jelikož oproti Flutteru nepřináší žádné větší výhody a je to spíše o preferenci programovacího jazyka používaného těmito technologiemi. A já, kdybych si měl mezi těmito třemi vybírat, zvolil bych Flutter.

V dnešní době, pokud aplikace nedělá žádné výpočetně náročné operace, rozdíl mezi nativní a multiplatformní technologií je zanedbatelný. A není třeba se bát, že by aplikace při volbě multiplatformní technologie byla značně pomalejší.

Díky tomu, že jsem měl při výběru volnou ruku, jediná podmínka byla, aby aplikace běžela na platformě Android. Zvolil jsem Jetpack Compose především kvůli tomu, že je to doporučovaná technologie pro vývoj aplikací samotným Androidem a používá programovací jazyk Kotlin. A pokud by někdy bylo třeba přidat podporu dalších platforem, nebude problém aplikaci předělat do Compose Multiplatform, a to bez nutnosti přepisovat celou aplikaci nebo větších zásahů do kódu.

2.2 Architektura aplikace

V této sekci se podívám na to, jaké mám možnosti při výběru architektury aplikace, a to mezi MVC (Model-View-Controller), MVP (Model-View-Presenter), MVVM (Model-View-ViewModel) a MVI (Model-View-Intent). Tyto nejběžněji používané vzory porovnám, jak z obecného hlediska, tak z hlediska technologie, kterou jsem vybral v předešlé sekci.

Všechny tyto vzory mají společnou předponu MV (Model-View) z čehož vyplívá, že se zabývají tím, jak zpracovat data od uživatele a jak je zobrazit uživateli zpět a mají tři komponenty. Navíc všechny zmíněné vzory vychází z tzv. „Mediator“ vzoru.

„The mediator pattern lets you reduce dependencies between objects. It restricts direct communication between two different layers of objects and forces them to collaborate only via a mediator object or objects.“ [17]

Obrázek 2.1 Ukázka MVC vzoru [17]

V mém případě jsou ty dvě komponenty Model a View a mediátor bude buď Controller, Presenter nebo ViewModel.

2.2.1 Model-View-Controller

Model-View-Controller [17], dále už jen MVC, je z těchto vzorů nejstarší a je typickým příkladem mediator vzoru. Za ta léta se MVC stále vyvíjel, a tedy neexistuje jediný správný způsob, jak tento vzor implementovat. Já se budu soustředit na MVC vzor podle obrázku 2.1.

V tomto vzoru Model a View komponenty spolu nekomunikují přímo, ale pomocí mediátoru, kterému se v tomto případě říká Controller. Vzor funguje tak, že z View komponenty přijde nějaká akce od uživatele, Controller komponenta ji zpracuje a změní nějaké údaje v Model komponentě. Tato změna se pak propaguje zpět přes Controller komponentu zpět do View komponenty, která se na základě toho aktualizuje s novými informacemi.

V Android aplikacích vytvořených použitím technologie Jetpack Compose, se MVC vzor ale moc nepoužívá, ale stojí za zmínku, protože z něj vycházejí ostatní vzory.

2.2.2 Model-View-Presenter

Model-View-Presenter [17], dále už jen MVP, je dost podobný MVC vzoru a tedy obrázek 2.1 tomuto vzoru odpovídá také, s jedinou změnou a to, že mediátor komponenta se nejmenuje Controller ale Presenter.

Princip fungování vzoru je podobný s jednou změnou. Presenter si navíc drží referenci na obrazovku z View komponenty, o kterou se stará. To může vést k jednoduššímu řízení obrazovky z View vrstvy.

V Android aplikacích, vytvořených použitím technologie Jetpack Compose, se MVP vzor moc nepoužívá.

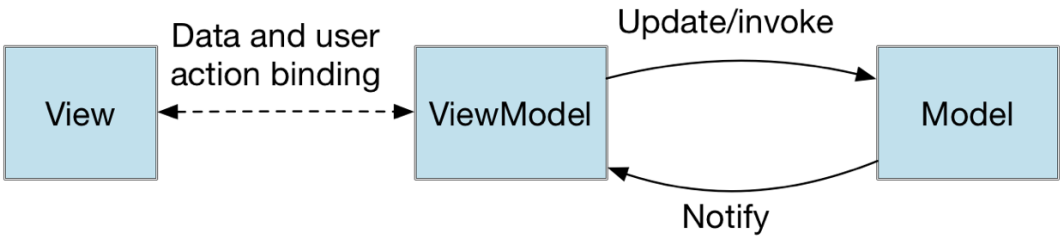
2.2.3 Model-View-ViewModel

Model-View-ViewModel, dále už jen MVVM, podobně jako předešlé vzory se skládá ze tří komponent, jak je vidět na obrázku 2.2, Model, View a ViewModel.

„The view model provides bindings between the view's events and actions in the model. This can happen so that the view model adds action dispatcher functions as properties of the view. In the other direction, the view model maps the model's state to the properties of the view.“ [17] Tedy se od předešlích vzorů líší tím, že ViewModel obsahuje stav a uživatelské akce, na který si View komponenta vytváří vazbu.

Z hlediska použití v Jetpack Compose je vzor běžně používaný. Výhodou použití MVVM s Jetpack Compose je uložení stavu ViewModelu při ukončení aplikace OS Android a opětovného načtení stavu při spuštění aplikace. To je umožněno díky tomu, že každá vlastnost je uložena ve ViewModelu jako jeden stav. [18, 19]

Nevýhodou může být právě to, že je každá vlastnost uložena jako jeden stav. ViewModel pak obsahuje soustu stavů a může být nepřehledný, jak lze vidět na příkladu 4, který obsahuje jen tři stavy.



Obrázek 2.2 Ukázka MVVM vzoru [20]

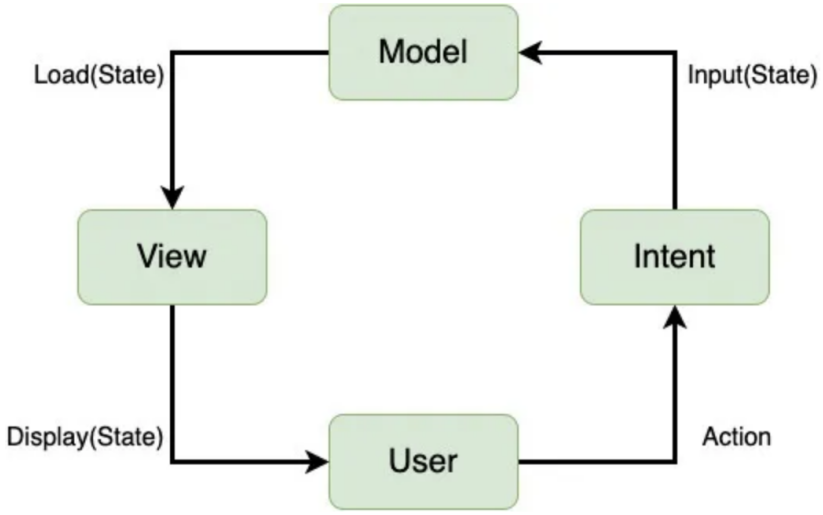
```
class ExampleViewModel(
    private val repository: Repository
): ViewModel() {

    private val _data = MutableStateFlow(listOf<String>())
    val data = _data.asStateFlow()

    private val _isLoading = MutableStateFlow(false)
    val isLoading = _isLoading.asStateFlow()

    private val _error = MutableStateFlow<String?>(null)
    val error = _error.asStateFlow()

    // další stavy a metody

}
```

Výpis kódu 4 Ukázka Jetpack Compose ViewModelu při použití MVVM vzoru

2.2.4 Model-View-Intent

Model-View-Intent [21], dále už jen MVI, je další z možností, kterou použít a navíc se velmi dobře používá s Jetpack Compose. MVI se skládá také ze tří komponent Model, View a Intent, a je určen především pro reaktivní programování. Intent může být rozhraní specifikující akce, které může View komponenta provádět. V Jetpack Compose to bude rozhraní s akcemi, které následně ViewModel zpracovává.

Na obrázku 2.3 lze vidět tok dat poté, co uživatel nebo i samotná aplikace provede změnu.



Obrázek 2.3 Ukázka toku dat MVI vzoru [21]

Z hlediska použití v Jetpack Compose je to podobné jako použití MVVM. Řeší to nevýhodu MVVM, a to tak, že ViewModel nebude mít spoustu stavů, ale všechny stavy se zapouzdří do neměnitelného objektu a ten bude vystaven jako jeden stav.

Bohužel touto změnou přijdeme o snadné obnovení stavu aplikace po ukončení. Protože jeden stav může obsahovat i vlastnosti, které obnovit nechceme a museli bychom tedy stav po obnovení ještě upravovat. Další nevýhodou je, že místo toho, aby ViewModel obsahoval spoustu stavů a akcí, tak obsahuje jednu metodu, která zpracovává akce z View komponenty, která může rychle narůst a kód bude nepřehledný. Na ukázce z aplikace 5, oproti MVVM, obsahuje ViewModel jeden stav. Navíc má jednu metodu pro akce z View vrstvy. Ukázka je zjednodušená, ale lze z ní vidět, že pokud by zobrazení mělo hodně akcí, metoda onEvent může rychle narůst a pak je těžké hledat, co přesně dělá jaká metoda v jakém zobrazení, obzvlášť když každá akce volá privátní metodu ve ViewModelu. [18, 19]

Použití MVI může mít i dopad na výkon, jelikož ViewModel používá pouze jeden stav. Kdykoliv se změní nějaká hodnota ve stavu, stačí pouze jedna, je překresleno celé zobrazení.

