VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

MOBILNÍ APLIKACE: SUPERSNADNÝ SDÍLENÝ NÁKUPNÍ SEZNAM

MOBILE APP: SUPERSIMPLE SHARED SHOPPING LIST

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **PATRIK KRHOVSKÝ**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. ADAM HEROUT, PhD.**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/20637/2017/xkrhov02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií   Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Krhovský Patrik**
Obor: Informační technologie
Téma: **Mobilní aplikace: Supersnadný sdílený nákupní seznam**
Mobile App: SuperSimple Shared Shopping List
Kategorie: Uživatelská rozhraní

Pokyny:

1. Seznamte se s problematikou návrhu a vývoje mobilních aplikací; zaměřte se na platformu Android a vývoj pomocí React Native.
2. Vyhledejte a analyzujte existující aplikace řešící podobný problém.
3. Navrhněte a prototypujte způsob interakce s aplikací a jednotlivé prvky uživatelského rozhraní.
4. Navrhněte a implementujte řešenou aplikaci.
5. Testujte vytvořenou aplikaci na uživatelích a iterativně ji vylepšujte.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování; vytvořte plakátek a krátké video pro prezentování projektu.

Literatura:

- Steve Krug: Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability, ISBN-13: 978-0321965516
- Android Developers: https://developer.android.com/index.html
- Susan M. Weinschenk: 100 věcí, které by měl každý designér vědět o lidech, Computer Press, Brno 2012

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3, značné rozpracování bodu 4.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Herout Adam, prof. Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Cílem této práce je vytvořit mobilní aplikaci pro sdílení nejdůležitějších věcí v domácnosti, kanceláři nebo studentském bytě mezi uživateli a upozornit ostatní, pokud nějaká z věcí dochází. Sdílení těchto věcí mezi uživateli probíhá v reálném čase pomocí databáze Google Cloud Firestore. Aplikace je implementována v React Native a je dostupná pro iOS a Android zařízení. Pro chod mobilní aplikace bylo implementováno i REST API v Node.js, které se mimo jiné stará například o odesílání push notifikací. Práce se zaměřuje na návrh mobilní aplikace, její implementaci a průběžné testování. Výsledná aplikace je publikována v Apple App Store a Google Play. Uživatel by díky ní neměl nikdy zapomenout koupit to, co opravdu potřebuje.

Abstract

The aim of this thesis is to create a mobile application for sharing daily-needed stuff with your family or friends and let each other know if something runs out and buy it. For sharing this stuff between users in real time is used database Google Cloud Firestore. The application is implemented in React native and it is available for iOS and Android devices. REST API is implemented in Node.js where is saved data from the mobile application and sending push notifications to mobile devices. This work is focused on user interface design which was implemented and tested. The application is available in Apple App Store and Google Play. With this application, a user should never forget to buy what he really needs.

Klíčová slova

Vitalist, nákupní seznam, mobilní aplikace, Android, iOS, React Native, Redux, Redux Saga, Immutable.js, Node.js, Firebase, Cloud Firestore, Cloud Messaging, REST API, PostgreSQL

Keywords

Vitalist, grocery list, mobile application, Android, iOS, React Native, Redux, Redux Saga, Immutable.js, Node.js, Firebase, Cloud Firestore, Cloud Messaging, REST API, PostgreSQL

Citace

KRHOVSKÝ, Patrik. *Mobilní aplikace: Supersnadný sdílený nákupní seznam.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Adam Herout, PhD.

Mobilní aplikace: Supersnadný sdílený nákupní seznam

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Adama Herouta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Patrik Krhovský
17. května 2018

Poděkování

Děkuji panu prof. Ing. Adamu Heroutovi, Ph.D. za odborné vedení bakalářské práce a jeho cenné rady k vývoji aplikace. Chtěl bych poděkovat taky Ondrovi Vaculíkovi za odborné rady při návrhu uživatelského rozhraní. Děkuji taky všem, kteří se podíleli na testování aplikace a pomohli tak k jejímu vývoji.

Obsah

1 Úvod **2**

2 Motivace k vytvoření aplikace **3**
2.1 Odlišnost od nákupního seznamu 3
2.2 Průzkum existujících aplikací 3

3 Programování mobilních aplikací **6**
3.1 React Native 6
3.2 Návrhový vzor Flux 7
3.3 Řešení asynchonních akcí 11
3.4 Google Firebase 12
3.5 Eslint 14

4 Návrh aplikace **15**
4.1 Cílová skupina uživatelů 15
4.2 Návrh uživatelského rozhraní 15
4.3 Návrh komunikace mobilní aplikace 21
4.4 Návrh struktury databáze 23

5 Implementace **26**
5.1 Implementace serverové aplikace 26
5.2 Implementace mobilní aplikace 30

6 Testování **35**
6.1 Nultá verze aplikace 35
6.2 Testování prototypu verze 1.0 37
6.3 Testování aplikace verze 1.0 37
6.4 Zveřejnění aplikace verze 1.0 39

7 Závěr **40**

Literatura **41**

A Obsah přiloženého CD **43**

Kapitola 1

Úvod

Tato práce popisuje tvorbu aplikace Vitalist, jejíž cílem je poskytnutí služby k jednoduchému sdílení stavu nejdůležitějších věcí v domácnosti mezi všemi členy a v případě změny stavu dá ihned ostatním vědět, co je potřeba koupit. Uživatel tak může spravovat seznam věcí a to klidně ve více seznamech, například v případě více domácností, kterou může být třeba společné bydlení studentů na kolejích. Aplikace byla vytvářena v jazyce *JavaScript*, konkrétně pomocí knihovny *React Native*, díky které je aplikace dostupná pro dva nejrozšířenější mobilní systémy, iOS a Android. Pro plnou funkčnost mobilní aplikace byla vytvořena taky serverová aplikace ve webovém frameworku *node.js*, která zajišťuje správu dat nebo taky odesílání notifikací na mobilní zařízení a emailových zpráv. Pro propagaci aplikace byla vytvořena taky webová prezentace.

V následujících kapitolách je řešena kompletní tvorba mobilní aplikace, která začíná popsání motivace k vytvoření aplikace a tím zodpovězení proč a kdo by měl tuto aplikaci využívat. Je zde taky popsán průzkum existujících aplikací a jejich porovnání jak mezi sebou, tak s vyvíjenou aplikací. V další kapitole jsou popsány technologie, které mobilní aplikace využívá. Patří mezi ně například *React Native*, návrhový vzor *Flux* nebo nástroje Google Firebase. Následuje návrh uživatelského rozhraní, prototypu aplikace a samozřejmě návrh komunikace mobilní aplikace se serverovou aplikací, databází a službami Google Firebase. Předposlední kapitola se zabývá implementací jak mobilní aplikace, tak důležitých částí serverové aplikace. Poslední kapitola je věnována tomu nejdůležitějšímu a tím je testování. Je zde vidět postupný vývoj aplikace a změny v uživatelském rozhraní, které vychází z výsledků testování.

Kapitola 2

Motivace k vytvoření aplikace

Každá domácnost má pár věcí, bez kterých se nedokáže obejít. Pokud na ně člověk zapomene při nákupu, nezbývá nic jiného, než se vrátit co nejdříve do obchodu a koupit je. Nebylo by ale příjemnější před odchodem z obchodu otevřít aplikaci a během chvilky zkontrolovat, zda máme opravdu to nejdůležitější v košíku? Naopak, pokud nějaká z těchto věcí dochází, nebo dokonce již došla, stačí zapnout aplikaci, změnit stav položky a dát tak jednoduše všem členům domácnosti vědět, co nejdůležitějšího je potřeba dnes nakoupit. Přesně o toto se snaží aplikace Vitalist.

2.1 Odlišnost od nákupního seznamu

Nákupní seznam slouží k tomu, aby si člověk zapsal vše, co potřebuje koupit a postupně na něm odškrkával to, co již vložil do košíku. Ne každý je ale ochotný takové seznamy psát, nebo dokonce ani neví, co přesně na něj napsat, jelikož to zjistí až procházením v obchodě. Co ví ale určitě, je to, které věci nesmí zapomenout koupit a tak lze jednoduše zjistit, jestli již nejsou spotřebovány. Těchto věcí většinou není mnoho a jejich seznam se nemění tak často. Proč je tedy neevidovat na jednom místě a nebo jejich stav rovnou nesdílet mezi lidmi, kteří je v mé domácnosti používají taky? Uživatelé si tak dají navzájem vědět, pokud nějaká z těchto věcí dochází, došla nebo byla naopak koupena.

Na aplikaci Vitalist tedy nelze nahlížet jako na nákupní seznam, ale pouze jako na nějakou pomůcku, díky které by její uživatelé neměli zapomenout koupit to, co opravdu potřebují, čím může být například mléko, zubní pasta nebo mycí prostředek na nádobí.

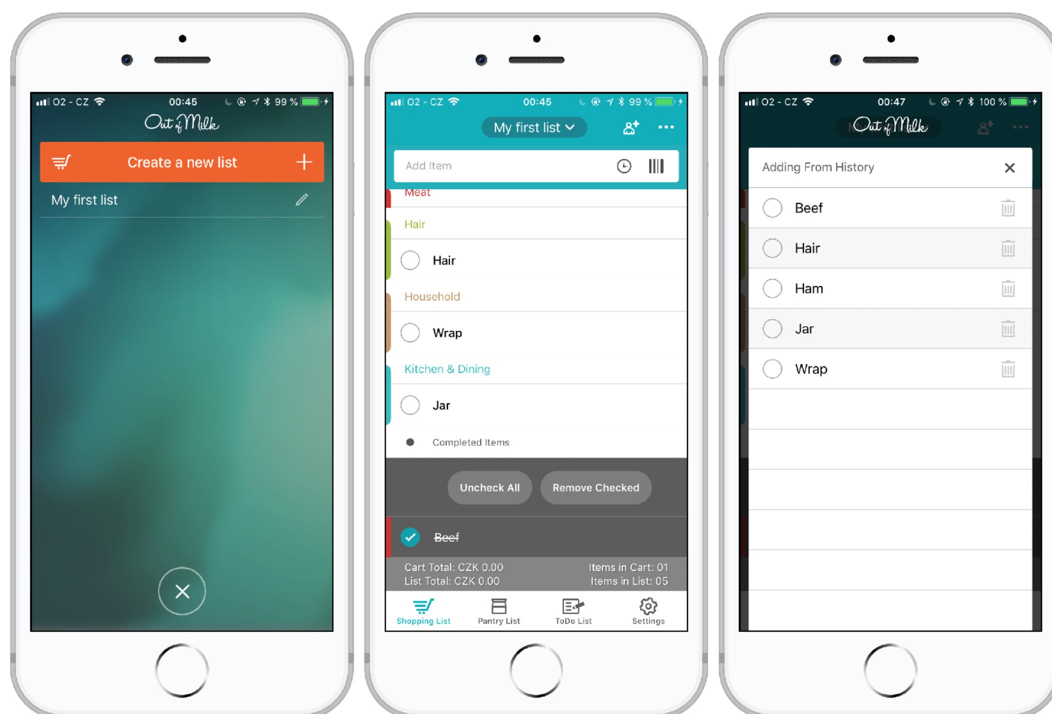
2.2 Průzkum existujících aplikací

Při průzkumu existujících aplikací nebyla nalezena žádná podobného směru, ale určitá dávka inspirace se dá čerpat právě od nákupních seznamů, kterých lze najít v Apple App Store[1] nebo Google Play[2] spoustu. Dá se taky předpokládat, že uživatel nákupního seznamu je potencionální uživatel této aplikace. Proto je potřeba mu práci s aplikací co nejvíce ulehčit a pokud možno co nejvíce přiblížit k práci s dřívějším nákupním seznamem.

[1] Obchod Apple App Store – https://appstore.com
[2] Obchod Google Play – https://play.google.com/store

2.2.1 Out of Milk

Aplikace *Out of Milk*[3] je jedna z nejpoužívanějších mobilních aplikací pro správu nákupního seznamu. Více než 5 milionů stažení na systému Android a hodnocení 4,6 hvězdičky hovoří samo za sebe. Aplikace se ale více než nákupní seznam tváří jako jednoduchý úkolovníček, který je dokonce součást aplikace. Aplikace řadí položky automaticky do kategorií, díky kterým je orientace v seznamu velice snadná a může pomoct i při nákupu. Uživatelé jistě ocení, že je aplikace dostupná pro systém Android i iOS. Lze si všimnout, že rozdíly v návrhu uživatelského rozhraní mezi Android a iOS verzí jsou zanedbatelné a i přitom je ovládání aplikace intuitivní pro uživatele jednoho nebo druhého systému. Samozřejmostí je sdílení seznamů. Obsahuje taky možnost zadání ceny jednotlivých položek, seznam často používaných položek nebo případné přidání položky naskenováním EAN kódu. Na obrázku 2.1 můžete vidět uživatelské rozhraní aplikace.