2.2.5 Shrnutí

Pro použití s Jetpack Compose se mi výběr zůžil na MVVM a MVI, které se snadněji implementují, než MVC nebo MVP. Nakonec jsem zvolil MVI, hlavně kvůli použití jednoho stavu, který lze pak jednoduše sdílet mezi jednotlivými Composable funkcemi tvořící obrazovku, bez nutnosti, aby jednotlivé funkce měly spoustu parametrů. Oproti MVVM, kde bych měl několik stavů a jednotlivé zobrazení by tak musely mít více parametrů.

```
data class ForgotPasswordState(
    val email: String = "",
    val isEmailInvalid: Boolean = false,
    val isProcessing: Boolean = false,
    // další vlastnosti
)
sealed interface ForgotPasswordViewEvent {
    data class OnEmailChange(val email: String): ForgotPasswordViewEvent
    // další události
}
class ForgotPasswordViewModel (
    private val authRepository: AuthRepository,
): ViewModel() {
    private val _forgotPasswordState =
        MutableStateFlow(ForgotPasswordState())
    val loginState = _forgotPasswordState.asStateFlow()

    fun onEvent(event: ForgotPasswordViewEvent) {
        when (event) {
            is ForgotPasswordViewEvent.OnEmailChange -> {...}
            // zpracování ostatních událostí
        }
    }
    // další metody
}
```

Výpis kódu 5 Ukázka Jetpack Compose ViewModelu při použití MVI vzoru

2.3 Návrh obrazovek

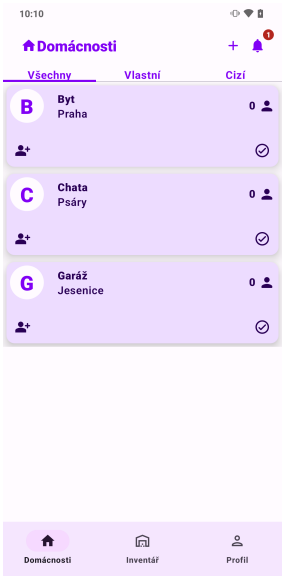
Při návrhu obrazovek jsem se snažil, aby aplikace byla co nejvíc podobná webové aplikaci, aby měl uživatel hladký přechod mezi webovou a mobilní aplikací. S tím, že jsem musel obrazovky optimalizovat pro malé zařízení.

Pro návrh, stejně tak i pro vývoj, jsem použil Android Studio, kde si lze jednotlivé obrazovky poskládat z jednotlivých komponent, ať už existujících nebo co jsem si vytvořil. Pomocí náhledů si pak tyto obrazovky zobrazit na zařízeních s různými velikostmi obrazovky a s předem definovanými daty lze náhledy i interaktivně spustit bez nutnosti použití emulátoru nebo reálného zařízení.

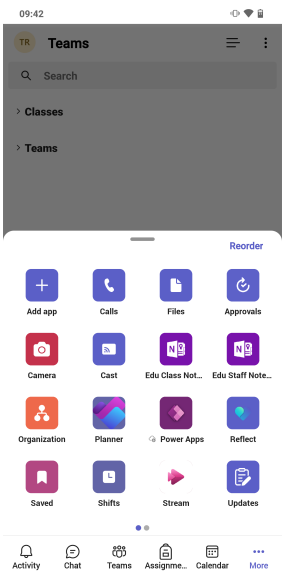
Při designování jsem se snažil dodržovat Material Design 3 [22]. Material Design 3 jsou doporučení, komponenty a nástroje od Googlu, které přináší nejlepší praktiky pro návrh uživatelského rozhraní.

2.3.1 Navigace

Hlavní navigace po přihlášení je řešena spodní lištou, kterou lze vidět na obrázku 2.4. Podle doporučení by se spodní lišta měla používat pro tři až pět destinací [23]. HouseKeeper je modulární a momentálně nabízí dva moduly, pro inventář a energii. S časem může modulů přibývat a spodní lišta už by nemusela být dostačující, protože by musela obsahovat více než pět destinací, což je proti doporučení. Potom by šlo spodní lištu upravit podle Microsoft Teams 2.5 tak, že by obsahovala tlačítko více, kde by byly schovány ostatní moduly. Uživatel by měl možnost si na spodní lištu dát moduly, které nejčastěji používá a ostatní by byly schovány.



Obrázek 2.4 Ukázka přehledu domácností z aplikace HouseKeeper



Obrázek 2.5 Ukázka spodní lišty z aplikace Microsoft Teams [24]

Základem každé obrazovky je „Scaffold“ komponenta, která dělí obrazovku na tři části, horní část, tělo obrazovky a spodní část, která může obsahovat například spodní navigační lištu a na některých obrazovkách není využita. Každá obrazovka, kromě té první po spuštění aplikace, obsahuje horní část, která se dělí na tři části. Úplně vlevo je navigace, která umožní uživateli vrátit se zpět na předchozí obrazovku. Následuje název obrazovky a v pravé části se nachází jednotlivé akce, které může uživatel provést, například na obrázku 2.4 je možno vidět akce pro vytvoření nové domácnosti a zobrazení pozvánek.

Kapitola 3
Implementace

V této kapilote se podíváme, jak jsem postupoval při implementaci samotné aplikace, abych splnil všechny požadavky. Co jsem použil za nástroje a knihovny, jak jsem strukturoval projekt a implementoval jednotlivé vrsty aplikace.

3.1 Použité nástroje

Pro ulehčení vývoje aplikace jsem používal nástroje, které lze rozdělit do těchto kategorií: správa kódu, psaní kódu, kontrola kódu a spuštění kódu.

3.1.1 Správa kódu

Pro správu kódu jsem používal Git společně se školním GitLabem, který slouží jako klient pro samotný Git.

Na začátku jsem si v GitLabu vytvořil jednotlivé issues podle požadavků aplikace. Následně jsem si je strukturoval, jelikož některé issues byly blokovány jinými, které musely být implementované dřív. Například issue pro správu lokací dává smysl implementovat až po issue pro správu domácností.

Z každého issue jsem vytvořil merge request, ve kterém jsem implementoval danou část, podle popisu v issue. Poté, co jsem doimplementoval vše potřebné, byl vždy kód i aplikace zkontrolovány mým vedoucím Ing. Zdeňkem Rybolou, Ph.D., který vždy zkontroloval jak kód, tak funkčnost aplikace. Aby ideálně ve vývojové větvi byla vždy funkční aplikace, která umí vše, co bylo v issues, které už jsou uzavřené.

3.1.1.1 GitLab pipeline

GitLab nabízí spoustu funkcí, jednou z nich je GitLab pipeline, která je velmi důležitou součástí vývoje. V mém případě jsem GitLab pipeline použil pro otestování, statickou analýzu a sestavení spustitelného souboru, který lze stáhnout do mobilu a aplikaci tím nainstalovat. Po každém nahrátí nového kódu na GitLab repozitář se pipeline spustí.

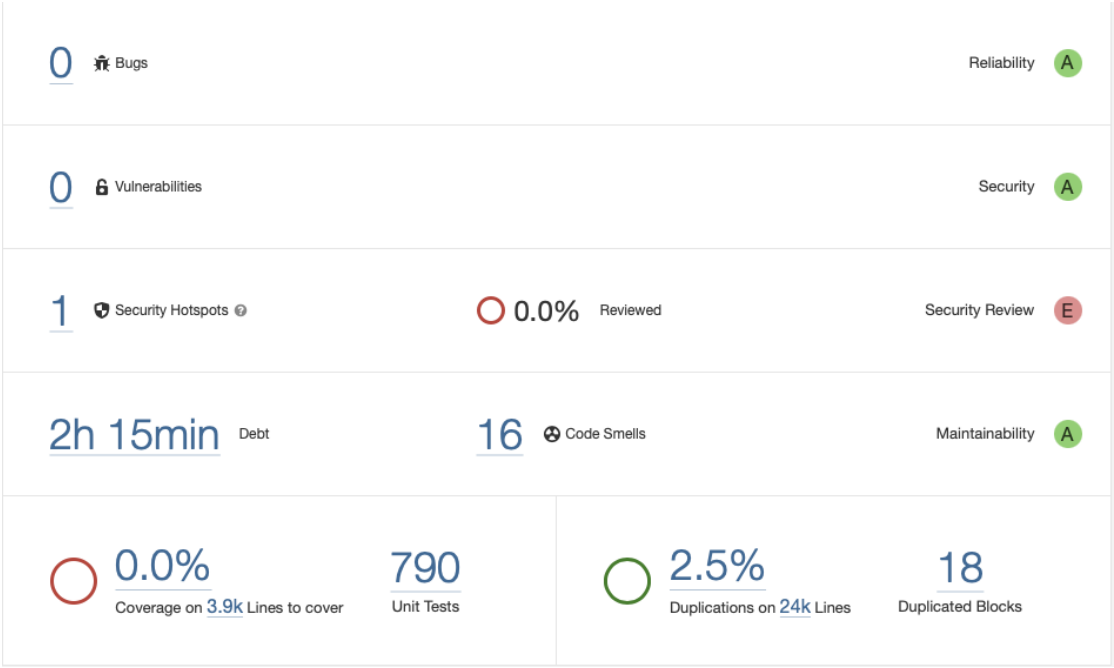
3.1.2 Psaní kódu

Pro samotné psaní aplikace jsem používal Android Studio, což je vývojové prostředí neboli IDE (Integrated Development Environment) nabízející vše potřebné pro vývoj mobilních aplikací pro OS Android. Od zvýrazňování, doplňování a formátování kódu, správu závislostí a překladů textů

do více jazyků, verzování pomocí integrovaného git klienta, debugging a profilování aplikace, spustění aplikace na emulátoru nebo reálném zařízení až po propojení s Google Play Console účtem pro jednodušší nahrání aplikace na Play store.

3.1.3 Kontrola kódu

Pro zajištění kvality kódu jsem využíval statickou analýzu pomocí SonarLint pluginu, integrovaného do Android Studia, napojeného na školní Sonarqube server. SonarLint mi vždy před nahráním kódu na GitLab hlásil tzv. „code smelly“, například: moc dlouhá funkce, funkce s moc parametry, nedosažitelný kód nebo nepoužité importy.



Obrázek 3.1 Ukázka statistiky kódu ze SonarQube

Na obrázku 3.1 je možno vidět výslednou statistiku ze SonarQube. Kód obsahuje jedno bezpečnostní ohnisko, což je způsobeno tím, že aplikace momentálně komunikuje s backendem pomocí HTTP a ne HTTPS (Hypertext Transfer Protocol Secure) a šestnáct code smellů. Z těchto šestnácti code smellů je dvanáct přítomných od vygenerování projektu a jsou to například nepoužité proměnné s předem definovanou barvou. Ostatní čtyři hlásí, že jsem překročil maximální povolený počet parametrů funkce, který je nastavený na sedm parametrů. U Composable funkcí, které Jetpack Compose používá pro vytváření UI, je to ale běžné a spíš by bylo lepší upravit nastavení SonarQube a povolit u těchto funkcí více parametrů.