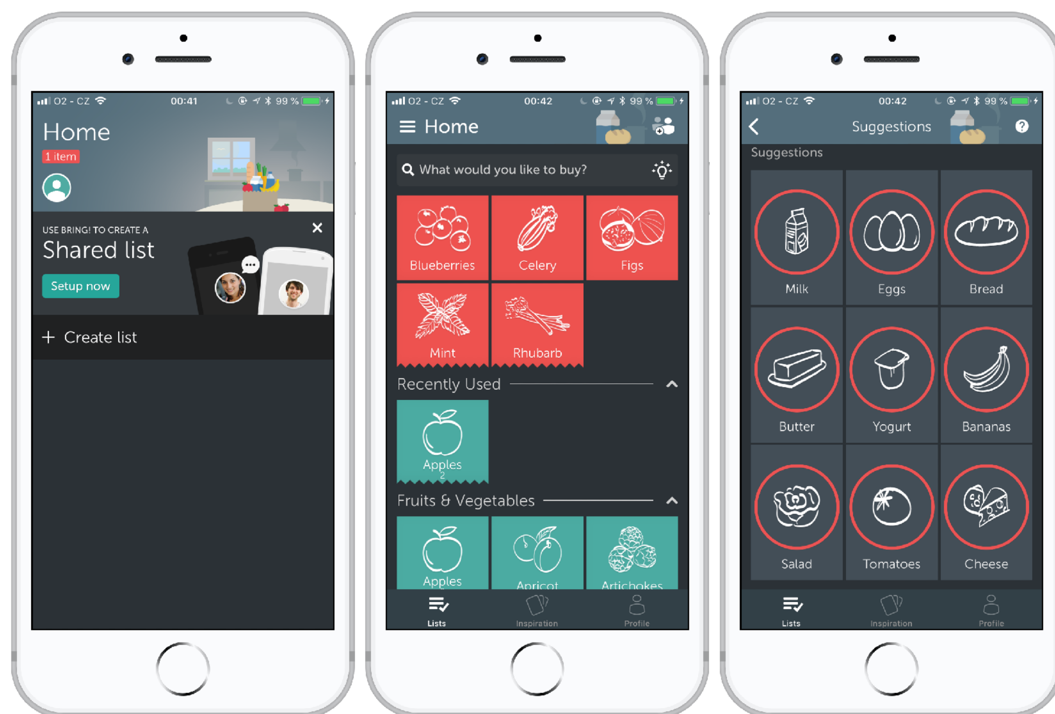
Obrázek 2.1: Uživatelské rozhraní aplikace *Out of Milk*

2.2.2 Bring!

Aplikace *Bring!*[4] má stažení o něco méně, přes 1 milion, ale její hodnocení je podobné, 4,5 hvězdičky. Při prvním otevření aplikace na mě udělalo velký dojem uživatelské rozhraní, které je dle mě velmi intuitivní, viz. obrázek 2.2. Aplikace uživatele sama vede k bezproblémovému průchodu funkcemi aplikace stručnými nápovědmi. Naopak od předešlé aplikace slouží výhradně pro správu nákupního seznamu. Položky řadí stejně jako předešlá aplikace do kategorií, navíc položka ale obsahuje ikonu, která na přehlednosti ještě přidává. Má předpřipraveno spoustu často používaných položek pro rychlejší ovládání, které si lze vybrat

[3] Aplikace Out of Milk – `https://outofmilk.com/`
[4] Aplikace Bring! – `https://getbring.com`

z jednotlivých kategorií a přidat je do seznamu pouhým kliknutím. Navíc se podle předešlého chování uživatele snaží nabídnout to, co by podle frekvence nakupování již mohlo chybět. Uživatel tak může tyto položky vložit na seznam a nebo zkontrolovat, jestli již náhodou nechybí. Tato funkce se trochu přibližuje aplikaci, jejíž vývojem se zabývá tato bakalářská práce. Aplikace je dostupná pro systémy iOS i Android a nechybí ani možnost sdílení seznamů.



Obrázek 2.2: Uživatelské rozhraní aplikace *Bring!*

Kapitola 3

Programování mobilních aplikací

V mobilních telefonech se dnes lze nejčastěji setkat se dvěma systémy, Android od firmy Google a iOS od firmy Apple. Pro tvorbu mobilní aplikace pro systém Android se nejčastěji využívá programovacího jazyku *Java*, naopak pro systém iOS je to jazyk *Objective C* nebo *Swift*. Systém Android používá zhruba 85 % mobilních zařízení, systém iOS zhruba 12 % [12]. Pokud je cílem ale vytvořit opravdu úspěšnou mobilní aplikaci, měla by být dostupná jak na systému Android, tak na systému iOS. Znamená to tedy, že pokud je potřeba naprogramovat stejnou aplikaci pro oba systémy, musí se programovat aplikace dvakrát, i když jsou rozdíly aplikace mezi systémy minimální. Jedna z možností, jak se této nepříjemnosti vyhnout, je popsána v této kapitole. Dále jsou zde taky popsány některé ze služeb Google Firebase, práce s daty v mobilní aplikaci, návrhový vzor *Flux* nebo princip řešení asynchronních akcí tak, aby neobtěžovaly uživatele a ten mohl dál nerušeně ovládat aplikaci.

3.1 React Native

React Native[1] [14][11] je cesta k vývoji mobilní aplikace s využitím jazyku *JavaScript* a knihovny *React*[2]. Stejně jako *React*, je i *React Native* vyvíjen firmou Facebook. *React Native* využívá nativních komponent, takže například místo HTML elementu `p` se využije komponenta `Text`, kterou *React Native* definuje. Ta bude poté nativně přeložena pro zařízení se systémem iOS jako `UIView` a pro zařízení se systémem Android jako `TextView`. Pomocí *React Native* tedy vytváříte plnohodnotnou nativní mobilní aplikaci, ne responzivní webovou aplikaci.

Ve vytvořené aplikaci běží dvě důležité vlákna, dále je budu nazývat jako `Main thread` a `JavaScript thread`.

- `Main thread` - je součástí každé mobilní aplikace. Stará se o zobrazení prvků uživatelského rozhraní a zpracování uživatelských gest

- `JavaScript thread` - zde běží prostředí, díky kterému lze zpracovat napsanou aplikaci v jazyce *JavaScript* v mobilních telefonech. Je to buď prostředí *JavaScriptCore*[3] nebo *V8*[4]

[1]Knihovna *React Native* – `https://facebook.github.io/react-native`

[2]Knihovna *React* je knihovna napsána v jazyce *JavaScript* pro vytváření komponent uživatelského rozhraní vyvíjená firmou Facebook. Více na: `https://reactjs.org/`

[3]JavaScriptCore – `https://developer.apple.com/documentation/javascriptcore`

[4]Google V8 – `https://code.google.com/p/v8/`

Tyto dvě vlákna spolu nikdy nekomunikují napřímo a navzájem se neblokují. O komunikaci mezi nima se stará prostředník (`bridge`), který je součástí knihovny *React Native* a dovoluje aplikaci, napsané v jazyce *JavaScript*, komunikovat s nativními komponenty. Komunikace mezi vlákny, o kterou se stará `bridge`, splňuje tyto vlastnosti:

- komunikace mezi vlákny probíhá asynchronně
- zprávy mezi vlákny jsou přenášeny v dávkách
- vlákna nemohou nikdy měnit stejné data zároveň

V knihovně *React Native* jsou vytvořeny například mobilní verze aplikací *Facebook*[5], *Instagram*[6], *Airbnb*[7] nebo *Skype*[8].

3.1.1 Rozdílné využití UI prvků pro Android a iOS

Jak bylo zmíněno v kapitole 3.1, například komponenta `Text` se vykreslí do nativních komponent *UIView* a *TextView*. Stejně tak může nově vytvořená komponenta vypadat různě, nebo využívat jiné prvky, podle systému, kde je aplikace spuštěna. A je to více než vhodné. Jeden z příkladu užití je naznačen v kapitole 4.2.3, konkrétně na obrázku 4.4, kde je vykreslena jiná komponenta po kliknutí na tlačítko v systému Android a iOS.

K vytvoření rozdílné komponenty stačí k názvu souboru přidat příponu `.android` nebo `.ios`. Knihovna *React Native* pak sama použije správnou komponentu. V případě menší části kódu, například rozdílného informačního textu nebo stylu, lze využít komponentu `Platform`, která je součástí knihovny. Díky takhle malé změně lze vytvořit uživatelské rozhraní, které splňuje pravidla obou systémů.

3.2 Návrhový vzor Flux

Flux[9] [13][19] je architektonický návrhový vzor pro vytváření uživatelských rozhraní zveřejněn firmou Facebook. Definuje jednosměrný datový tok v *React* komponentách. Skládá se z následujících částí:

- `Action` – pomocné metody, které usnadňují předání dat pro `Dispatcher`
- `Dispatcher` – obdrží akci a notifikuje o ní jednotlivá úložiště
- `Store` – úložiště, které udržují stav aplikace a mění ho podle změn o kterých je notifikuje `Dispatcher`
- `Controller View` – *React* komponenta, která obdrží stav aplikace a přenáší ho do dalších komponent jako parametr

[5]Aplikace Facebook – https://facebook.com
[6]Aplikace Instagram – https://instagram.com
[7]Aplikace Airbnb – https://airbnb.com
[8]Aplikace Skype – https://skype.com
[9]Návrhový vzor *Flux* – https://facebook.github.io/flux

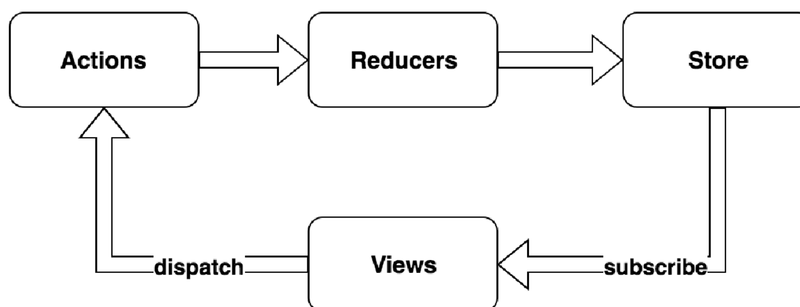
3.2.1 Redux

Redux[10] [7][16] je knihovna, která spravuje stav *JavaScript* aplikace, nejčastěji používaná společně s knihovnou *React*. Vychází z návrhového vzoru *Flux*, ale není jeho přesnou implementací. Tento datový tok popisuje schéma na obrázku 3.1. *Redux* má 3 základní principy:

- **Jeden zdroj pravdy** - stav celé aplikace (všech komponent) je uložen v jednom hlavním objektu. Díky tomu je snadnější ladění aplikace nebo uložení celého stavu do paměti a jeho pozdější načtení
- **Stav je pouze ke čtení** - komponenty nemohou měnit stav aplikace přímo. Jediná cesta ke změně stavu je vložením *redux* akce (pomocí funkce `dispatch`), která tento stav upraví
- **Používání čistých funkcí pro změny** - pro zapsání změny do stavu aplikace jsou používány čisté funkce[11]

3 stavební prvky, z kterých se *redux* skládá, jsou:

- `Store` - obsahuje stav celé aplikace a metody, díky kterým s ním lze pracovat:
  - `getState` – vrací aktuální stav aplikace
  - `dispatch` – odesílá akci ke zpracování. Jediná možnost jak upravit stav aplikace
  - `subscribe` – přihlášení posluchače (komponenty) k poslouchání změn stavu aplikace
  - `unsubscribe` – odhlášení posluchače
- `Action` - akce je *čistý JavaScript objekt*[12], který obsahuje zprávu a typ zprávy, pro rozlišení akce v části zvané `reducer`
- `Reducers` - jsou to čisté funkce, které podle typu a dat zprávy vrací nový stav aplikace. Stávající stav neupravují, ale vytváří nový upravený stav, jedná se o metodiku tzv. neměnných dat (viz. kapitola 3.2.3)



Obrázek 3.1: Schéma datového toku v knihovně Redux

[10] Knihovna *Redux* – https://redux.js.org/

[11] Čistá funkce (*pure function*) je taková funkce, která pokud má stále stejný vstup, vrací stále stejný výstup. Zároveň nemá žádné vedlejší efekty – nemění své argumenty

[12] *JavaScript objekt* – https://www.w3schools.com/js/js_objects.asp

3.2.2 Redux Saga

Redux Saga[13] [8] je knihovna, která se do aplikace implementuje jako *redux middleware*[14]. Je navržena tak, aby uživatelům ulehčila práci s asynchronními akcemi (například volání REST API nebo zápis do paměti telefonu) a to pomocí využití *JavaScript generátorů*[15], které dovolují psát asynchronní kód, který vypadá synchronně a je ho velmi snadné otestovat.

Redux saga nabízí spustu tzv. efektů, díky kterým lze spusit generátory. Mezi nejzajímavější patří například:

- `take` – generátor čeká, než bude zavolána redux akce, jejiž typ je vstupní parametr funkce
- `put` – odeslání redux akce ke zpracování
- `fork` – spustí generátor neblokovaně
- `race` – spustí neblokovaně zadané generátory a čeká na dokončení prvního z nich
- `all` – je schopný spustit více generátorů neblokovaně (provádí se najednou)

Díky tomuto přístupu lze například řešit optimisticky asynchronní akce, které jsou podrobněji popsány v kapitole 3.3. Další způsob využití je třeba pro tzv. *undo* akce, které lze řešit pomocí efektu `race` následovně:

```
function* saveItem(data) {
	// Zobrazeni UI undo elementu
	yield put(showUndo())

	// Optimisticke nahrazeni dat
	yield put(saveItem(data))

	// Cekani undo akci maximalne 5 sekund
	const { undo, save } = yield race({
		undo: take(action => action.type == 'UNDO'),
		save: call(delay, 5000),
	})

	// Schovani UI undo elementu
	yield put(hideUndo(())

	// Byla zavolana undo akce
	if (undo) {
		// Vraceni dat zpet do stavu pred ulozenim
		yield put(saveItem(oldData))
		return
```

[13]Knihovna *Redux Saga* – https://redux-saga.js.org/

[14]*Redux middleware* je navržen tak, aby funkce, které ho tvoří, byli provedeny před tím, než se zavolá hlavní metoda. Jinak řečeno, jsou funkce, které tvoří middleware, zavolány ještě před tím, než je *redux* akce odeslána ke zpracování do části `reducer`

[15]Funkce s hvězdičkou (generátor) se umí pozastavit, vrátit hodnotu a poté opět znovu spustit. Pozastaví se na místě, kde je klíčové slovo `yield`. Vhodí se například pro zpracování asynchroních akcí. Jsou součástí *ECMAScript 6*

```
        }

        // Ulozeni zmeny na API
        yield call(callApi, data)
}
```

Další příklady použití jsou uvedeny v kapitole 5.2.1, kde se řeší autorizační tok aplikací, a taky v kapitole 5.2.2, kde je popsán příklad volání API společně se spuštěním indikátorů aktivity.