Statistika také ukazuje 0% pokrytí kódu testy, i když existuje 790 testů. To je z důvodu, že se mi nepodařilo pomocí GitLab pipeliny a Sonar pluginu nahrát zprávu na SonarQube server, která tyto informace obsahuje.

3.1.4 Spuštění kódu

Pro sestavení a následné spuštění kódu na emulátoru nebo na reálném zařízení používám Gradle.

Gradle je sestavovací nástroj, který umožňuje jednoduchou správu závislostí a s použitím pluginů nabízí mnohem více. Například na ukázce 6 je vidět použití sonarqube pluginu s Gradlem,

```
sonar {
    properties {
        val url = providers.environmentVariable("URL").get()
        val projectName = providers.environmentVariable("NAME").get()
        val projectKey = providers.environmentVariable("KEY").get()
        val projectToken = providers.environmentVariable("TOKEN").get()
        property("sonar.host.url", url)
        property("sonar.projectName", projectName)
        property("sonar.projectKey", projectKey)
        property("sonar.login", projectToken)
    }
}
```

Výpis kódu 6 Zjednodušená ukázka použití Sonarqube pluginu s pomocí Gradlu

```
src ..... zdrojové kódy
└─ households ..... zdrojové kódy o domácnostech
   ├─ ui ..... obrazovky
   ├─ viewmodels ..... logika pro propojení obrazovek a repozitářů
   ├─ models ..... použité modely
   ├─ repositories ..... repozitáře
   └─ utilities ..... pomocné funkce a třídy
```

Obrázek 3.2 Zjednodušená struktura projektu podle funkčních požadavků

který se využívá v GitLab pipelině při statické analýze kódu pro nahrání výsledků na Sonarqube server.

Android Studio má v sobě Gradle dobře integrovaný. Sestavit nebo spustit projekt na emulátoru nebo spárovaném zařízení je pouze na jedno kliknutí.

3.2 Struktura projektu

Volba správné struktury pojektu je velmi důležitá pro budoucí rozšiřování nebo nějaká upravování. Správná struktura projektu umožňuje novým členům na projektu se rychleji seznámit s projektem, protože vše je na správném místě a není potřeba zdlouhavého hledání, kde se co nachází.

Při volbě struktury projektu jsem zohlednil dvě možnosti. Strukturovat projekt podle funkčních požadavků 3.2, podobně by byly strukturovány lokace, kategorie, produkty a kusy nebo podle vrstev a jejich účelu 3.3.

Nakonec jsem zvolil druhou možnost a strukturoval jsem projekt podle komponent a účelu s tím, že jsem si to v každé komponentě rozdělil podle jednotlivých funkčních požadavků.

```
src ..... zdrojové kódy
├─ ui ..... obrazovky
├─ viewmodels ..... logika pro propojení obrazovek a repozitářů
├─ models ..... použité modely
├─ repositories ..... repozitáře
└─ utilities ..... pomocné funkce a třídy
```

Obrázek 3.3 Zjednodušená struktura projektu podle vrstev a účelu

3.3 Implementace MVI

Jak už jsem psal při návrhu, pro implementaci aplikace jsem zvolil návrhový vzor MVI. V této sekci popíšu, jak mezi sebou jednotlivé komponenty komunikují a jak jsem implementoval jednotlivé komponenty, ze kterých se MVI vzor skládá.

3.3.1 Komunikace komponent

Na obrázkách 3.4, 3.5 je možné vidět třídy potřebné pro vytvoření lokace a na obrázku 3.6 už je pak možné vidět samotný proces vytváření lokace a jak mezi sebou jednotlivé třídy komunikují. View komponenta, v mém případě LocationCreateViews, obsahující obrazovku pro vytváření lokací, která se skládá z jednotlivých Composable funkcích tvořící danou obrazovku. Všechny funkce z LocationCreateViews si drží referenci na metodu onEvent z LocationCreateViewModel pomocí, které spolu tyto komponenty komunikují. Po provedení akce uživatelem nebo samotnou aplikací se zavolá metoda onEvent. Metoda přijímá jako parametr LocationCreateEvent, což je samotný Intent, který definuje všechny možné akce, které můžou být na obrazovce provedeny ať už uživatelem nebo samotnou aplikací. LocationCreateViewModel si tuto akci zpracuje a aktualizuje stav, který způsobí, že se obrazovka překreslí s novými informacemi. Mám to dělané tak, že metoda onEvent pouze zavolá privátní metodu se stejným názvem jako je daná akce, jen první písmeno je malé, která se stará o logiku dané akce. Pokud je potřeba komunikace s backendem, například u akce OnCreateButtonClick, tak ViewModel zavolá odpovídající metodu z LocationRepository v tomto případě metodu createLocation.

3.3.2 View komponenta

Většina operací prováděných s daty, které aplikace používá, jsou tzv. CRUD (Create, Read, Update, Delete) operace. Na základě těchto operací jsem rozdělil obrazovky jednotlivých funkčních požadavků. Každý požadavek, kromě správy uživatelů a atributů, se skládá ze tří obrazovek:

- přehledu (například přehled všech lokací),
- obrazovky pro vytvoření (například obrazovka pro vytváření lokací),
- obrazovky pro detail, ze které je možno i editovat nebo mazat.

Atributy lze vždy spravovat z detailu obrazovky zdroje, se kterým jsou spojeny. Například definice atributů kategorie lze spravovat z obrazovky s detailem kategorie.

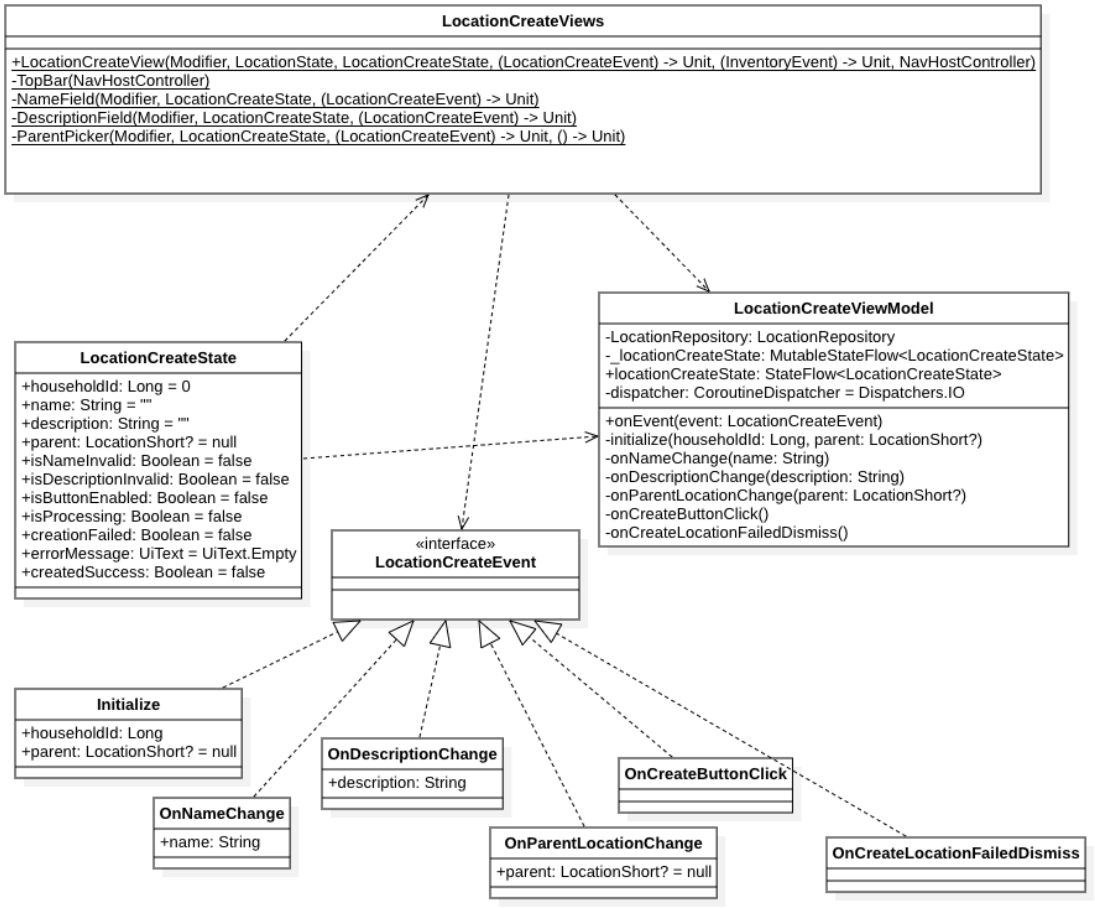
3.3.2.1 Navigace

Pro navigaci mezi jednotlivými obrazovkami používám nativní Androidí knihovnu Navigation Compose [25]. Základním blokem je Composable NavHost, ve kterém si lze vytvořit grafovou strukturu obrazovek. Poté pomocí NavHostControlleru lze mezi jednotlivými vrcholy grafu přecházet.

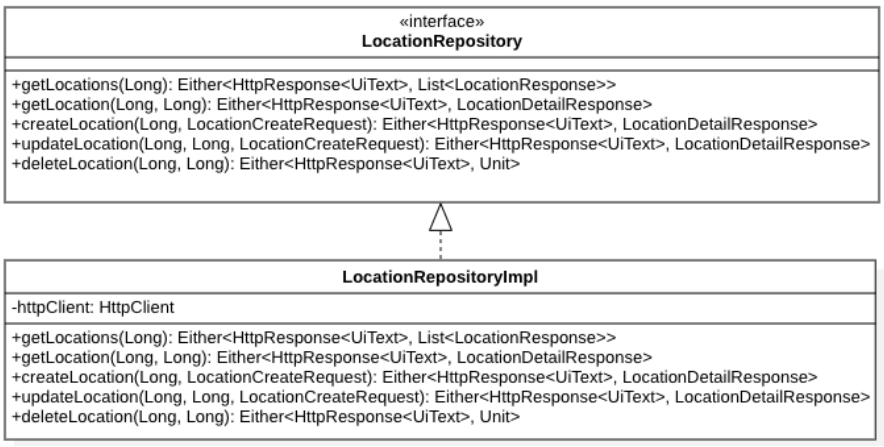
Moje grafová struktura pro navigaci se skládá ze dvou podgrafů:

- jeden, když uživatel ještě není autentizovaný,
- druhý, když je již autentizovaný.