3.2.3 Neměnná data

Neměnná (`immutable`) data nemohou být po jejich vytvoření změněna, což vede k mnohem jednoduššímu vývoji aplikací bez defenzivního kopírování. Neměnná data jsou zároveň perzistentní data[16] a díky tomu mohou knihovny, které implementují tyto datové struktury, provádět různé optimalizace při práci s daty, protože vědí, že data nemohou být změněna [18].

Jednou z implementací této techniky je knihovna *Immutable*[17] od firmy Facebook, která využívá například jako optimalizaci pro datovou strukturu `Map` techniku *HAMT*[18]. Díky této optimalizaci je přístup k datům v této struktuře podstatně rychlejší [6].

3.2.4 Normalizace dat

Normalizace dat slouží k rozdělení `JSON` zanořených objektů na jednoduché objekty podle předem daného schématu. Díky tomu je lze uložit a pracovat s nimi jednodušseji. Další výhodou je taky možnost držet všechny entity na jednom místě (například v redux stavu aplikace) a tím nedochází k jejich duplikaci. To znamená, že při případné úpravě entity ji stačí upravit pouze na tomto jednom místě a ta bude následně vykreslena ve všech komponentách, které ji používají. Příklad normalizace si lze ukázat na výsledeku vráceném z API, který vypadá následovně:

```
{
        id: 1,
        title: 'List',
        items: [{
                id: 1,
                title: 'Item 1',
        }, {
                id: 2,
                title: 'Item 2',
        }],
}
```

[16]Perzistentní datové struktury definují omezení, že všechny operace na ní prováděné, vrátí novější verzi datové struktury a zachovají původní strukturu neporušenou, namísto aktualizace původní struktury na stejném místě v paměti.

[17]Knihovna *Immutable.js* – https://facebook.github.io/immutable-js

[18]*HAMT* je implementace asociativního pole, které kombinuje vlastnosti hašovací tabulky a stromové struktury.

Jedná se o entitu seznamu, která obsahuje položky jako zanořený objekt. V případě, že by tuto položku využívalo více seznamů, musela by se editovat ve všech těchto seznamech. Normalizace tohoto objektu by mohla vypadat následovně:

```
{
	lists: [{
		id: 1,
		title: 'List',
		items: [1, 2],
	}],
	items: [{
		id: 1,
		title: 'Item 1',
	}, {
		id: 2,
		title: 'Item 2',
	}],
}
```

Následná editace poté proběhne pouze u jedné položky. Seznam udržuje pouze identifikátor položek, které obsahuje. Data položek jsou do seznamu přidány až v momentě, kdy jich je potřeba vykreslit v komponentě. Tyto data lze získat pomocí selektoru, které lze vytvořit pomocí knihovny *reselect*[19].

Knihovna *normalizr*[20] je jednou z implementací normalizace dat. Umožňuje definování entit, primárních klíčů nebo taky složitějších normalizačních pravidel.

3.3 Řešení asynchonních akcí

Volání asynchronní akce může zabrat nějaký čas a po tento čas by se určitě aplikace neměla tvářit tak, že se nic neděje – právě naopak. Aplikace musí po každém kliknutí ihned reagovat a jasně vyvolat akci, aby uživatel věděl, že se něco děje. Způsoby, jak vyřešit asynchronní akce jsou dva:

- zobrazení indikátoru aktivity a po tuto dobu omezit uživateli ovládání
- provést tzv. optimistickou akci, nechat uživatele dále pracovat a upozornit ho pouze v případě problému [17]

Optimistické uživatelské prostředí je dnes základem dobré aplikace. Jedná se vlastně o to, že se akce v rámci klientské aplikace provede ještě před tím, než je znám výsledek asynchronní akce. Například v případě editace názvu položky, je po odeslání této akce ihned vykonána změna na datech v klientské aplikaci a až poté odesílána na serverovou aplikaci. Uživatel tak ihned vidí, co se změnilo a může s aplikací pracovat nerušeně dál. V případě problému s odesláním na serverovou aplikaci by byl informován.

[19] Knihovna *reselect* – https://github.com/reduxjs/reselect
[20] Knihovna *normalizr* – https://github.com/paularmstrong/normalizr

3.4 Google Firebase

Firebase[21] je vývojářská platforma od firmy Google, která pomáhá vývojářům k rychlému vývoji mobilních a webových aplikací, bez budování zbytečné infrastruktury. Nabízí služby z oblasti vývoje aplikace, zvyšování kvality aplikace a rozvoje podnikání [4].

3.4.1 Služba Cloud Firestore

Cloud Firestore je jedna ze služeb, kterou platforma Firebase nabízí. Jedná se o flexibilní a škálovatelnou *NoSQL* dokumentovou databázi, která nabízí možnost jednoduchého uložení, synchronizování a dotazování se nad daty [1]. Nabízí taky možnost offline režimu pro mobilní a webové aplikace. Služba byla uvedena na trh ke konci roku 2017 a je zatím dostupná pouze ve verzi beta. Platí se za počet operací výkonaných nad databází.

Cloud Firestore nebo Realtime database?

Mnohem déle je na trhu služba *Realtime database*. Narozdíl od databáze *Cloud Firestore* ukládá *Realtime database* data ve formátu `JSON` a nenabízí tedy složitější dotazování se nad daty jako *Cloud Firestore*. Databáze není navíc ani škálovatelná, takže při překročení maximálních limitů je potřeba vytvořit novou instanci a škálovat tak manuálně. Narozdíl od *Cloud Firestore* se zde ale platí za šířku pásma a kapacitu databáze, což může být ve spouště aplikací výhoda, viz. 3.4.1.

Způsob strukturování dat

Databáze obsahuje kolekce, které si lze představit jako tabulky. Jednotlivým řádkům v těchto tabulkách se říká dokumenty. Data v dokumentech je možné různě strukturovat, aby manipulaci s nimi byla pokud možno co nejsnadnější. Základní způsoby jsou:

- **Vnořené data v dokumentech** – hodí se pro neměnné seznamy dat. Například pole uživatelů využívající seznam. Je potřeba brát v potaz to, že se přichází o možnost dotazování se nad daty, které jsou vnořené uvnitř dokumentů
- **Subkolekce** – hodí se pro velké množství dat, které patří jen k jednomu dokumentu. Dokument seznamu pak obsahuje například kolekci položek
- **Kořenové kolekce** – ukládání dat podobně jako v relační databázi, podle entit a odkazování se jednoznačnými identifikátory

Při návrhu struktury a složitém dotazování nad daty je potřeba dát si pozor na vytváření indexů a jejich omezení[22].

Pravidla

Pro zabezpečení databáze je potřeba nastavit pravidla. Každý dotaz nad databází obsahuje identifikátor uživatele, díky kterému lze například omezit uživatele jen na data, kterých je vlastník. Pravidla lze i různě kombinovat a využívat k tomu i data z jiných kolekcí. Jedním z příkladů může být přístup pouze k těm položkám, které patří k seznamům, kterých je uživatel vlastník nebo je má nasdíleny. Pravidla by poté vypadaly následovně:

[21] Platforma *Firebase* – `http://firebase.google.com/`

[22] Vytváření indexů a jejich omezení ve službě *Cloud Firestore* – `https://firebase.google.com/docs/firestore/query-data/indexing`

```
// Pravidlo pro dokumenty v~kolekci items
match /items/{item} {
	// Povoli cteni pouze polozek, ktere patri do seznamu,
	// kde je prihlaseny uzivatel vlastnik nebo ma seznam nasdileny
	allow read:
	if request.auth.uid == get(
		/databases/$(database)/documents/lists/$(resource.data.listId)
	).data.createdById
	|| get(
		/databases/$(database)/documents/lists/$(resource.data.listId)
	).data.collaborators[request.auth.uid] != null;
}

// Pravidlo pro dukumenty v~kolekci lists
match /lists/{list} {
	// Povoli cteni pouze seznamu, kde je prihlaseny uzivatel vlastnik
	// nebo ma tento seznam nasdileny
	allow read:
	if request.auth.uid == resource.data.createdById
	|| resource.data.collaborators[request.auth.uid] != null;
}
```

Kvóty

Kvóty ve verzi, která je dostupná zdarma, jsou 50 000 čtení, 20 000 zápisů a 20 000 odstranění[23]. U čtení si je potřeba dát pozor, pokud jsou v rámci jednoho uživatele aplikace potřeba stahovat velká data. V případě, že je uživatel schopný po zapnutí aplikace synchronizovat například 100 záznamů a lze předpokládat, že ji nezapne jen jednou nebo jen na jednom zařízení, se dá velice rychle dostat k limitu. Je potřeba nad takovými věcmi přemýšlet již při návrhu komunikace mobilní aplikace s ostatními službami. Jedno z řešení je využít *Realtime database* nebo využívat z databáze *Cloud Firestore* pouze ta data, která byla změněna po synchronizaci dat v aplikace. Tato synchronizace může probíhat se serverou aplikací (namísto databáze *Cloud Firestore*) například po zapnutí aplikace. Tento způsob komunikace je popsán v kapitole 4.3.

3.4.2 Služba Firebase Cloud Messaging

Firebase Cloud Messaging (dále jen *FCM*) je služba, která řeší spolehlivé zasílání zpráv na různé platformy a to navíc zcela zdarma. Pomocí služby *FCM* lze upozornit uživatele, že jsou k dispozici nějaká nová data a to zcela bez spuštění aplikace, jelikož proces přijímaní zpráv funguje na pozadí[24] [2]. Těmto zprávám se taky někdy říká *push notifikace*. Zprávy jsou odesílány na zařízení pomocí identifikačního čísla zařízení. V případě hromadných zpráv lze využít například rozdělení uživatelů podle systému, na kterém běží aplikace.

[23] Podrobné popsání kvót u služby *Cloud Firestore* – https://firebase.google.com/docs/firestore/quotas

[24] Pouze v případě, kdy je odesílána zpráva typu `notification message`

Typy zpráv

Pomocí služby *FCM* lze zaslat 2 typy zpráv:

- `notification message` – zpráva je automaticky zobrazena koncovému uživateli jménem mobilní aplikace. Tyto zprávy zpracovává a zobrazuje systém mobilního zařízení
- `data message` – zpracování těchto zpráv je na klientské aplikaci a to pouze v případě, že je aplikace v popředí. Nelze tedy zobrazit tento typ zprávy, pokud je aplikace vypnutá nebo pracuje v pozadí

3.5 Eslint

Eslint[25] je nástroj, který staticky analyzuje kód a udržuje jeho konzistenci v rámci celého projektu podle jeho nastavení. Upravuje jak styl, tak upozorňuje na chyby, které kód obsahuje, ještě před jeho spuštěním. Může to být například nepoužití definované proměnné nebo nepoužitá knihovna, která bylo do kódu vložena. *JavaScript*, jako netypovaný jazyk, je na tyto chyby obzvlášť náchylný. Nastavení pravidel je na vývojáři. Pro účely tohoto projektu bylo využito volně dostupné nastavení od firmy STRV[26] pro klientskou i serverovou aplikaci.

[25] *Eslint* – `https://eslint.org/docs/about/`
[26] STRV eslint config – `https://github.com/strvcom/eslint-config-javascript`

Kapitola 4

Návrh aplikace

Tato kapitola se zabývá postupným návrhem mobilní aplikace Vitalist. Definuje se cílová skupina uživatelů a následně jsou popsány jednotlivé scény navrhnutého uživatelského rozhraní. Následuje návrh komunikace mobilní a serverové aplikace s dalšími službami, popis možností a zdůvodnění konečného řešení. Na konci kapitoly je navržena struktura databáze podle dat, které definuje návrh uživatelského rozhraní. Zabývá se rozdílem dat uložené v relační databázi a v databázi pro synchronizaci v reálném čase.

4.1 Cílová skupina uživatelů

Aplikace je primárně určena do domácností, kde se musí počítat se všemi věkovými skupinami. S aplikací může pracovat jak rodič, tak jeho dítě. Rodič je ten, který seznam vytváří a upravuje, naopak po dítěti chce vědět pouze to, aby dalo vědět, pokud došla nějaká z položek v seznamu. Například že dopilo mléko. Aplikace ale může najít své využití i jinde, a to třeba u studentů nebo kolegů v práci, pokud například sdílí jednu kancelář a mají zde nějaké společné věci. Lze taky uvažovat o možném využití v komerční sféře, například v restauracích nebo hotelech v rámci personálu. Aplikace má nyní anglickou mutaci, do budoucna se ale počítá s rozšířením o další mutace.

4.2 Návrh uživatelského rozhraní

Jak bylo zmíněno v úvodu práce, cílem je vytvořit aplikaci pro systém iOS i Android a k tomu je přizpůsoben taky návrh. Kapitola 3.1.1 popisuje možnost rozdílného využití komponent podle systému, kde je aplikace spuštěna. Lze tedy využít nebo vytvořit rozdílné komponenty pro jednotlivé systémy tak, aby uživatel aplikace na daném systému neměl problém s pochopením ovládání. Naopak by měl po spuštění aplikaci ihned vědět, co mu nabízí a jak ji ovládat [15]. Základní pravidla tvorby uživatelského rozhraní pro systém iOS definují *Human Interface Guidelines*[1] od firmy Apple. Pro systém Android jsou to naopak *Android Design Guidelines*[2] od firmy Google. Kompromis, mezi těmito pravidly a ukázky využití jednotlivých komponent, definuje *Material Design*[3] od firmy Google, z kterých se

[1] Human Interface Guidelines od firmy Apple – https://developer.apple.com/ios/human-interface-guidelines

[2] Android Design Guidelines – https://developer.android.com/design/

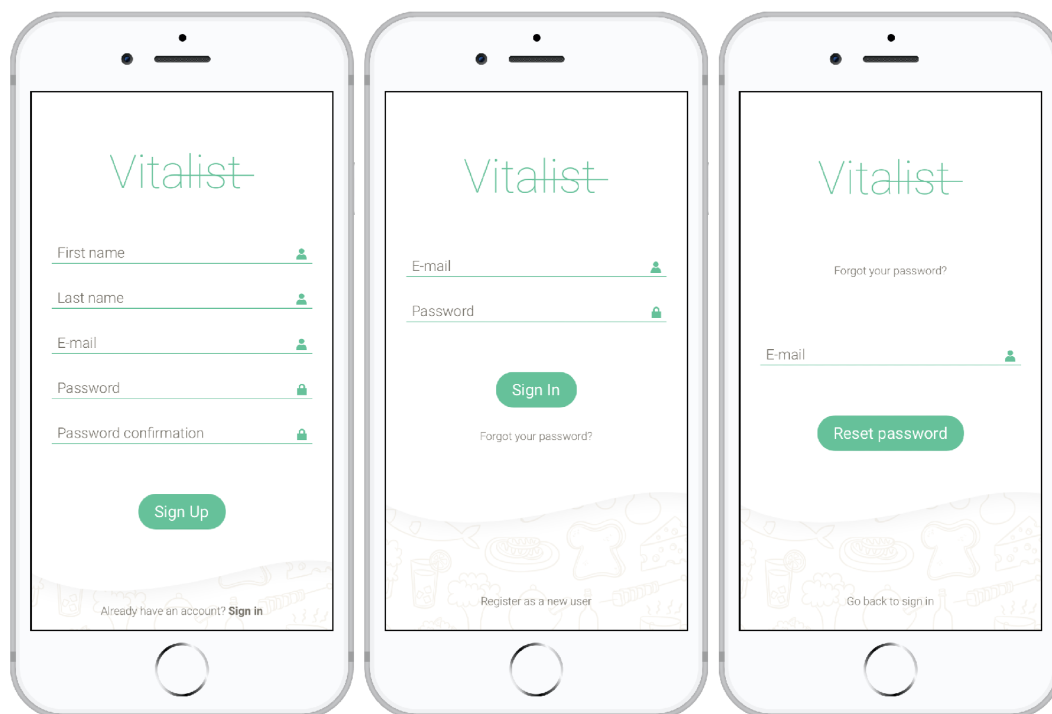
[3] Material Design – https://material.io/

snaží tato aplikace co nejvíce vycházet. *Material Design* se snaží hlavně o demonstrování možných použití jednotlivých dostupných komponent.

Při návrhu uživatelského rozhraní je důležité zjistit, co vše má aplikace umět a které z těchto funkcí jsou nejdůležitější a budou tedy nejčastěji prováděny. V případě aplikace Vitalist je to jednoznačně změna stavu položky. Uživatel by měl možnost změnit stav pouhým kliknutím. Seznam by měl tedy obsahovat jednotlivé položky a to tak, aby se v seznamu dalo jednoduše orientovat. Je potřeba počítat taky s dalšími akcemi, které lze v seznamu provádět. Jsou to: přidání položky, editace položek, nasdílení seznamu a správa kolaborátorů, přejmenování seznamu nebo jeho odstranění. U seznamu budou taky evidovány jednotlivé akce pro zachování historie a případné kontroly, kdo a jak seznam měnil. Uživatel může mít více seznamů a je tedy potřeba mu nabídnout možnost vybrání jiného seznamu nebo jeho založení. V neposlední řadě jsou to taky funkce jako přihlášení nebo registrace, přijmutí nebo odmítnutí kolaborace seznamu a správa uživatelského profilu. Součástí aplikace jsou taky push notifikace nebo emailové zprávy. Tyto případy užití jsou popsány níže.

4.2.1 Přihlášení a registrace

Jelikož má aplikace pouze mobilní verzi, lze předpokládat, že při prvním stažení bude pravděpodobnější registrace než-li přihlášení. Proto se po otevření začíná na registračním formuláři z kterého se lze dostat na obrazovku s přihlášením. V případě zapomenutého hesla se vyplní formulář, po jehož odeslání se odešle emailová zpráva na uvedený email s dalšími instrukcemi.

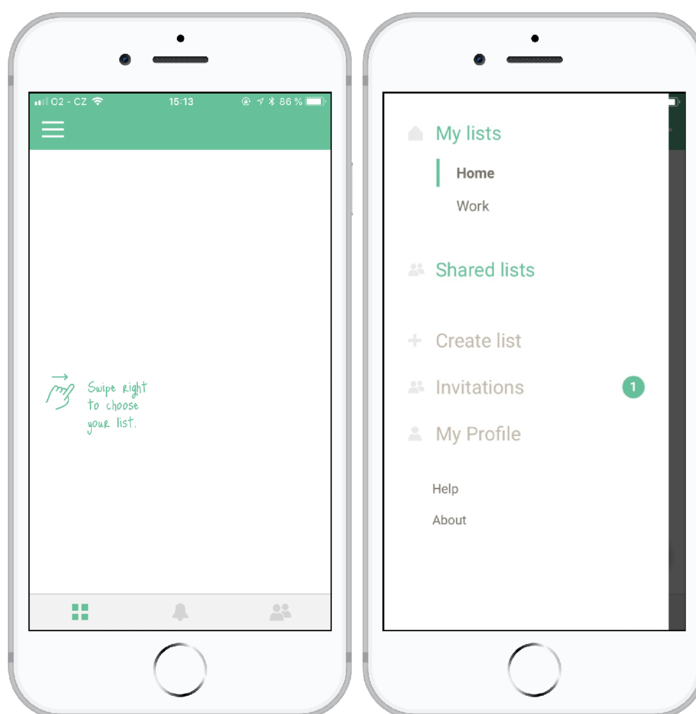


Obrázek 4.1: Zleva registrace, přihlášení a obnova zapomenutého hesla

4.2.2 Seznamy

V případě registrace je uživateli vytvořen základní seznam s položkami, na základě kterého může být demonstrována základní funkcionalita. Aplikace se snaží uživatele navádět k práci nápovědmi, aby byla práce s ní co nejjednoduší. Jedna z takových nápověd je například pro vybrání seznamu po přihlášení, kterou lze vidět na první obrazovce na obrázku 4.2. Díky malé nápovědě je uživatel schopný s aplikací pracovat o mnohem lépe a není potřeba vytvářet složité tutoriály, které uživatele spíše obtěžují než-li pomáhají.

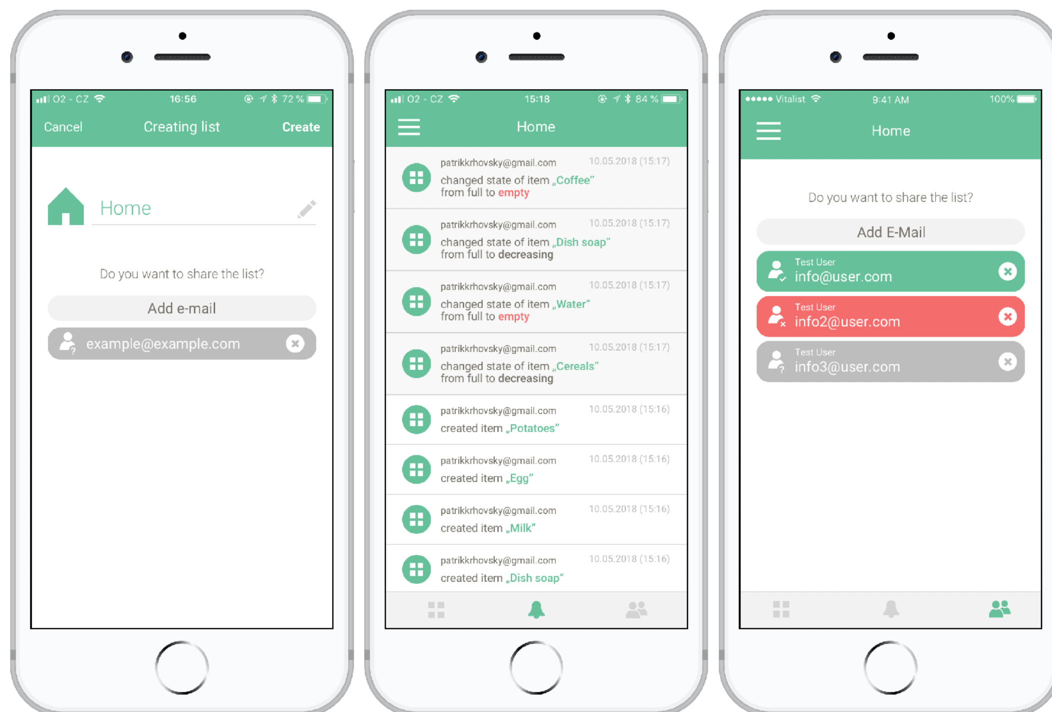
Pro vybírání seznamu bylo vytvořeno menu, které je vždy dostupné pomocí jednoduchého gesta – potáhnutí zleva doprava. Této komponentě se říká `drawer` a je zobrazena na druhé obrazovce na obrázku 4.2. Seznamy jsou v menu rozděleny do dvou kategorií: *My lists* a *Shared lists*. První kategorie obsahuje seznamy, které vytvořil uživatel, druhá naopak ty, které mu byly nasdíleny. Dále jsou zde odkazy na vytvoření seznamu (*Create list*), pozvánky (*Invitations*) a uživatelský profil (*My profile*).



Obrázek 4.2: Zleva nápověda pro vybrání seznamu a menu aplikace, které se objeví při potáhnutí na obrazovce zleva doprava (komponenta `drawer`)

Při vytváření seznamu stačí zadat pouze titulek, viz. první obrazovka na obrázku 4.3. Je zde ale taky možnost tento seznam rovnou sdílet s dalšími uživateli a to pouhým přidáním emailové adresy. Pokud takový uživatel ještě neexistuje, je mu vytvořen neaktivní profil, který se aktivuje po registraci. Na uvedenou emailovou adresu je mu zaslána pozvánka. Po vytvoření seznamu je uživatel přesměrován na seznam, který nyní může začít plnit položkami. Tato část je popsána v kapitole 4.2.3.

Pro rychlé přepnutí mezi jednotlivými části seznamu, kterými jsou položky log aktivity a sdílení, byly přidány záložky na spodní část obrazovky, jak lze vidět na první obrazovce na obrázku 4.2. Tato komponenta se nazývá `TabBar`. Pod záložkou *Log aktivity* je vypsána historie změn v seznamu, obrázek 4.3, které jsou ikonami rozděleny do 3 kategorií:



Obrázek 4.3: Zleva vytvoření seznamu s možností nasdílet seznam dalším uživatelům, log aktivity a správa kolaborátorů

- **Seznam** – vytvoření seznamu, přejmenování seznamu
- **Sdílení** – přidání / odebrání kolaborátora, příjmutí / odmítnutí kolaborace, opuštění seznamu
- **Položky** – přidání / editace / odebrání položky, změna stavu položky – tato změna má navíc jiné pozadí pro lepší orientaci v logu aktivit

Na poslední záložce je správa kolaborátorů, kde uživatel najde stav jednotlivých kolaborátorů (jestli uživatel přijmul / odmítnul jeho pozvánku nebo se případně ještě nevyjádřil), viz. třetí obrazovka na obrázku 4.3. Nechybí ani možnost přidání nebo odstranění kolaborátora. V případě, že má uživatel seznam pouze nasdílený, má zde samozřejmě možnost opustit seznam.

4.2.3 Správa položek v seznamu

Uživatelé, kteří obsluhují seznam, lze rozdělit na 2 role, vlastník a kolaborátor. Vlastník může provádět veškeré operace nad seznamem, které jsou:

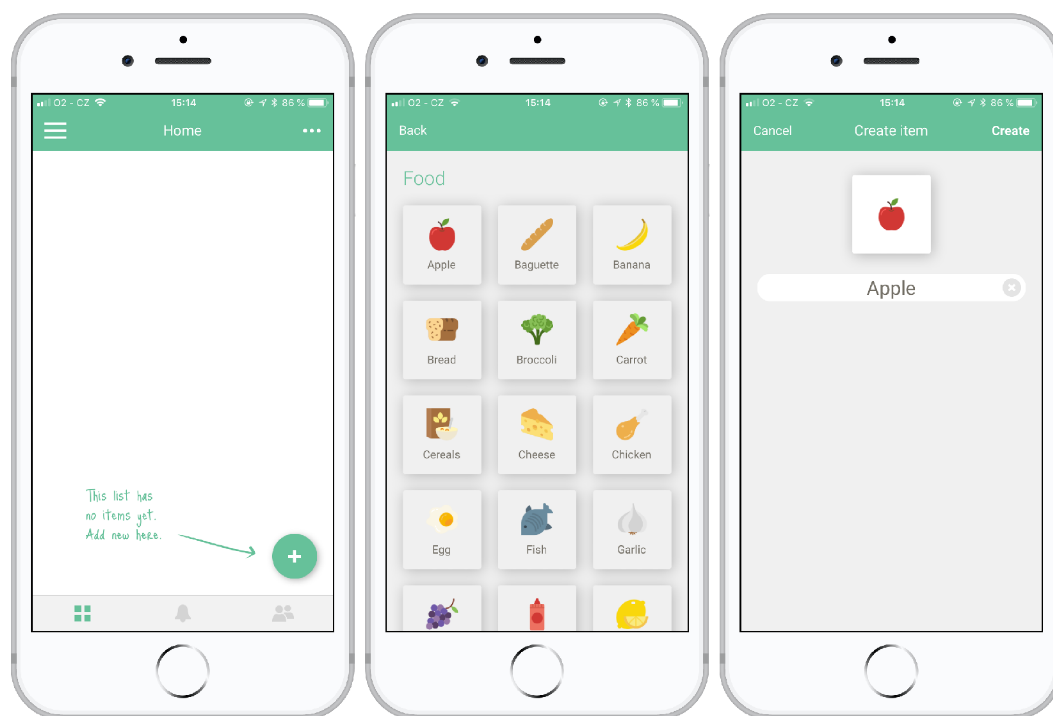
- přidání / editace / odebrání položky v seznamu
- změna pořadí položek v seznamu
- změna stavu položky
- přejmenování / odstranění seznamu

- nasdílení seznamu

Kolaborátor může měnit pouze stav položky. Zbylé operace nemá povoleno provádět (kontroluje se v serverové aplikaci) a nejsou mu tedy v uživatelském prostředí ani nabízeny.