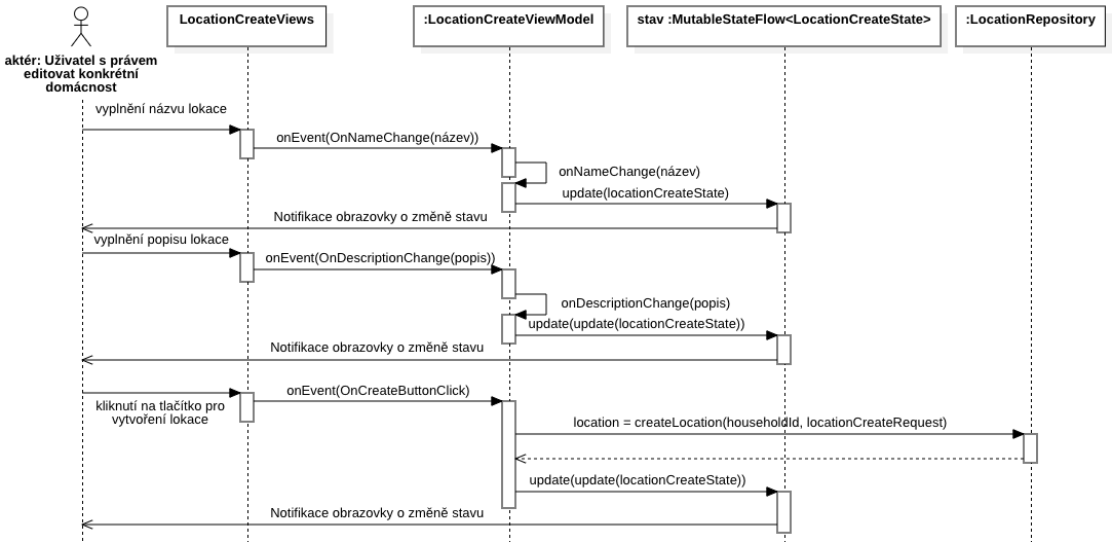
Na ukázce 7 lze vidět můj graf pro navigaci. Pro zpřehlednění jsem vynechal ze závorek za NavigationItem řetězce identifikující dané obrazovky, ukázku těchto řetězců lze vidět na 8. Z ukázky 8 lze vidět i použití povinných i nepovinných parametrů pro navigování. Tedy pro navigování na obrazovku s vytořením nové lokace je potřeba identifikátor domácnosti, a pokud uživatel chce, může vytvořit podlokaci přímo, pokud při navigování obrazovka obdrží i identifikátor nadřazené lokace.



Obrázek 3.4 Ukázka diagramu tříd pro vytváření lokací



Obrázek 3.5 Ukázka Model komponenty pro vytváření lokací



Obrázek 3.6 Ukázka procesu vytváření lokace

```
sealed class NavigationItem(val route: String) {
    data object Auth: NavigationItem() {
        data object Login: NavigationItem()
        data object ForgetPassword: NavigationItem()
        data object CreateAccount: NavigationItem()
    }
    data object LoggedIn: NavigationItem() {
        data object Home: NavigationItem()
        data object CreateHousehold: NavigationItem()
        data object DetailHousehold: NavigationItem()
        data object DetailLocation: NavigationItem()
        data object CreateLocation: NavigationItem()
        data object DetailCategory: NavigationItem()
        data object CreateCategory: NavigationItem()
        data object DetailProduct: NavigationItem()
        data object CreateProduct: NavigationItem()
        data object DetailItem: NavigationItem()
        data object CreateItem: NavigationItem()
    }
}
```

Výpis kódu 7 Ukázka grafové struktury navigace

```
val detailItemRouteId = "DETAIL_ITEM/{householdId}/{itemId}"
val createLocationRouteId = "CREATE_LOCATION/{householdId}?parentId={parentId}"
```

Výpis kódu 8 Ukázka identifikace jednotlivých vrcholů grafu

```
DataTable(
    columns = listOf(
        DataColumn { Text("Název produktu") },
        DataColumn { Text("Typ měření") },
        DataColumn { Text("Jednotka měření") },
        DataColumn { Text("kapacita") },
    )
) {
    products.forEach { product ->
        row {
            cell { Text("pomerančový džus") }
            cell { Text("objem") }
            cell { Text("litr") }
            cell { Text("2") }
        }
    }
}
```

Výpis kódu 9 Ukázka tvorby tabulek

3.3.2.2 Zobrazení tabulek

Jednotlivé kusy a produkty jsou zobrazeny v tabulce, stejně jak tomu je ve webové aplikaci.

Android nenabízí žádnou nativní knihovnu, kterou bych mohl použít pro zobrazování tabulek. Abych si tuto funkcionalitu nemusel implementovat sám, našel jsem knihovnu Compose Data Table [26], která nabízí Kotlin DSL (Domain Specific Language) pro jednoduchou tvorbu tabulek. Na ukázce 9 je možno vidět použití tohoto DSL.

3.3.2.3 Zobrazení lokací a kategorií

Lokace a kategorie lze hierarchicky skládat. Tedy každá lokace pod sebou může mít několik dalších lokací, stejné platí pro kategorie.

Pro zobrazení těchto hierarchií jsem se inspiroval u zobrazování souborových struktur, jelikož tvoří podobnou strukturu jako lokace nebo kategorie. K tomuto účelu jsem si našel knihovnu Bonsai [27], která mi zobrazení těchto struktur ulehčuje. Bonsai nabízí Kotlin DSL pro tvorbu těchto hierarchických struktur. Na ukázce 10 je možno vidět tvorbu těchto stromových struktur pomocí rekurzivního volání funkce TreeScope.Locations, kde pokud daná lokace nemá žádné potomky vytvoří se list jinak se vytvoří vnitřní vrchol a zavolá se stejná funce pro všechny potomky. Stejně jsou dělány i kategorie. Na obrázku 3.7 je možno vidět výsledné zobrazení přehledu lokací. Kliknutím na šipku se rozbalí daná lokace a objeví se všechny podlokace. Kliknutím na jméno lokace se otevře obrazovka s detailem lokace.

3.3.2.4 Dialogová okna

V případě, že uživatel provedenou změnu nemůže vidět hned nebo, že došlo k chybě, zobrazí se mu dialogové okno obsahující zprávu o tom, co se stalo. Toto okno obsahuje buď tlačítko pro opakování akce a tlačítko pro zrušení akce nebo jen tlačítko pro zrušení akce.

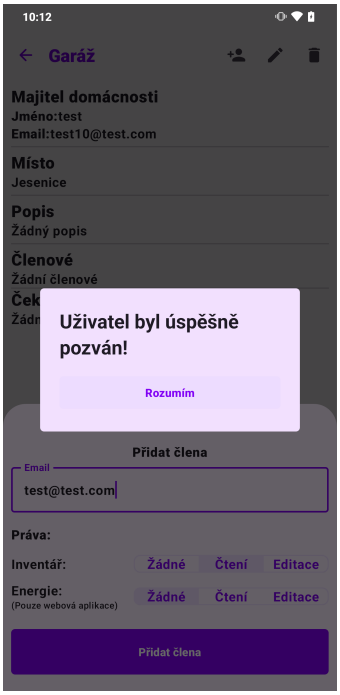
Na obrázku 3.8 lze vidět ukázku dialogového okna indikující, že se podařilo pozvat uživatele do domácnosti.

Na obrázku 3.9 lze vidět ukázku dialogového okna po selhání pozvání uživatele do domácnosti, kdy má uživatel možnost zkusit akci znovu, což zavře dialogové okno a uživatel může znovu

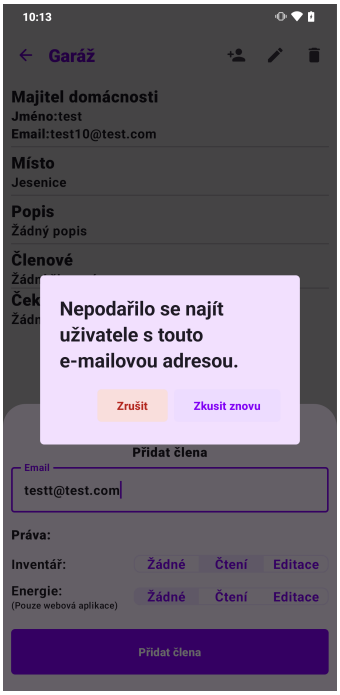
```
@Composable
fun createTree(
    locations: List<LocationResponse>
): Tree<LocationShort> {
    return Tree {
        locations.forEach {
            Locations(location = it)
        }
    }
}
@Composable
fun TreeScope.Locations(
    location: LocationResponse
) {
    if (location.children.isEmpty())
        Leaf(
            content = location.toLocationShort(),
            name = location.name
        )
    else
        Branch(
            content = location.toLocationShort(),
            name = location.name
        ) {
            location.children.forEach { Locations(location = it) }
        }
}
```

Výpis kódu 10 Ukázka tvorby stromových struktur

Obrázek 3.7 Ukázka zobrazení struktury lokací



Obrázek 3.8 Ukázka dialogového okna po úspěšném provedení akce



Obrázek 3.9 Ukázka dialogového okna po selhání provedení akce

kliknout na tlačítko pro pozvání, nebo má možnost kliknout na zrušit, což zavře dialogové okno i obrazovku pro pozvání uživatele.

3.3.3 Intent komponenta

Při implementaci návrhového vzoru MVI se v Android aplikacích využívá ViewModel, strukturovaný jako na ukázce 5, kde Intent komponenta je zabudovaná do samotného ViewModelu.

Samotné Intenty mám implementované pomocí rozhraní. Každá třída, která implementuje toto rozhraní, představuje jednu akci, kterou může uživatel provést na jednotlivých obrazovkách. ViewModely mají pouze jednu veřejnou metodu, kterou jednotlivé obrazovky využívají a kam posílají dané akce neboli Intenty. Tato metoda tyto akce pouze přijme a přepošle je do správné privátní metody, která tyto akce zpracuje a aktualizuje stav.

3.3.3.1 Životní cyklus ViewModelů

Pokud není specifikováno jinak, každý ViewModel je svázaný s vrcholem navigačního grafu, ve kterém byl instancován. Když uživatel přechází na jinou obrazovku, aplikace automaticky uvolní všechny zdroje, které alokovala a už nejsou potřeba a tedy i všechny ViewModely, které jsou svázané s danou obrazovkou neboli vrcholem grafu. Příkladem jsou ViewModely pro vytváření. Pokud chci vytvořit lokaci, stačí pokud je ViewModel pro vytváření lokací držen v paměti jen po dobu vytváření lokace.

Na druhou stranu někdy je nutné sdílet ViewModely mezi obrazovkami a nestačí, aby každá obrazovka měla vlastní instanci ViewModelu. Navigační knihovna [25], kterou jsem použil, sdílení umožňuje, a to díky tomu, že ViewModel nemusí být svázaný jenom s daným vrcholem grafu, ale i s určitým podgrafem nebo i s celým grafem. Příkladem, kde jsem využil instancování ViewModelů pro určitý podgraf, je ViewModel pro správu všech lokací dané domácnosti. Tento

ViewModel je potřeba držet v paměti po celou dobu, co je uživatel přihlášen, jelikož při vytváření lokace může uživatel specifikovat nadřazenou lokaci, kde vybere ze seznamu lokací, které už ale aplikace má v paměti a není potřeba dělat další dotaz na backend, podobně u vytváření produktu, kde je možnost specifikovat výchozí lokaci nebo u vytváření jednotlivých kusů kde, je potřeba specifikovat lokaci, kde jsou kusy uloženy.

3.3.3.2 Stránkování

Jednotlivé kusy a produkty přicházejí z backendu stránkované. Android nabízí nativní knihovnu Paging Compose, která se snaží stránkování ulehčit. V mém případě jsem tuto knihovnu nepoužil, protože by zanášela zbytečnou komplexitu do kódu a řešil jsem si stránkování sám bez knihovny. Pro ukládání jednotlivých stránek jsem použil asociativní pole, kde klíčem je číslo stránky a hodnotou jsou informace o dané stránce. Pokud uživatel změní stránku produktů, nejdříve se zkontroluje, zdali tato stránka už v asociativním poli není. Pokud je, uživateli se informace načtou hned, pokud není, musí se zavolat backend, který nám tyto informace poskytne, a až se načtou, zobrazí se uživateli.

3.3.4 Model komponenta

Model komponenta se skládá z repozitářů, které jsou odpovědné za posílání požadavků na backend a přijetí odpovědí a modelů, které používá REST API rozhraní definované na backendu. Tedy v případě vytvoření lokace si ViewModel zpracuje informace od uživatele a odešle požadavek pomocí repozitáře na backend, ten odpoví a repozitář odpověď předá zpět ViewModelu, který následně tuto odpověď zpracuje a aktualizuje stav, který způsobí překreslení obrazovky a zobrazení výsledku uživateli.

3.3.4.1 Ktor klient

Pro odesílání požadavků na backend používám knihovnu Ktor client, která nabízí Kotlin DSL pro odesílání REST API požadavků. Ktor je velmi rozšiřitelný díky velkému množství pluginů s možností vytvořit si vlastní. Na ukázce 11 je možno vidět konfiguraci Ktor klienta, kde pomocí metody install lze přidávat další pluginy. Například využívám plugin pro správu Cookies, kde poskytnu klientovi úložiště, kam má Cookies ukládat, a klient následně toto úložiště využívá. Když mu v odpovědi přijde v hlavičce Cookie, uloží si ho do tohoto úložiště a při posílaní požadavků ho pak přidává do hlavičky automaticky.

3.4 Vkládání závislostí

Vkládání závislostí je návrhový vzor, který usnadňuje instancování jednotlivých tříd, v mém případě hlavně ViewModelů. Tedy na místě, kde vytvářím ViewModel, nepotřebuji mít všechny závislosti, které daný ViewModel potřebuje, ale zavolám funkci, která tyto závislosti vyřeší za mě a pouze mi vrátí instanci ViewModelu.

K tomuto účelu používám knihovnu Hilt [28], která mi usnadňuje to, že nemusím vkládat závislosti sám. Základem je vytvoření modulu 12, kde definuji, jak vytvořit jednotlivé závislosti. Poté vytvořím ViewModel jako na ukázce 13. Pokud ho pak v kódu potřebuji, mohu na jakékoliv Composable obrazovce zavolat fukci `hiltViewModel<>()`, kde v špičatých závorkách specifikuji typ ViewModelu, který má být instancován. Hilt za mě vytvoří daný ViewModel se všemi závislostmi, které potřebuje a jsou definované v modulu.

```
fun provideHttpClient(
    cookiesStorage: CookiesStorage
): HttpClient = HttpClient(Android) {
    install(ContentNegotiation) {
        json(Json {})
    }
    install(DefaultRequest) {
        header(HttpHeaders.ContentType, ContentType.Application.Json)
        accept(ContentType.Application.Json)
        header(HttpHeaders.AcceptLanguage, Locale.current.language)
    }
    install(HttpCookies) {
        storage = cookiesStorage
    }
}
```

Výpis kódu 11 Ukázka konfigurace Ktor klienta

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {
    @Provides
    @Singleton
    fun provideHttpClient(
        cookiesStorage: CookiesStorage
    ): HttpClient = HttpClient(Android) {
        install(ContentNegotiation) {
            json()
        }
    }

    @Provides
    @Singleton
    fun provideAuthRepository(
        httpClient: HttpClient
    ): AuthRepository = AuthRepositoryImpl(httpClient)
}
```

Výpis kódu 12 Ukázka Hilt modulu

```
@HiltViewModel
class ForgotPasswordViewModel @Inject constructor(
    private val authRepository: AuthRepository,
): ViewModel() {
    // kód vynechán
}
```

Výpis kódu 13 Ukázka ViewModelu

Kapitola 4
Testování a nasazení

Testování je velmi důležitou částí vývojového cyklu a zajišťuje kvalitu daného softwaru. Čím více bude aplikace otestována, tím pravděpodobněji bude obsahovat méně chyb. Testuje se v každé části vývojového cyklu, ale v této kapitole se zaměřím na testovaní samotné aplikace a její funkčnosti.

4.1 Jednotkové testování

Základem testovaní jsou jednotkové testy. Jejich cílem je izolovat jednotlivé metody a ověřit jejich funkčnost, zdali se chovají tak, jak mají. Těchto testů by mělo být nejvíc, jelikož jsou nejrychlejší na napsání.

Pro psaní jednotkových testů jsem použil JUnit5 [29] knihovnu a pro mockování závislostí, které daná třída potřebuje, jsem použil knihovnu mockk [30].

Jednotkové testy jsou automatizované pomocí GitLab pipeliny. Při každém nahrání nového kódu na GitLab se všechny jednotkové testy spustí a vyhodnotí.

Na ukázce 14 je možno vidět vytvoření testovací třídy, která testuje RegisterViewModel. Nejdříve se připraví všechny potřebné závislosti. U AuthRepository je vidět použití mockk knihovny pro vytvoření mocku, jelikož chceme izolovat samotný ViewModel a testovat pouze jeho logiku a ne metody AuthRepository nebo jiných závislostí.

Na ukázce 15 už je samotný jednotkový test testující metodu registrování uživatele. Jednotkové testy se skládají ze tří částí a řídí se vzorem AAA (Arrange, Act, Assert).

První část Arrange říká, že v této části si máme připravit data a vše potřebné pro spuštění testované metody. V ukázce 15 si připravuji odpověď, kterou mi vrátí repozitář a pak mockuji, co se má stát při zavolání repozitáře. V druhé části Act, zavolám samotnou metodu, kterou testuji. A nakonec v třetí části Assert zkontroluji výsledky podle toho, jaké jsem očekával.

4.2 UI testování

Cílem UI testů je zajistit správnost uživatelských interakcí s aplikací a jejich odpovědí na tyto interakce. Příkladem je přihlašování, pokud uživatel zadá chybné přihlašovací údaje a klikne na tlačítko přihlásit se, aplikace by měla zobrazit dialog indikující chybné údaje a uživatele nepřihlásit.

UI testy jsou spouštěné buď na emulátoru nebo na reálném zařízení. Jelikož zařízeních s OS Android je velké množství, zaručit, že se aplikace bude chovat na většině zařízeních stejně, je velmi obtížné, obzvlášť pokud bych tyto testy prováděl ručně. Automatické UI testy mi to usnadňují díky tomu, že tyto testy jsou spoustěny na emulátoru nebo na reálném zařízení, které

```
class RegisterViewModelTest {
    private val authRepository: AuthRepository = mockk()
    private val userCache: UserCache = mockk()
    private val preferences = MockSharedPreference()
    private val testDispatcher = StandardTestDispatcher()
    private val registerViewModel = RegisterViewModel(
        authRepository, userCache, preferences, testDispatcher
    )
```

Výpis kódu 14 Ukázka přípravy testované třídy