Správa položek začíná možností přidat položku. Tuto možnost by měl mít uživatel pořád a lehce dostupnou, aby ji nemusel nijak složitě hledat. Tuto funkci zajišťuje obtékající tlačítko, tzv. `FAB`[4], který je zobrazen na první obrazovce na obrázku 4.4. Na této obrazovce si lze taky všimnout další nápovědy, která v případě prázdného seznamu navádí právě ke kliknutí na tlačítko pro přidání položky do seznamu. Po kliknutí na toto tlačítko se uživatel přesune k přidání položky.

Přidání položky je rozděleno na 2 kroky, kde v prvním kroku uživatel vybere ikonu. Ikony jsou řazeny do kategorií, viz. prostřední obrazovka obrázku 4.4, pro lepší orientaci – stejně jako to měly popisované aplikace v kapitole 2.2. Ikonu lze vybrat kliknutím na ni. Po následném vybrání ikony je uživatel přesunut na obrazovku s editací názvu položky, která lze vidět vpravo na obrázku 4.4. Na této obrazovce se převyplní titulek podle vybraného obrázku. Tento titulek lze samozřemě změnit, ale jedná se o způsob rychlého přidání položky, jelikož se předpokládá, že se ve většině případů měnit nebude.



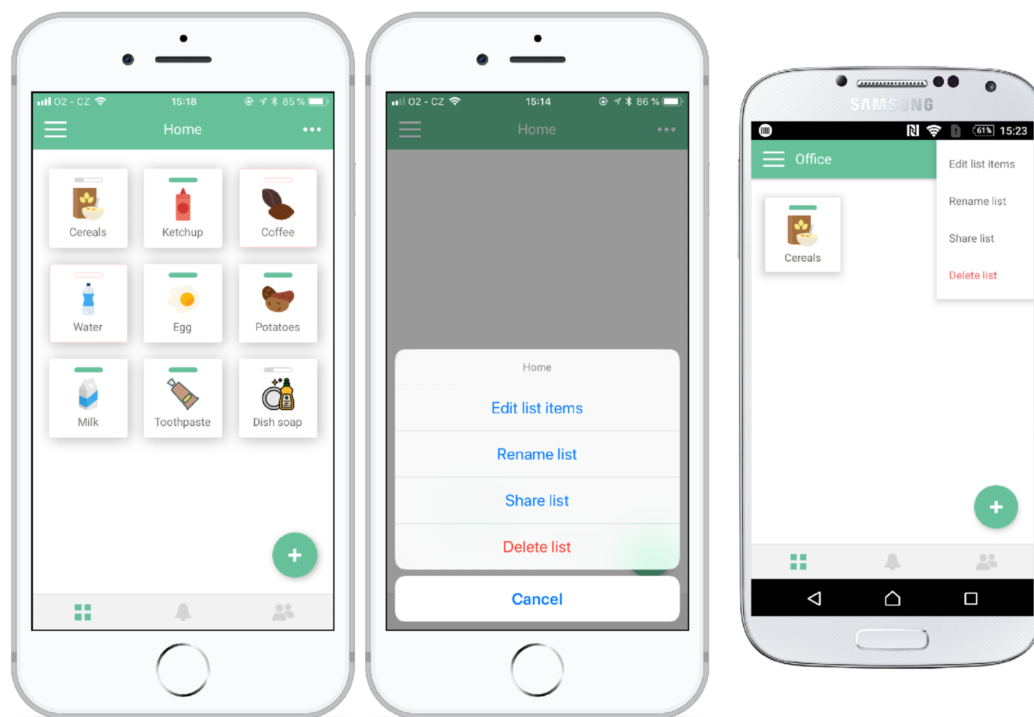
Obrázek 4.4: Zleva tlačítko pro přidání položky do seznamu, vybrání ikony pro položku a formulář pro vytvoření položky

Položky jsou v seznamu řazeny do mřížky a na řádek jsou vloženy 3 položky. Není tedy předpokládáno velké množství těchto položek, ale naopak jen ty nejdůležitější, aby měl uživatel nad seznamem pokud možno co nejlepší přehled. Každá položka obsahuje ukazatel stavu, ikonu a titulek. Po předchozím testování, které je popsáno v kapitole 6.1, bylo zjištěno, že uživatelé se nejvíce příklání ke 3 stavům položky a to:

[4]Floating action button – https://material.io/design/components/buttons-floating-action-button.html

- **Plný stav** – ukazatel stavu je plně zelený
- **Dochází** – ukazatel stavu je skoro prázdný a šedý
- **Prázdný stav** – ukazatel stavu je červený a prázdný. Navíc je celá položka obtáhnuta červeným orámováním pro lepší zýraznění

Při kliknutí na položku se tento stav změní. Tento seznam lze vidět na první obrazovce na obrázku 4.5.



Obrázek 4.5: Zleva zobrazení položek v seznamu, komponenta `ActionSheet` a komponenta `Material Menu`

Další a poslední akčním tlačítko se nachází v pravé části navigace, viz. první obrazovka obrázku 4.5. Po kliknutí se zobrazí další akce, které lze provádět nad seznamem. Jsou to akce:

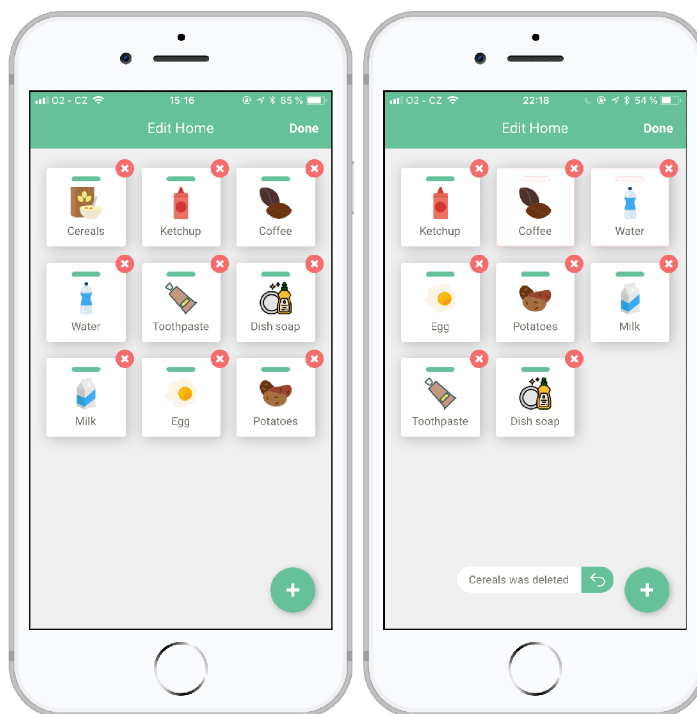
- Editace položek v seznamu – přesune uživatele do editačního režimu, viz. níže
- Přejmenování seznamu
- Správa kolaborátorů – přesune uživatele na záložku se správou kolaborátorů
- Odstranění seznamu

Toto akční tlačítko je zajímavé hlavně odlišnou komponentou pro výpis akcí po kliknutí, podle toho, na kterém systému je mobilní aplikace spuštěna. Na každém systému je totiž očekávána jiná komponenta a pro zachování zvyků na systémech, kde aplikace běží, je na systému iOS použita komponenta `ActionSheet`[5], kterou lze vidět na druhé obrazovce na

[5]Komponenta `ActionSheet` – `https://developer.apple.com/ios/human-interface-guidelines/views/action-sheets/`

obrázku 4.5. Naopak uživatelé systému Android očekávají spíše vyjížděcí menu[6], které je součástí *Material Design*, a lze ho vidět na třetí obrazovce na obrázku 4.5. Tyto fakta byla zjištěna testováním, které je popsáno v kapitole 6.3.

Při přesunu uživatele do editačního režimu, viz. první obrazovka na obrázku 4.6, se mění funkce položek. Nyní se při kliknutí na položku uživatel přesune k její editaci, která funguje podobně jako přidání položky. Při dlouhém podržení položky v editačním režimu je možné položku přesouvat a uživatel si tak může seřadit položky podle svého uvážení. Poslední funkce v editačním režimu je možnost odstranění položky kliknutím na křížek v červeném kolečku. Po odstranění se ve spodní části obrazovky zobrazí komponenta, díky které lze vzít tuto akci zpět. Tato komponenta jde vidět na druhé obrazovce na obrázku 4.6. Uživatel ví, co chce udělat. Není tedy potřeba trvat na potvrzení, že chce položku opravdu odstranit, stačí mu důvěřovat a dát mu možnost vrátit zpět poslední provedenou akci. [5]



Obrázek 4.6: Zleva editační režim a komopnenta pro vrácení zpět poslední akce

4.3 Návrh komunikace mobilní aplikace

Vývojář mobilní aplikace, která uchovává data, má několik možností, jak s nimi pracovat. Nejjednodušší možnost je ukládání do lokální paměti telefonu. Další možnost je využít nějakou z databází s kterou bude komunikovat mobilní aplikace. Je zde taky možnost vytvořit si serverovou aplikaci, která bude tyto služby obstarávat a aplikaci bude nabízet přístup například přes REST API. Na trhu je již nějakou dobu i platforma Google Firebase, která nabízí spoustu nástrojů, které pomáhají vývojářům mobilních aplikací řešit tyto problémy. Služby Google Firebase byly rozepsány v kapitole 3.4.

Pří návrhu komunikace aplikace Vitalist je potřeba brát v úvahu tyto základní fakta:

[6]Komponenta *Material Menu* – https://material.io/design/components/menus.html

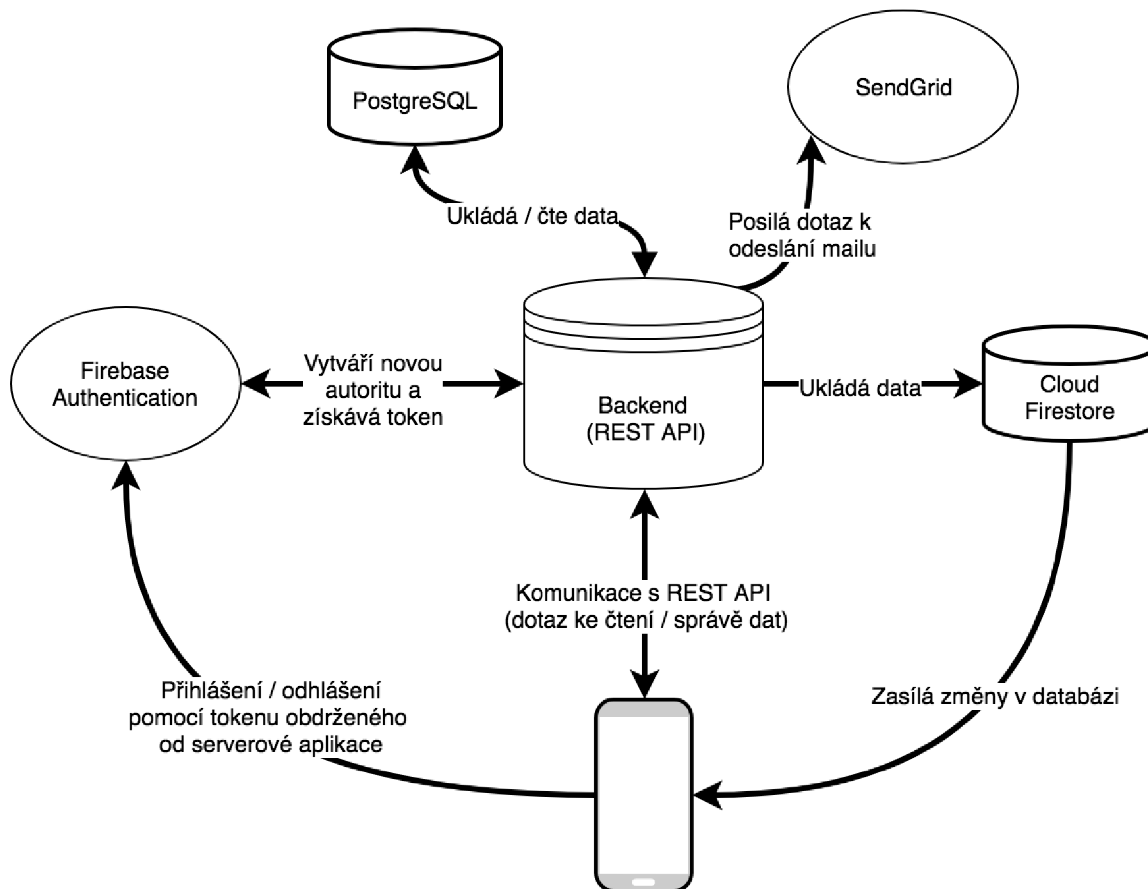
- **Aplikace musí pracovat v reálném čase** – k dosažení tohoto cíle pomůže služba z Google Firebase – *Cloud Firestore* (kapitola 3.4.1)
- **Aplikace bude odesílat notifikace do zařízení** – nejjednodušší možnost zasílání notifikací do zařízení se systémem iOS i Android opět nabízí Google Firebase – *Cloud Messaging* (kapitola 3.4.2)
- **Aplikace bude odesílat emaily** – lze využít službu *SendGrid*[7]