```
@Test
fun `onRegisterButtonClick success`() {
    val userResponse = UserWithSensitiveInfoResponse.testUser()
    coEvery{authRepository.register(any())} returns Either.Right(userResponse)
    every {userCache.saveUser(userResponse)} returns Unit

    registerViewModel.onEvent(RegisterViewEvent.OnRegisterButtonClick)
    testDispatcher.scheduler.advanceUntilIdle()

    verify(exactly = 1){userCache.saveUser(userResponse)}
    val registerState = registerViewModel.registerState.value
    Assertions.assertTrue(registerState.registerSuccess)
    Assertions.assertFalse(registerState.isProcessing)
}
```

Výpis kódu 15 Ukázka jednotkového testu

já vyberu, čímž lze zaručit kompatibilitu s daným zařízením. Navíc stačí tyto testy napsat pouze jednou a pak už je jen spouštět na různých zařízeních, což při vývoji aplikace ušetří spoustu času.

Pro psaní UI testů jsem použil JUnit4 [31] knihovnu a pro mockování závislostí, které daná třída potřebuje, jsem použil knihovnu mockk [30].

Podobně jako jednotkové testy je možné automatizovat i UI testy a po každém nahrání kódu na GitLab tyto testy spustit a vyhodnotit. Bohužel s aktualním stavem použitého GitLabu není možné tyto testy automatizovat. K automatizování testů je zapotřebí GitLab Runner buď s ARM (Advanced RISC Machine) architekturou nebo s x86-64 architekturou s podporou hardwarové akcelerace. GitLab Runner, který používá moje pipeline, má architekturu x86-64, ale jelikož je tento Runner kontejnerizovaný, tak nemá hardwarovou akceleraci.

Na ukázce 16 je možno vidět vytvoření testovací třídy pro přihlašovací obrazovku a jeden test testující, zdali po zadaní emailu je tlačítko pro přihlášení zablokováno. Základem těchto testů je pravidlo, které nám umožní testovat Composable funkce. Na začátku každého testu je nutné specifikovat v metodě pravidla setContent jakou obrazovku chci testovat a s jakými daty. Poté už stačí pouze pomocí pravidla hledat komponenty, které se na obrazovce nacházejí a interagovat s nimi nebo kontrolovat jejich stav. Komponenty se hledají pomocí metody onNode, kde jako parametr specifikuji identifikátor dané komponenty, například to může být pomocí textu, který obsahuje. Tato metoda vrátí nalezenou komponentu, se kterou lze interagovat nebo kontrolovat její stav.

V ukázce 16 samotného testu nejdříve zkontroluji, zdali je tlačítko pro přihlášení vypnuté. Poté zkusím zadat nevalidní email a kontroluji, zdali je zobrazená chybová hláška, že email je nevalidní. Nakonec zadám správný email a zkontroluji, zdali chybová hláška zmizela a tlačítko pro přihlášení je pořád vypnuté.

4.3 Akceptační testování

Akceptační testovaní je testování, jehož cílem je ověřit, zdali software splňuje definované požadavky. Typicky je to poslední fáze testování a je to prováděno zpravidla klientem, pro kterého je software tvořen. V mém případě to je můj vedoucí Ing. Zdeněk Rybola, Ph.D., který toto testování provedl.

4.4 Nasazení

Jak jsem již uvedl v analýze rozhraní backendu, aktualní stav aplikace nesplňuje požadavky pro nahrání na Google Play. Kromě toho zatím není produkčně nasazený backend, momentálně je pouze za FIT VPN (Virtual Private Network) a komunikace probíhá prostřednictvím protokolu HTTP, tedy komunikace není šifrovaná.

Z toho důvodu lze aplikaci používat pouze po stáhnutí a nainstalování APK souboru a je nutné být připojený na FIT VPN.

```
@RunWith(AndroidJUnit4::class)
class LoginViewTest {
    @get:Rule
    val rule = createAndroidComposeRule<ComponentActivity>()
    private val emailFiled = hasText("Email")
    private val passwordField = hasText("Password")
    private val loginButton = hasText("Login")

    @Test
    fun `login email input button disabled`() {
        rule.setContent {
            val loginViewModel = viewModel<LoginViewModel>()
            LoginViews.LoginView(
                navController = rememberNavController(),
                loginState = loginViewModel.loginState.collectAsState().value,
                onLoginEvent = loginViewModel::onEvent
            )
        }
        val emailError = rule.activity.getString(R.string.invalid_email)
        rule.onNode(loginButton).assertIsNotEnabled()
        rule.onNode(emailFiled).performTextInput("test")
        rule.onNode(hasText(emailError)).assertExists()
        rule.onNode(emailFiled).performTextInput("@test.com")
        rule.onNode(emailFiled).assertTextContains("test@test.com")
        rule.onNode(hasText(emailError)).assertDoesNotExist()
        rule.onNode(loginButton).assertIsNotEnabled()
    }
```

Výpis kódu 16 Ukázka UI testu

Kapitola 5
Budoucnost aplikace

Momentálně by bylo ideální, kdyby se na backendu přidala funkce pro smazání uživatelových dat a jeho účtu, aby aplikace mohla být přidána na Google Play a mohl ji začít používat větší okruh lidí. Navíc instalovat aplikaci pomocí APK není moc příjemné, obzvlášť když při nějaké aktualizaci musím stahovat nové APK a znovu to nainstalovat a tento proces by se eliminoval s přidáním aplikace na Google Play.

Dále by bylo užitečné přidat do mobilní aplikace modul pro správu energií, aby se srovnala funkcionalita s webovou aplikací. Uživatelé, kteří by používali pouze jednu z platforem, by tak nepřicházeli o žádnou funkcionalitu, protože by obě platformy uměly to stejné a mohli by tak využívat platformu, která jim vyhovuje více.

Dále se nabízí přidat funkcionalitu, kterou nabízejí některé ostatní aplikace nebo, která by uživatelům zjednodušila a zpříjemnila používání aplikace.

5.1 Fotografie

Umožnit uživatelům přidávat k produktům fotografie, ať už přímo focené z aplikace nebo z galerie. Z fotografií by bylo možné vytěžit informace o produktu, například název, popis nebo kapacitu a uživatel by tak nemusel tyto informace zadávat ručně. Navíc by se například u jídla nebo pití daly vytěžit výživové údaje, které by se pak daly přidat jako atributy jednotlivých kusů.

5.2 Čárové nebo QR kódy

Dále se nabízí možnost přidat k produktům čárové nebo QR kódy, pomocí kterých by se pak dalo vyhledávat. Uživatel by tak mohl načíst čárový nebo QR kód z produktu a rychleji zjistit, na kterých lokacích a kolik kusů produktu se tam nachází. Podobně by se dal urychlit proces vyhledávaní jednotlivých kusů nacházejícíh se na nějaké lokaci, kdyby se pro danou lokaci dal vygenerovat čárový nebo QR kód. Uživatel by tedy přišel k lokaci, naskenoval by čárový nebo QR kód a v aplikaci by se mu zobrazily produkty, které se na dané lokaci nacházejí.

5.3 Upozornění

Aplikace momentálně podporuje upozornění o expiraci nebo překročení limitu kusů pomocí zaslání emailu. Přídání možnosti pro použití push notifikací by tak uživatelům usnadnilo správu těchto kusů. Uživatel by měl možnost si vybrat, zda chce dostávat oznámení pomocí emailu nebo pomocí push notifikací přímo na mobilní zařízení.

Závěr

Hlavním cílem práce bylo vytvořit mobilní aplikaci pro OS Android integrovanou do existujícího systému HouseKeeper. Výsledkem je aplikace napsaná v Jetpack Compose s použitím programovacího jazyka Kotlin a návrhového vzoru MVI, splňující všechny požadavky. Aplikace umožňuje uživateli přihlásit se a spravovat jeho domácnosti a jejich inventáře včetně správy lokací, kategorií a jejich atributů, produktů a jejich atributů a jednotlivých kusů a jejich atributů.

Při vytváření aplikace jsem se řídil klasickým softwarovým procesem, skládajícím se z analýzy, návrhu, implementace a testování.

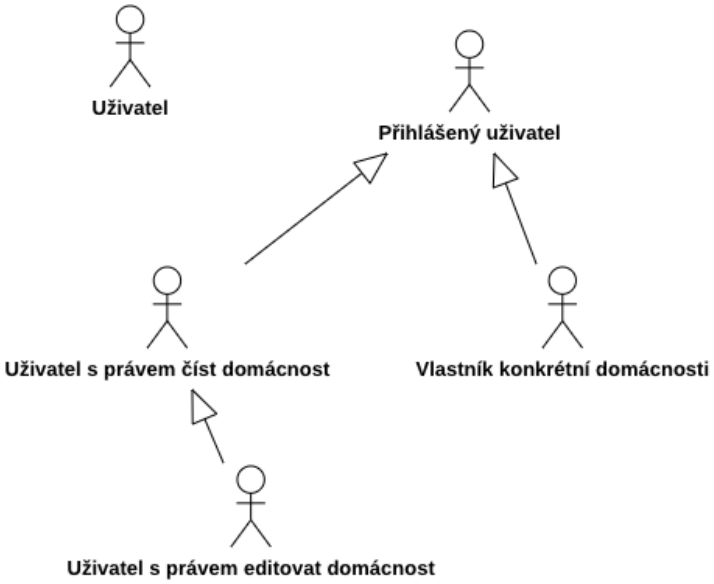
Nejdříve jsem provedl analýzu. Zanalyzoval jsem aktuální stav webové aplikace HouseKeeper, přičemž jsem se zaměřil na analýzu rozhraní a funkcionality backendové části. Zanalyzoval jsem také konkrétní funkční a nefunkční požadavky na mobilní aplikaci. Podle nich jsem následně provedl analýzu tří existujících aplikací zabývajících se stejnou problematikou.

Z analýzy mi vzešly požadavky, které má aplikace splňovat, podle kterých jsem následně aplikaci navrhl a implementoval.

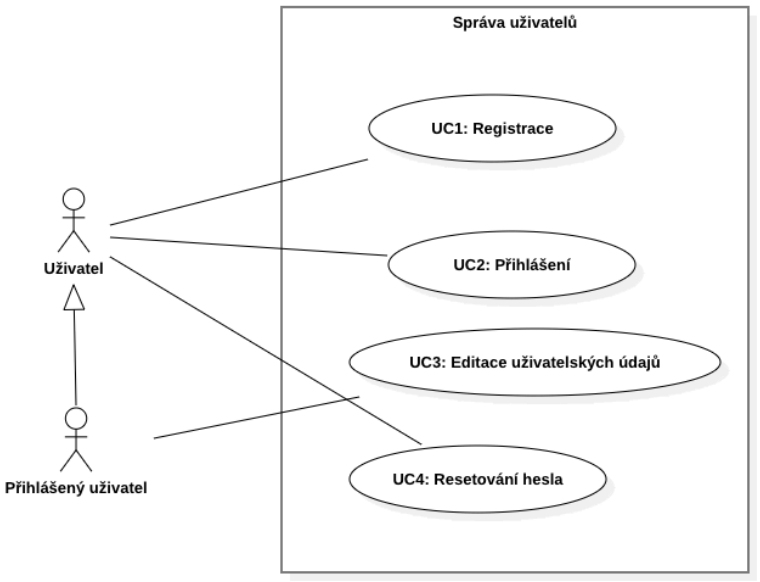
Nakonec jsem aplikaci zdokumentoval a otestoval, jak jednotkovými testy, a to přesněji Model a ViewModel komponenty, tak i View komponentu UI testy.

Příloha A

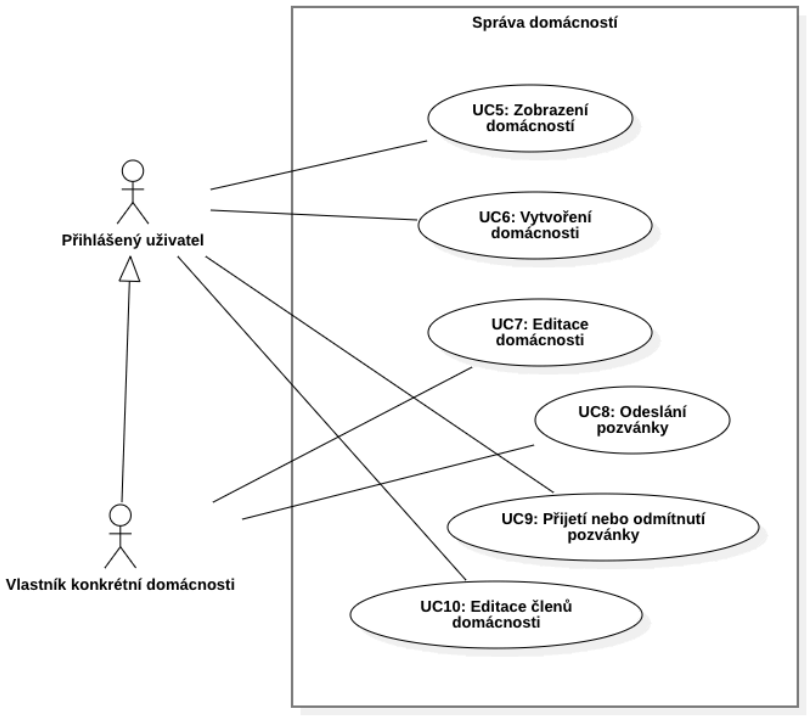
Diagramy případů užití



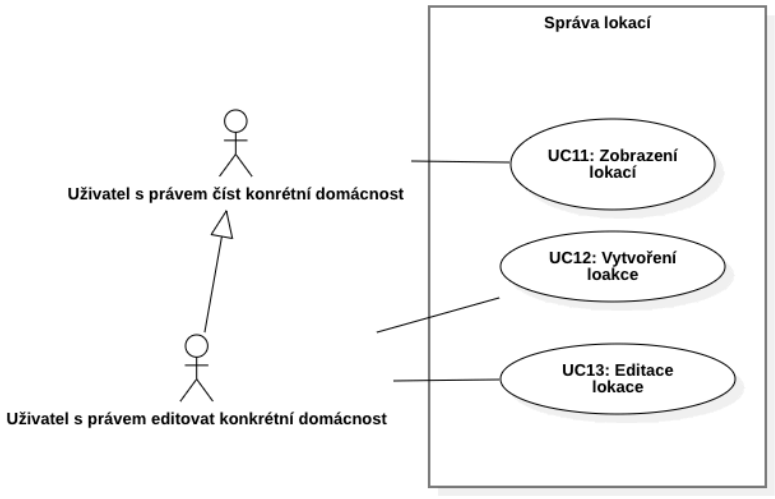
Obrázek A.1 Ukázka vztahů mezi aktéry



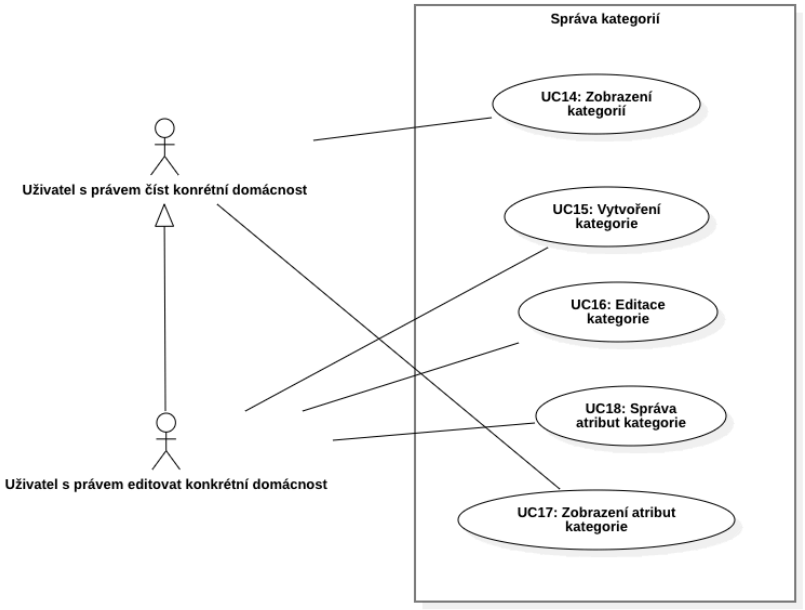
Obrázek A.2 Případy užití pro správu uživatelů



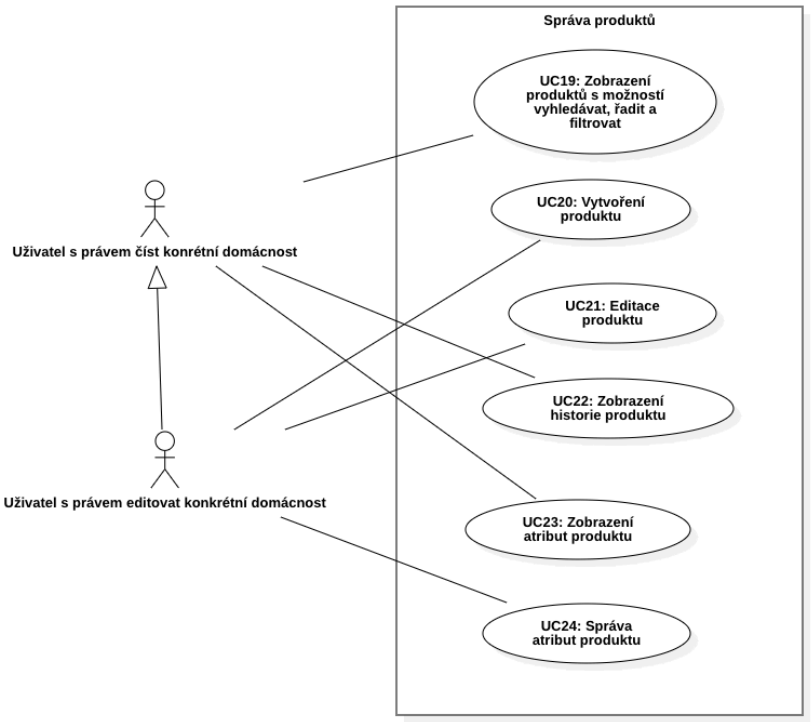
Obrázek A.3 Případy užití pro správu domácností



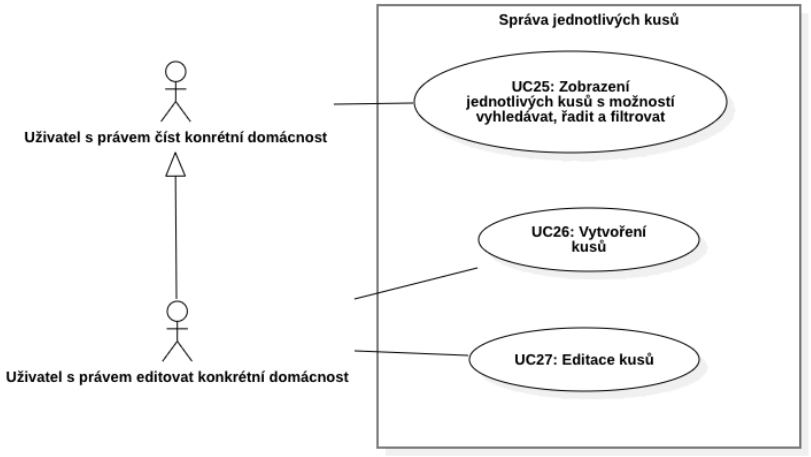
Obrázek A.4 Případy užití pro správu lokací



Obrázek A.5 Případy užití pro správu kategorií



Obrázek A.6 Případy užití pro správu produktů



Obrázek A.7 Případy užití pro správu jednotlivých kusů

Příloha B

Výsledky testování

Obrázek B.1 Výsledky UI testování

Test Summary

22	0	0	51.158s
tests	failures	skipped	duration

100%
successful

Packages **Classes**

Class	Tests	Failures	Skipped	Duration	Success rate
cz.cvut.fit.housekeeper_android.ExampleInstrumentedTest	1	0	0	0.012s	100%
cz.cvut.fit.housekeeper_android.ForgotPasswordViewTest	2	0	0	6.187s	100%
cz.cvut.fit.housekeeper_android.HouseholdDetailVIewTest	3	0	0	10.844s	100%
cz.cvut.fit.housekeeper_android.HouseholdsViewTest	6	0	0	13.059s	100%
cz.cvut.fit.housekeeper_android.LoginViewTest	5	0	0	8.904s	100%
cz.cvut.fit.housekeeper_android.ProfileViewTest	3	0	0	7.036s	100%
cz.cvut.fit.housekeeper_android.RegisterViewTest	2	0	0	5.116s	100%

Obrázek B.2 Výsledky jednotkového testování

Test Summary

790	0	0	4.811s
tests	failures	ignored	duration

100%
successful

Packages Classes

Package	Tests	Failures	Ignored	Duration	Success rate
cz.cvut.fit.housekeeper_android.repositories	102	0	0	2.573s	100%
cz.cvut.fit.housekeeper_android.utilities	14	0	0	0.036s	100%
cz.cvut.fit.housekeeper_android.viewmodels	48	0	0	0.569s	100%
cz.cvut.fit.housekeeper_android.viewmodels.auth	56	0	0	0.177s	100%
cz.cvut.fit.housekeeper_android.viewmodels.household	104	0	0	0.443s	100%
cz.cvut.fit.housekeeper_android.viewmodels.inventory	91	0	0	0.234s	100%
cz.cvut.fit.housekeeper_android.viewmodels.inventory.category	122	0	0	0.254s	100%
cz.cvut.fit.housekeeper_android.viewmodels.inventory.item	69	0	0	0.186s	100%
cz.cvut.fit.housekeeper_android.viewmodels.inventory.location	39	0	0	0.085s	100%
cz.cvut.fit.housekeeper_android.viewmodels.inventory.product	145	0	0	0.254s	100%

Obrázek B.3 Pokrytí kódu jednotkovými testy

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
cz.cvut.fit.housekeeper_android.viewmodels.household		98%		88%	41	338	9	589	9	194	6	45
cz.cvut.fit.housekeeper_android.viewmodels.inventory.product		97%		79%	88	435	7	739	8	218	4	45
cz.cvut.fit.housekeeper_android.viewmodels.auth		96%		82%	38	184	9	316	9	100	6	25