Po sepsání předešlých faktů jsem se rozhodl i k vytvoření serverové aplikace, která bude aplikaci poskytovat REST API a to z těchto důvodů:

- **Logika aplikace na jednom místě** – jeden z nejzásadnějších důvodů. Při případném rozšíření mobilní aplikace o webovou aplikaci nemusí být psána znovu logika aplikace, jelikož je obsažena v serverové aplikaci (týká se hlavně ukládání dat do databáze, rozesílání notifikací a emailů). Další výhodou je taky případný přechod mezi službami, který stačí řešit pouze zde a ne na všech klientských aplikacích
- **Vytíženost *Cloud Firestore*** – jak je uvedeno v kapitole 3.4.1, kvóty databáze *Cloud Firestore* nejsou nijak vysoké, ale ceny za jejich navýšení ano. Pokud by se měly všechny data načítat z ní, mohla by se tato služba značně prodražit. Je tedy zvolen způsob přednačtení dat z REST API a databáze *Cloud Firestore* informuje připojené klienty pouze o případných změnách. Implementace této techniky je popsána v kapitole 5.2.3

Pro lepší pochopení komunikace poslouží schéma na obrázku 4.7, které zobrazuje jednotlivé datové toky. Lze vidět, že mobilní klient komunikuje se serverovou aplikací a ta tyto změny posílá do databáze *Cloud Firestore*. Mobilní klient poslouchá pouze na změny, které se ho v této databázi týkají a ty případně zpracovává. Mobilní klient ale do databáze *Cloud Firestore* nic nezapisuje, pouze data čte. Stejný přístup komunikace je i u notifikací. Se všemi externími službami (jako je třeba služba *Sendgrid* nebo objektově-relační databáze *PostgreSQL*) komunikuje pouze serverová aplikace.

[7]Služba *SendGrid* – https://sendgrid.com



Obrázek 4.7: Schéma komunikace mobilní aplikace s ostatními službami

4.4 Návrh struktury databáze

Po návrhu uživatelského rozhraní lze snadno navrhnout i strukturu databáze, jelikož samotným návrhem uživatelského rozhraní jsme definovali i data, která bude aplikace potřebovat k jejímu chodu. Jak bylo naznačeno v kapitole 4.3, data se budou ukládat do 2 databází. Jako hlavní databáze byla zvolena objektově-relační databáze *PostgreSQL*, která bude obsahovat všechna data, která aplikace potřebuje pro svůj chod. Na druhé straně databáze *Cloud Firestore* bude obsahovat pouze ta data, která je potřeba udržovat aktuální a synchronizovat je mezi zařízeními v reálném čase.

Objektově-relační databáze *PostgreSQL* obsahuje celkem 7 tabulek, jejiž vazby jsou znázorněny na obrázku 4.8. Dokumentová databáze *Cloud Firestore* ukládá data ve formátu `JSON` a obsahuje následující kolekce:

- `lists` – obsahuje informace o seznamech ve formátu:

```
{
        id: <uuid>,
        title: <string>,
        order: <number>,
        createdAt: <string>,
        updatedAt: <string>,
        createdById: <uuid>,
```

```
        items: <array of uuid>,
        collaborators: <object of {
            id: <string>,
            invitedAt: <string>,
            acceptedAt: <string>,
            declinedAt: <string>,
        }>,
}
```

kde `collaborators` je jako objekt z toho důvodu, že databáze Cloud Firestore nezvládá vyhledávat v poli, ale pouze v objektu. Díky tomu lze pak napsat příkaz typu `WHERE collaborators.<uuid> IS NOT NULL` pro vyhledání všech seznamů, kde je uživatel kolaborátor

- `items` – obsahuje informace o položkách ve formátu:

```
{
        id: <uuid>,
        title: <string>,
        icon: <string>,
        order: <number>,
        state: <number>,
        createdAt: <string>,
        updatedAt: <string>,
        createdById: <uuid>,
        listId: <uuid>,
}
```

- `activities` – obsahuje informace o aktivitě v seznamech. Nabízí se její zrušení z hlediska úspory dat. Obsahuje data ve formátu:

```
{
        id: <uuid>,
        type: <string>,
        data: <json>,
        createdAt: <string>,
        updatedAt: <string>,
        createdById: <uuid>,
        listId: <uuid>,
}
```

items
PK id <uuid>
title <string>
order <number>
icon <string>
state <number>
FK createdById <uuid>
FK listId <uuid>
createdAt <timestamp>
updatedAt <timestamp>

notificationTokens
PK id <uuid>
deviceId <string>
token <string>
FK userId <uuid>
createdAt <timestamp>
updatedAt <timestamp>

1:n

1:n

lists
PK id <uuid>
title <string>
order <number>
FK createdById <uuid>
updatedAt <timestamp>
createdAt <timestamp>

users
PK id <uuid>
email <string>
emailConfirmedAt <string>
passwordHash <string>
firstName <string>
lastName <string>
FK invitedById <string>
invitedAt <timestamp>
createdAt <timestamp>
updatedAt <timestamp>
resetPasswordSentAt <timestamp>

1:n

1:n

1:n

1:n

1:n

userLogin
PK uniqueId
refreshToken <string>
agent <string>

activities
PK id <uuid>
type <string>
data <json>
FK createdById <uuid>
FK listId<uuid>
createdAt <timestamp>
updatedAt <timestamp>

1:n

1:n

usersInLists
PK id <uuid>
userId <uuid>
listUd <uuid>

Obrázek 4.8: ER-model databáze

Kapitola 5

Implementace

Po návrhu služby přišla na řadu implementace. Před implementací mobilní aplikace bylo potřeba nejprve implementovat serverovou aplikaci, s kterou mobilní aplikace komunikuje prostřednictvím protokolu HTTP a vytvořeného REST API a její implementace je popsána na začátku kapitoly. Dále je v kapitole popsán vývoj mobilní aplikace pro zařízení se systémy iOS a Android. Kapitola se zabývá jak popisem toho, které technologie aplikace využívají, tak taky základními, ale zajímavými principy, které byly implementovány.

5.1 Implementace serverové aplikace

Pro plnohodnotné fungování klientské aplikace byla vytvořena serverová aplikace (dále označována jako API), která využívá architektury REST založené na protokolu HTTP. API bylo implementováno v jazyce *JavaScript* s použitím serverového frameworku *node.js*[1]. Pro jednodušší implementaci REST API byl použit webový framework *koa.js*[2]. Tabulka 5.1 obsahuje endpointy, které byly v rámci API implementovány.

Data odesílaná nebo vracená z API jsou ve formátu `JSON`. Pro všechny tyto endpointy jsou vytvořeny integrační testy. Samozřejmostí je kontinuální integrace pomocí služby Travis CI[3] a automatické nasazení na cloudový server běžící na službě Heroku[4]. Na službě Heroku jsou vytvořeny celkem 3 instance, které jsou přidány do jedné `pipeline`[5]. Instance jsou:

- **development** – zde je nasazen kód z repozitáře z větve `development` a slouží pro vývoj klientské aplikace
- **staging** – zde je nasazen kód z repozitáře z větve `master` a slouží pro otestování serverové a klientské aplikace před uvolněním do produkce. Po otestování je tato verze povýšena[6] do instance `production`
- **production** – produkční verze API

[1] *Node.js* – `https://nodejs.org/en/`

[2] *Koa.js* – `https://koajs.com/`

[3] Služba Travis CI je služba, která nabízí kontinuální integraci projektů V tomto případě se jednalo o automatické sestavení, otestování projektu a jeho následné nasazení. Více o službě na `https://travis-ci.com/`

[4] Heroku je cloudová platforma, která umožňuje společnostem vyvíjet, zvěřejnit, monitorovat a skálovat aplikace. Více o službě Heroku na `https://www.heroku.com/what`

[5] Heroku pipeline – `https://devcenter.heroku.com/articles/pipelines`

[6] Heroku promoting – `https://devcenter.heroku.com/articles/pipelines#promoting`

Obrázek 5.1: Seznam a popis jednotlivých implementovaných endpointů

HTTP příkaz	**Endpoint**	**Krátký popis**
	Autentizace	
POST	/auth/native	Autentizace uživatele
POST	/auth/token	Obnova přístupového tokenu
	Info	
GET	/info	Testovací endpoint
	Položky	
POST	/lists/{listId}	Vytvoření položky v seznamu
PATCH	/items/{itemId}	Editace položky v seznamu
DELETE	/items/{itemId}	Smazání položky ze seznamu
	Seznamy	
GET	/lists	Vrací uživatelovi seznamy
POST	/lists	Vytvoření seznamu
GET	/lists/{listId}	Vrací seznam podle zadaného identifikátoru
PATCH	/lists/{listId}	Editace seznamu
DELETE	/lists/{listId}	Odstranění seznamu
PATCH	/lists/{listId}/collaborators	Upravuje kolaborátory seznamu
GET	/lists/collaborator	Vrací seznamy sdílené s uživatelem
PUT	/lists/{listId}/collaborator/accept	Přijmutí kolaborace v seznamu
PUT	/lists/{listId}/collaborator/decline	Odmítnutí kolaborace v seznamu
GET	/lists/{listId}/activities	Vrací log aktivity seznamu
	Uživatelé	
POST	/users	Vytvoření nového uživatele
PATCH	/users	Editace uživatelského profilu
PUT	/users/password	Změna uživatelského hesla
POST	/users/reset-password	Podání žádosti o změnu zapomenutého hesla
PUT	/users/reset-password	Změna zapomenutého hesla
POST	/users/notification-token	Uložení tokenu pro odesílání notifikací
DELETE	/users/notification-token	Odstranění tokenu pro odesílání notifikací

Tento způsob nasazení aplikace je první z 12 bodů *The twelve-factor app*[7], kterými jsem se při vývoji aplikace řídil.

[7] The twelve-factor app – https://12factor.net/

5.1.1 Autentizace uživatelů

Autentizace uživatelů probíhá pomocí standardu *OAuth 2.0*[8]. Po přihlášení uživatele se tedy vrací přístupový token (`access token`) a jeho platnost. Tento token je tvořen podle standardu *JSON Web Token*[9] s využitím stejnojmenné knihovny[10] a hašovací funkce `SHA-256`. Přístupový token klientská aplikace přidává do hlavičky REST dotazů odesílajících se na API pod klíčem `Authorization`. Ten je poté serverovou aplikací dekódován a ověřuje se jeho platnost – 60 minut.

Společně s přístupovým tokenem je odesílán i obnovovací token (`refresh token`). Ten slouží pro získání nového přístupového tokenu bez nutnosti nového přihlášení uživatele pomocí emailu a hesla. Je tvořen stejně jako přístupový token, ale jeho platnost je neomezená. Je uložen v databázi společně s identifikátorem uživatele a rozšiřujícími informacemi, jako je například jméno klientské aplikace nebo IP adresa, pro jednoznačnou identifikaci nebo pro případ, kdy by si uživatel chtěl zkontrolovat, které zařízení jsou nyní přihlášeny. V případě odhlášení uživatele ze zařízení je tento záznam z databáze smazán a token zneplatněn. Znamená to tedy, že v případě přihlášení uživatele z více zařízení, má každé z nich svůj vlastní obnovovací token.

Pro přístup ke službám Google Firebase je navíc posílán ještě přístupový token pro tyto služby, dále označován jako `firebase token`, jehož platnost a obnovení je shodné s přístupovým tokenem do serverové aplikace. Serverová aplikace komunikuje se služby Google Firebase pomocí `service account`[11], díky kterému má plný přístup ke všem službám. `Firebase token` serverová aplikace získá pomocí metody `createCustomToken`[12]. Tento token slouží pro přihlášení klientské aplikace do služeb Google Firebase.