cz.cvut.fit.housekeeper_android.viewmodels.inventory.location		96%		82%	28	159	6	238	7	97	4	22
cz.cvut.fit.housekeeper_android.viewmodels.inventory.category		95%		78%	76	316	7	479	8	151	4	30
cz.cvut.fit.housekeeper_android.viewmodels.inventory.item		95%		78%	55	261	7	431	13	156	4	32
cz.cvut.fit.housekeeper_android.repositories		93%		75%	48	206	7	378	0	107	0	58
cz.cvut.fit.housekeeper_android.viewmodels		90%		80%	28	154	20	310	8	95	4	23
cz.cvut.fit.housekeeper_android.viewmodels.inventory		89%		77%	47	281	19	405	17	202	2	48
cz.cvut.fit.housekeeper_android.models.enums		80%		0%	32	78	31	83	30	76	1	29
cz.cvut.fit.housekeeper_android.models.responses		60%		75%	232	632	13	256	160	395	16	91
cz.cvut.fit.housekeeper_android.models.internal		46%		n/a	20	35	16	37	20	35	3	9
cz.cvut.fit.housekeeper_android.models.requests		35%		0%	297	427	8	150	147	277	10	75
cz.cvut.fit.housekeeper_android.utilities		31%		34%	83	115	147	184	56	77	19	26
cz.cvut.fit.housekeeper_android.ui.views		0%		0%	1,545	1,545	1,900	1,900	369	369	253	253
cz.cvut.fit.housekeeper_android.ui.views.inventory.product		0%		0%	1,087	1,087	1,662	1,662	323	323	257	257
cz.cvut.fit.housekeeper_android.ui.views.inventory.item		0%		0%	937	937	1,352	1,352	292	292	238	238
cz.cvut.fit.housekeeper_android.ui.views.household		0%		0%	955	955	1,317	1,317	233	233	193	193
cz.cvut.fit.housekeeper_android.ui.views.inventory.category		0%		0%	594	594	819	819	147	147	119	119
cz.cvut.fit.housekeeper_android.ui.views.auth		0%		0%	493	493	527	527	108	108	96	96
cz.cvut.fit.housekeeper_android.ui.views.inventory.location		0%		0%	444	444	563	563	108	108	86	86
cz.cvut.fit.housekeeper_android.ui.views.inventory		0%		0%	140	140	180	180	37	37	31	31
cz.cvut.fit.housekeeper_android.ui		0%		0%	60	60	94	94	40	40	27	27
cz.cvut.fit.housekeeper_android.ui.theme		0%		0%	37	37	42	42	12	12	4	4
cz.cvut.fit.housekeeper_android		0%		0%	39	39	58	58	21	21	9	9
hilt_aggregated_deps		0%		n/a	35	35	35	35	35	35	35	35
dagger.hilt.internal.aggregatedroot.codegen		0%		n/a	1	1	1	1	1	1	1	1
Total	117,757 of 161,347	27%	10,013 of 11,911	15%	7,480	9,988	8,856	13,145	2,218	3,906	1,432	1,907

Bibliografie

1. MASKAĽ, Nicolas Stefan. *Webová aplikace pro správu domácího inventáře*. 2024.
2. WORLD WIDE WEB CONSORTIUM. *Server-sent events* [online]. 2024. [cit. 2024-04-07]. Dostupné z: `https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events`.
3. MELNIKOV, Alexey; FETTE, Ian. *The WebSocket Protocol* [online]. 2011-12. [cit. 2024-04-07]. Request for Comments, č. 6455. Dostupné z DOI: `10.17487/RFC6455`.
4. FIELDING, Roy T.; RESCHKE, Julian. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [online]. RFC Editor, 2014-06 [cit. 2024-04-07]. Request for Comments, č. 7231. Dostupné z DOI: `10.17487/RFC7231`.
5. DUSSEAULT, Lisa M.; SNELL, James M. *PATCH Method for HTTP* [online]. RFC Editor, 2010-03 [cit. 2024-04-07]. Request for Comments, č. 5789. Dostupné z DOI: `10.17487/RFC5789`.
6. BRYAN, Paul C.; NOTTINGHAM, Mark. *JavaScript Object Notation (JSON) Patch* [online]. RFC Editor, 2013-04 [cit. 2024-04-07]. Request for Comments, č. 6902. Dostupné z DOI: `10.17487/RFC6902`.
7. GOOGLE. *User Data - Play Console Help* [online]. 2024. [cit. 2024-04-18]. Dostupné z: `https://support.google.com/googleplay/android-developer/answer/13316080?sjid=13967892336497456 48-EU#account_deletion`.
8. JENČO, Dávid. *Webová aplikace pro domácí evidenci spotřeby energií*. 2024.
9. HARDT, Dick. *The OAuth 2.0 Authorization Framework* [online]. RFC Editor, 2012-10 [cit. 2024-04-07]. Request for Comments, č. 6749. Dostupné z DOI: `10.17487/RFC6749`.
10. CHESTER SOFTWARE (XALTOS TECHNOLOGIES LTD). *Home Inventory, Food, Shopping* [online]. 2023. [cit. 2024-04-08]. Dostupné z: `https://play.google.com/store/apps/details?id=com.chestersw.homelist&hl=en_US`.
11. SUSAMP APPS. *My Stuff Organizer* [online]. 2018. [cit. 2024-04-08]. Dostupné z: `https://play.google.com/store/apps/details?id=com.ebizzapps.mystufforganizer&utm_source=website`.
12. HENIGE, Cameron. *HouseBook - Home Inventory* [online]. 2019. [cit. 2024-04-08]. Dostupné z: `https://play.google.com/store/apps/details?id=chenige.chkchk.wairz`.
13. OBJECT MANAGEMENT GROUP. *OMG Unified Modeling Language TM (OMG UML)* [online]. 2015. [cit. 2024-04-18]. Dostupné z: `https://www.omg.org/spec/UML/2.5/PDF`.
14. GOOGLE. *Jetpack Compose UI App Development Toolkit* [online]. [cit. 2024-04-18]. Dostupné z: `https://developer.android.com/develop/ui/compose`.

15. JETBRAINS S.R.O. *Compose Multiplatform UI Framework* [online]. 2024. [cit. 2024-04-18]. Dostupné z: https://www.jetbrains.com/lp/compose-multiplatform/.
16. GOOGLE. *Flutter* [online]. [cit. 2024-04-18]. Dostupné z: https://flutter.dev.
17. SILEN, Petri. *Clean Code Principles and Patterns, 2nd Edition*. LeanPub, 2024. ISBN 979-8373835732.
18. LACKNER, Philipp. *What Is the Best Architecture for Android Apps?* [online]. 2022. [cit. 2024-05-15]. Dostupné z: https://www.youtube.com/watch?v=cnU2zMnmmpg.
19. LACKNER, Philipp. *MVVM vs. MVI - Understand the Difference Once and for All* [online]. 2024. [cit. 2024-05-15]. Dostupné z: https://www.youtube.com/watch?v=b2z1jvD4VMQ.
20. LINARES-VÁSQUEZ, Mario. *MVC, MVVM, MV*, MV...What?* [online]. 2020. [cit. 2024-04-18]. Dostupné z: https://uniandes-se4ma.gitlab.io/books/chapter8/mvc-mvvm-mv-mvwhat.html.
21. MADUSHANKA, Krishan. *Android MVI (Model-View-Intent) Architecture-Example code* [online]. 2022. [cit. 2024-04-18]. Dostupné z: https://krishanmadushankadev.medium.com/android-mvi-model-view-intent-architecture-example-code-bc7dc8edb33.
22. GOOGLE. *Material Design* [online]. [cit. 2024-04-18]. Dostupné z: https://m3.material.io.
23. GOOGLE. *Navigation bar – Material Design 3* [online]. [cit. 2024-04-18]. Dostupné z: https://m3.material.io/components/navigation-bar/guidelines.
24. MICROSOFT CORPORATION. *Microsoft Teams* [online]. 2016. [cit. 2024-04-18]. Dostupné z: https://play.google.com/store/apps/details?id=com.microsoft.teams&hl=cs&gl=US.
25. GOOGLE. *Navigation with Compose* [online]. 2024. [cit. 2024-05-01]. Dostupné z: https://developer.android.com/develop/ui/compose/navigation.
26. PROCTOR, Sean. *An implementation of the Material Design data table for Compose.* [online]. 2024. [cit. 2024-05-01]. Dostupné z: https://github.com/sproctor/compose-data-table.
27. CAFÉ, Adriel. *A multiplatform tree view for Jetpack Compose* [online]. 2024. [cit. 2024-05-01]. Dostupné z: https://github.com/adrielcafe/bonsai.
28. GOOGLE. *Hilt* [online]. 2024. [cit. 2024-05-08]. Dostupné z: https://dagger.dev/hilt/.
29. JUNIT TEAM. *JUnit 5* [online]. 2024. [cit. 2024-05-01]. Dostupné z: https://junit.org/junit5/.
30. MOCKK. *Mockk* [online]. 2024. [cit. 2024-05-01]. Dostupné z: https://mockk.io.
31. JUNIT TEAM. *JUnit* [online]. 2021. [cit. 2024-05-09]. Dostupné z: https://junit.org/junit4/.

Obsah přiloženého média

`housekeeper-android` ..... zdrojový kód aplikace
`housekeeper.apk` ..... spustitelný soubor s aplikací (nutnost připojení na FIT VPN)
`thesis.pdf` ..... text práce ve formátu PDF