5.1.2 Databáze

Serverová aplikace využívá objektově-relační databází *PostgreSQL*. Pro práci s databází je použita knihovna *Sequelize*[13]. Knihovna využívá objektově relační zobrazení, které konvertuje data mezi relační databází a objekty. Aplikace obsahuje i automatické migrace, které knihovna *Sequelize* taky poskytuje[14]. Slouží pro jednoduchý přechod mezi verzemi aplikace a pro plnou funkčnost automatického nasazení. V případě, že verze obsahuje nové migrace, jsou tyto migrace automaticky spuštěny před spuštěním samostatné aplikace na cloudovém serveru. Toho lze docílit pouhým přidáním následujícího řádku do konfiguračního souboru[15] používaného službou Heroku:

```
release: node_modules/.bin/sequelize db:migrate
```

5.1.3 Využití front pro rychlejší zpracování dotazu

Při vývoji API v jazyce *JavaScript* je důležité myslet na to, že *JavaScript* je jednovláknový programovací jazyk [10]. Pokud tedy přijde dotaz, je vložen do fronty (`event queue`),

[8] Standard *OAuth 2.0* – `https://oauth.net/2/`

[9] Standard JSON Web Token – `https://jwt.io/`

[10] *Knihovna JSON Web Token* – *`https://github.com/auth0/node-jsonwebtoken`*

[11] Firebase service account – https://firebase.google.com/docs/admin/setup

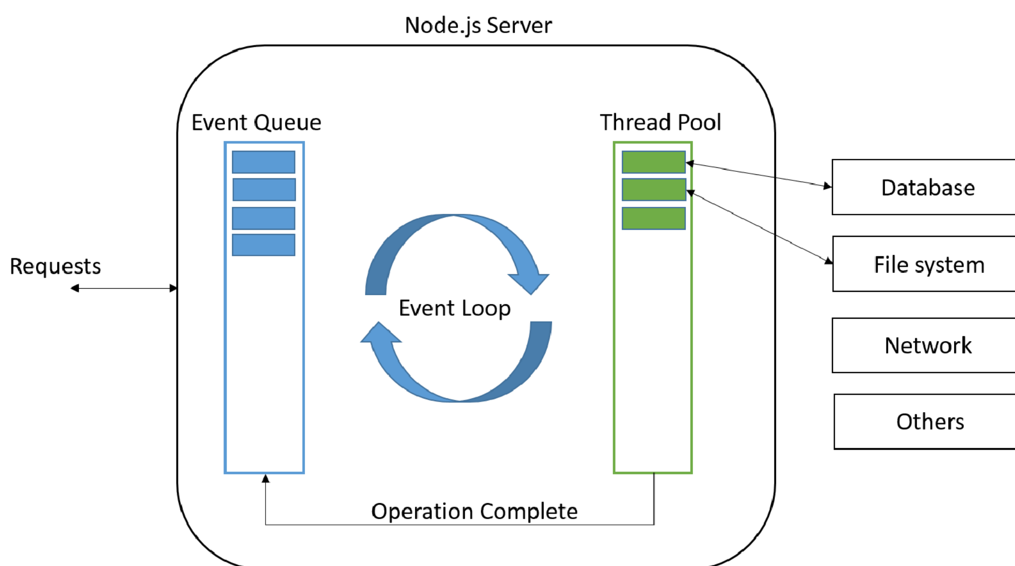
[12] Vytvoření přístupového tokenu pro služby Firebase – `https://firebase.google.com/docs/auth/admin/create-custom-tokens`

[13] Knihovna *Sequelize* – `http://docs.sequelizejs.com/`

[14] Migrace pomocí knihovny *Sequelize* – `http://docs.sequelizejs.com/manual/tutorial/migrations.html`

[15] Heroku Procfile – `https://devcenter.heroku.com/articles/procfile`

kterou zpracovává `event loop`. Pokud je tedy nějaký dotaz na frontě, je zpracován. Pokud zpracování dotazu obsahuje nějakou blokující IO operaci, například získání dat z databáze, je tato operace přidána neblokující IO operací do fronty `worker pool`. `Worker` je v tomto případě posix vlákno běžící v pozadí, které zpracovává blokující IO operace. Mezi tím může jednovláknový `ęvent loop` dále zpracovávat další přicházející dotazy od klientů. Jakmile jsou data z databáze načtena, je vložen do fronty `callback`, který bude zpracování pomocí `event loop`. Následně mohou být získané data odeslány klientovi. Tento proces znázorňuje obrázek 5.2.



Obrázek 5.2: Znázornění zpracování požadavků na *node.js* server [9]

Počet `workerů` je ale menší, než počet klientů, kteří mohou v jeden moment komunikovat s aplikací. V případě, že by operace na pozadí trvaly příliš dlouho, nestihli by se obsloužit ostatní klienti. Je proto potřeba dbát na rychlost těchto operací a vykonávat pouze ty operace, které jsou nezbytné [3]. Například odeslání uvítací emailové zprávy po registraci klienta, není nutné provést před obsloužením klienta, tedy vrácením výsledku.

Jeden z principů, jak tuto situaci řešit, je vložit takovou operaci do fronty, kterou bude poté zpracovávat jiný proces a tím nebude blokován proces, který zpracovává dotazy od klientů. Pro frontu je využita knihovna *redis* [16]. Redis je databáze, která ukládá své data v paměti a pro přidání záznamu lze tedy využít neblokující IO operaci. Pro práci s frontou je využita knihovna *bull*[17]. Implementovaná aplikace využívá následující fronty:

- `emails` – fronta pro odesílání emailových zpráv (po registraci, obnova zapomenutého hesla nebo pozvánka do seznamu),
- `firestore` – fronta pro odesílání změn do databáze *Cloud Firestore*,
- `notifactions` – fronta pro odesílání notifikací na mobilní zařízení pomocí služby *Firebase Cloud Messaging*.

[16]Databáze *Redis* – `https://redis.io/`
[17]Knihovna *bull* – `https://github.com/OptimalBits/bull`

Přidání takové operace do fronty se značí jako `job`. Ten obsahuje typ, pro rozeznání operace, která se má vykonat, a data, která jsou potřeba k vykonání operace. V případě uvítací zprávy by mohl výsledný záznam mohl vypadat takto:

```
{
	type: 'EMAILS/WELCOME-EMAIL',
	data: { email: 'example@example.com' },
}
```

Proces, který poté zpracovává frontu, pak podle typu vykoná operaci, která je vykonána nezávisle na fungování hlavního procesu serverové aplikace. Zpracování fronty by pak mohlo vypadat například takto:

```
queue.process(async job => {
	// Zvoleni akce podle typu
	switch(job.type) {
		case 'EMAILS/WELCOME-EMAIL':
			// Asynchronni akce
			await sendWelcomeEmail(job.data.email)
	}
})
```

5.2 Implementace mobilní aplikace

Mobilní aplikace byla implementována v jazyce *JavaScript* pomocí knihovny *React Native* a je dostupná pro systém iOS a Android. Aplikace byla implementována podle předem navrhnutého uživatelského prostředí, které bylo popsáno v kapitole 4.2. Díky kompilátoru *babel*[18] jsou využity nejnovější konstrukce z ECMAScript 2018[19].

5.2.1 Authentizace uživatele

Po zapnutí aplikace je načtena informace o přihlášeném uživateli uložená v lokální paměti telefonu. Ta v sobě nese identifikátor uživatele, přístupový token k API, jeho platnost a token pro obnovu přístupového tokenu. Tato informace se do paměti ukládá po přihlášení uživatele nebo v případě její změny. Přístupový token je totiž potřeba obnovit před jeho splatností. V případě zneplatnění obnovovacího tokenu je uživatel z aplikace odhlášen. Tento proces je spuštěn po startu aplikace pomocí efektu `fork`, který je součástí knihovny *redux saga*, a vypadá následovně:

```
export function* authFlow() {
	// Cekani na nacteni dat z~pameti telefonu
	yield take(REHYDRATE)

	// Ziskani nactenych authorizacnich data
	let auth = yield select(authSelectors.getAuth)

	while (true) {
```

[18] Kompilátor *babel* – https://babeljs.io/

[19] Co je nového v ECMAScript 2018 – https://medium.com/front-end-hacking/javascript-whats-new-in-ecmascript-2018-es2018-17ede97f36d5

```
            // Pokud neni uzivatel prihlasen,
            // cekame na jeho prihlaseni nebo registraci
            if (!auth) {
                const { signInAction, signUpAction } = yield race({
                    signInAction: take(AUTH.SIGN_IN),
                    signUpAction: take(AUTH.SIGN_UP),
                })

                if (signInAction) {
                auth = yield call(signIn, signInAction)
                }

                if (signUpAction) {
                auth = yield call(signUp, signUpAction)
                }

                // V~pripade chyby se ceka na nove prihlaseni
                if (!auth) {
                    continue
                }
            }

            // Spusteni obnovy pristupoveho tokenu
            // dokud se uzivatel neodhlasi
            yield race({
                signOutAction: take(AUTH.SIGN_OUT),
                refreshTokenLoop: call(tokenLoop, auth),
            })

            auth = null
        }
}

function* tokenLoop(auth) {
        while (true) {
            try {
                // Obnoveni pristupoveho tokenu
                const response = yield call(api.auth.token, auth)

                // Ulozeni ziskanych tokenu
                yield call(setTokens, response)

                // Cekani na expiraci tokenu
                yield call(delay, response.expiresIn - MIN_TOKEN_LIFESPAN)

            } catch (error) {
                // Odhlaseni v~pripade chyby
                yield call(signOut)
```

```
        }
    }
}
```

5.2.2 Komunikace s API

Komunikace s API probíhá výhradně v *redux saga* generátorech z důvodu lepšího zpracování akce, kterým může být například zapnutí indikátoru aktivity po spuštění asynchronní akce. Jednou z akcí může být například získání seznamů po přihlášení uživatele. Generátor se spustí po vložení *redux* akce `LISTS/FETCH` a vykoná následující operace:

```
// Vlozi redux akci, ktera spusti indikator aktivity
yield put({ type: 'LISTS/FETCH_PENDING' })

try {
    // Zavolani API
    const data = yield call(api.lists.fetch)

    // Vlozi redux akci, ktera ulozi data a vypne indikator aktivity
    yild put({ type: 'LISTS/FETCH_FULFILLED', payload: data })
} catch (error) {
    // Zpracovani chyby
    yield put({ type: 'LISTS/FETCH_REJECTED', payload: error })
}
```

5.2.3 Práce s daty

Načtené data z API je potřeba někam uložit a pracovat s nimi v rámci celé aplikace. K tomu slouží knihovna *redux*, která byla popsána v kapitole 3.2.1. Při zpracování dat z API jsou data ukládána v normalizovaném stavu (kapitola 3.2.4). Díky tomu lze udržovat samostatné entity, podobně jako v databázi. Při změně tak stačí upravit pouze jeden záznam, který je pak změněn ve všech komponentách, které ho využívají. Pro ukládání entit je využita struktura `Immutable Map`, jejiž výhody jsou popsány v kapitole 3.2.3. Struktura stavu aplikace pak vypadá následovně:

```
{
    // Pomocne data komponent
    app: { ... }
    // Informace o~prihlasem uzivateli
    auth: { accessToken: string, profile: { ... }, ... },
    // Entity
    entities: {
        activities: Map,
        items: Map,
        lists: Map,
        users: Map,
    },
    // Doplnujici informace o~nactenych listech
    lists: { user: { isFetching: bool, }, ... },
```

```
}
```

Pro udržování aktuálních dat je využita databáze *Cloud Firestore*, která synchronizuje data v mobilním zařízení. Z důvodu úspory kvót databáze, jsou synchronizovány pouze nové nebo upravené data v databázi *Cloud Firestore*. Tohoto výsledku lze dosáhnout tím, že se uloží čas, kdy bylo zavoláno API pro načtení prvotních dat, a z databáze *Cloud Firestore* se budou získávat pouze ty data, které byly upraveny nebo vytvořeny po tomto čase. Výsledný dotaz na databázi *Cloud Firestore* pak vypadá následovně:

```
db.collection('items')
        .where('listId', '==', listId)
        .where('updatedAt', '>', initTime)
```

Pro získávání změn v databázi se využívá funkce `onSnapshot`, která vrací nový stav databáze[20]. Jak již bylo zmíněno, *JavaScript* je jednovláknová aplikace a nelze tedy spustit nové vlákno, které by upravovalo přímo stav aplikace na základě přijatých změn. Řešení nabízí opět knihovna *redux saga* a to konkrétně technika `channel`[21]. Díky tomu lze odeslat zprávu do kanálu, která je následně získána *redux saga* generátorem, zpracována a uložena do stavu aplikace. Toto použití znázorňuje následující kód:

```
function getListsChannel() {
        return eventChannel(emit => query.onSnapshot(emit))
}

fork(() => {
        const channel = getListsChannel()

        while (true) {
                // Cekani na zpravu
                const data = yield take(channel)

                // Ulozeni dat
                saveLists(data)
        }
})
```

5.2.4 Komponenta

Komponenta je základní stavební kámen aplikace využívající knihovnu *React*. Základem je tvořit menší komponenty, z kterých poté bude složena jedna větší komponenta a dbát na jejich možnost znovupoužitelnosti. Příklad dekompozice na komponenty znázorňuje obrázek 5.4. S využitím knihovny *redux*, lze komponenty rozdělit na prezentační komponenty a tzv. kontejnery. Rozdíl mezi nimi je znázorněn v tabulce 5.3. Připojení kontejneru ke stavu aplikace a získání dat, nebo naopak změny dat, lze docílit pomocí funkce `connect`[22].

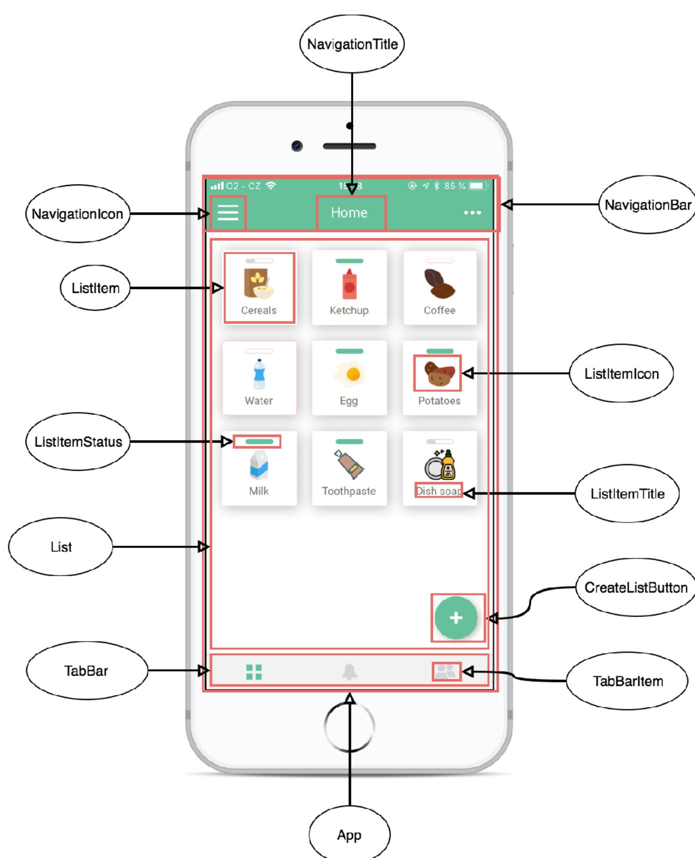
[20] Struktura `QuerySnapshot` – https://firebase.google.com/docs/reference/js/firebase.firestore.QuerySnapshot

[21] Popis techniky *redux saga* `channel` – https://redux-saga.js.org/docs/advanced/Channels.html

[22] Funkce `connect` je součást knihovny *react redux* – https://github.com/reduxjs/react-redux

Obrázek 5.3: Rozdíl mezi prezentační komponentou a kontejnerem [7]

	Prezentační komponenta	**Kontejner**
Účel	Definuje, jak věci vypadají (styly)	Definuje, jak věci pracují (získání dat a jejich editace)
Připojena k *reduxu*	Ne	Ano
Čtení dat	Z parametrů komponenty	Ze stavu aplikace
Změna dat	Pomocí funkcí získaných z parametrů	Pomocí funkcí připojených do *reduxu*



Obrázek 5.4: Znázornění dekomopozice scény na komponenty

Kapitola 6

Testování

V kapitole 4.2 byl popsán finální návrh uživatelského prostředí. Tomuto návrhu předcházelo několik verzí a jejich následné testování. Na základě výsledků testování byla vždy vytvořena nová verze, která se snažila odstranit nedostatky zjištěné testováním. Postupný vývoj a testování je shrnuto v této kapitole. Na závěr byla aplikace publikována v obchodě Google Play a Apple App Store pod názvem Vitalist.

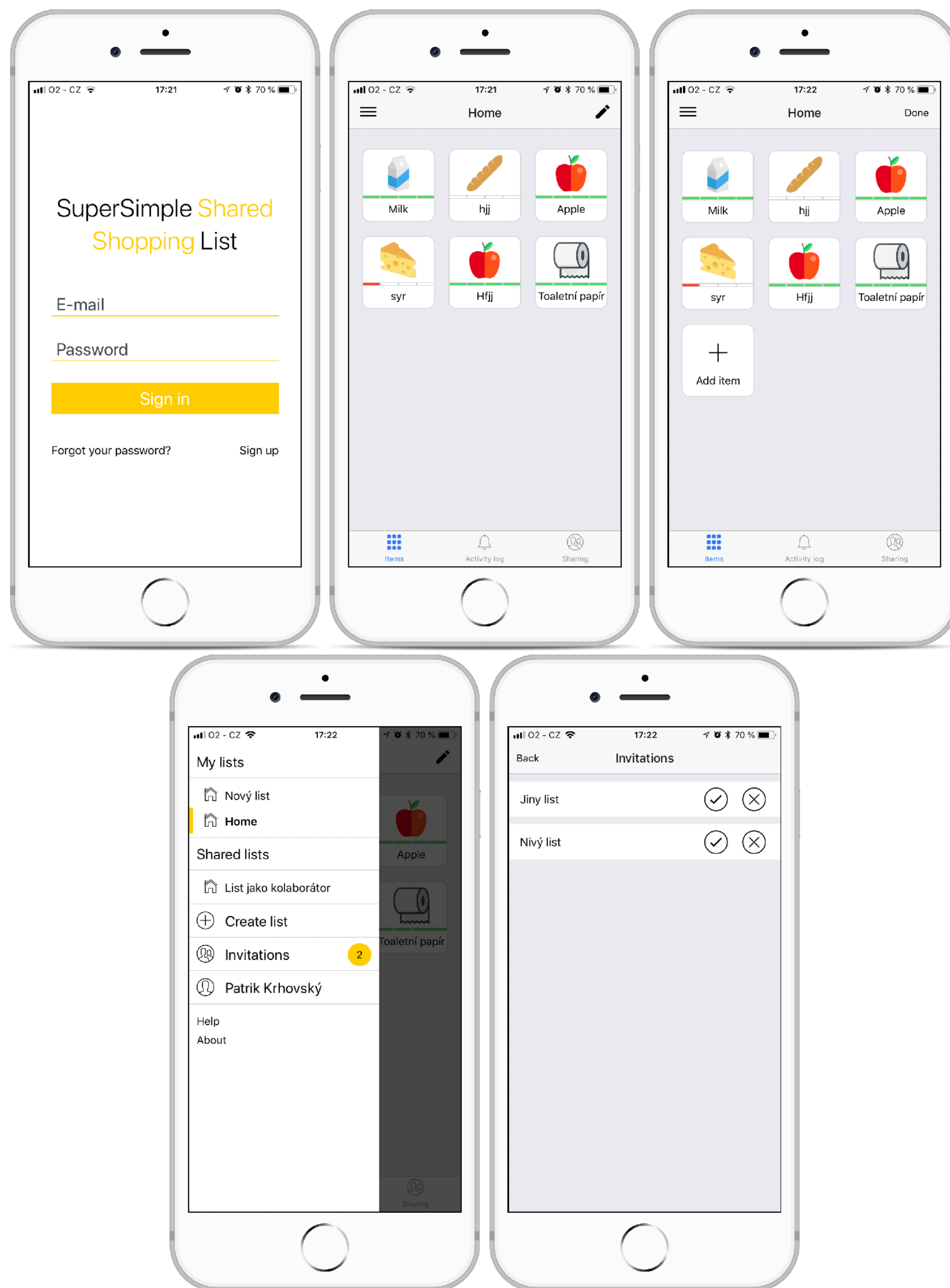
6.1 Nultá verze aplikace

Cílem nulté verze aplikace bylo hlavně otestování dostupných technologií, otestování návrhu komunikace popsánem v kapitole 4.3 a taky otestování vytvořené serverové aplikace. Aplikace byla implementována podle nakreslených wireframů, neměla ale žádný předem navrhnutý vzhled, což se nakonec ukázalo jako osudné. Z toho důvodu totiž nemohl být vytvořen prototyp a při testování bylo zjištěno příliš mnoho chyb. Tím se vývoj celé aplikace protáhl. Výsledek uživatelského rozhraní nulté verze lze vidět na obrázku 6.1.

Aplikace byla nainstalována do telefonů celkem 10 testerům ve věku od 18 do 45 let. 6 testerů používalo systém Android, zbylí 4 testeři systém iOS. Testeři byli seznámeni s cílem mobilní aplikace a následně byli požádáni o vykonání akcí podle předem nachystaného scénáře. Scénář byl zaměřen na postupné projítí celou funkcionalitou aplikace. Reakce testerů byli zapisovány a následně analyzovány. Byly zjištěny následující problémy:

- Nepochopení účelu aplikace – většina uživatelů na začátku nepochopila účel aplikace a používali ji jako nákupní seznam. Bylo způsobeno hlavně názvem aplikace, který v ten moment byl shodný s názvem bakalářské práce. Následně byla aplikace přejmenována na *Vitalist - not a grocery list.* Upraven byl taky popis aplikace.
- Nemožnost sdílení seznamu při vytváření
- Editační režim vypadá stejně jako zobrazení detailu seznamu a uživatel poté neví, kde se právě nachází
- Možnost přesouvat položky na systému Android nefunguje správně, přesouvá se při tom mezi záložkami
- Přidání nové položky je schováno v editačním režimu
- Příliš mnoho stavů položky

- Plochy, které reagují na kliknutí, jsou malé



Obrázek 6.1: Uživatelské rozhraní v nulté verzi. Lze si všimnou, že rozdíl mezi 2 a 3 obrazovkou jsou minimální a při tom je v třetí obrazovce zapnutý editační režim, která obsahuje i tlačítko pro přidání položky

6.2 Testování prototypu verze 1.0

Cílem další verze aplikace bylo odstranit nedostatky, které se zjistili na základě prvního testování. Byl vytvořen vzhled všech scén, které bude aplikace obsahovat, pomocí aplikace Sketch[1], která je dostupná pro zařízení se systémem macOS. Takto vytvořené scény byly naimportovány do aplikace InVision[2], pomocí které byl vytvořen prototyp aplikace. Díky vytvořenému prototypu mohlo být uživatelské rozhraní ihned otestováno bez nutnosti jeho implementace do mobilní aplikace. Jeho vývoj byl tudiž podstatně rychlejší než v nulté verzi.

Mezi nedostatky, které byly v této verzi odstraněny, patří například:

- možnost sdílet list při jeho vytváření
- obtékající tlačítko pro přidání nové položky, které je nově pořád dostupné
- položka má pouze 3 stavy - plný, dochází a prázdný

Tyto změny jsou obsaženy i ve finálním návrhu, viz. obrázky 4.3 a 4.4.

Testování bylo prováděno na stejném vzorku lidí jako první testování, aby bylo zjištěno, zda byly odstraněny nalezené chyby. K tomuto vzorku bylo přidáno ještě 5 nových testerů, kvůli zachování objektivnosti, jelikož testeři předchozí verze již znali funkcionalitu a problémy, které se měly odstranit. Z toho 2 testeři byli dlouhodobí uživatelé systému iOS a 3 testeři byli dlouhodobí uživatelé systému Android. Testováním byly odhaleny 2 nedostatky:

- Uživatelé nemohli najít funkcionalitu přejmenování a odstranění seznamu
- 3 uživatele nenapadlo kliknout na záložku s kolaborátory pro sdílení listu

6.3 Testování aplikace verze 1.0

Při implementaci vytvořeného prototypu bylo potřeba odstranit nedostatky, které byly objeveny v prototypu. První se týkal funkcionality přejmenování nebo odstranění seznamu. Použití této funkcionality bylo možné z multiplatformní komponenty `ActionSheet`[3], která se objevila po dlouhém podržení titulku seznamu v menu. Tato funkce je znázorněna na obrázku 6.3. Ukázalo se, že 10 uživatelů z 15 nedokázalo seznam přejmenovat ani odstranit, z nichž šest testerů byli uživatelé systému Android. Po navedení uživatelů na otevření nabídky se navíc ukázalo, že uživatelé systému Android jsou překvapení komponentou `ActionSheet`, která není běžně na systémech Android používána.

Tento problém byl odstraněn přidáním této funkcionality pod tlačítko, které se nachází v navigačním panelu při zobrazení seznamu. Po kliknutí na něj je otevřena pro uživatele systému iOS nabídka v komponentě `ActionSheet`. Pro uživatele systému Android je to komponenta `Material Menu`. Druhý nedostatek, týkající se sdílení seznamu, byl odstraněn přidáním tlačítka pro sdílení seznamu do komponenty, která byla popsána v předchozím odstavci. Toto finální řešení je zobrazeno na obrázku 4.5.

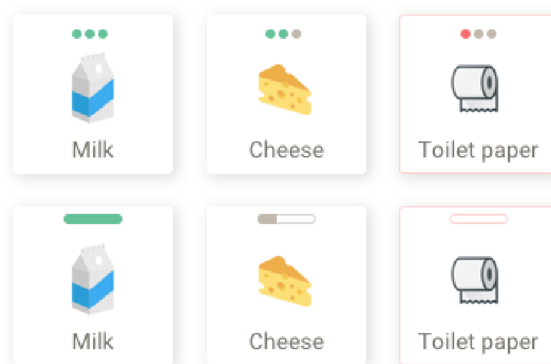
Po implementaci finálního uživatelského rozhraní bylo spuštěno otevřené beta testování na Google Play. Na systémech iOS bylo spuštěno testování pro externí testery. Při testování

[1] Sketch – https://www.sketchapp.com/

[2] InVision – http://invisionapp.com/

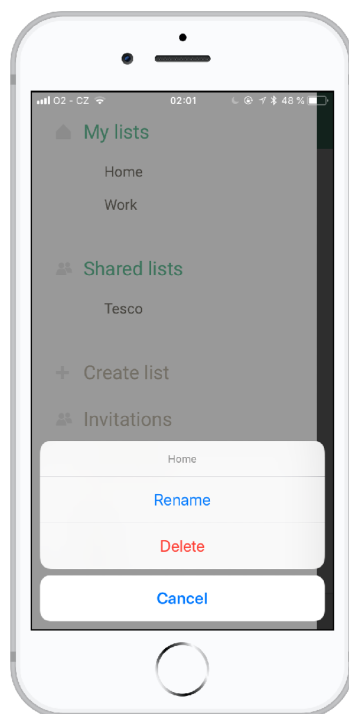
[3] Pro multiplatformní `ActionSheet` byla použita knihovna https://github.com/beefe/react-native-actionsheet, která na zařízeních se systémem iOS zobrazuje stejnojmennou nativní komponentu a na zařízeních se systémem Android stylově podobnou komponentu pomocí modal boxu.

aplikace byl ještě nalezen problém s komponentou, která zobrazuje aktuální stav položky. Komponenta byla tvořena 3 kuličky, kde každá z nich znázorňuje jeden stav. Při posledním stavu ale byla poslední kulička stále vyplněna a uživatelé čekali, že dalším kliknutím bude stav teprve plně prázdný, tudíž očekávali 4 stavy položky. Problém a jeho řešení je zobrazen na obrázku 6.2.



Obrázek 6.2: Rozdíl mezi starou a nově vytvořenou komponentou pro zobrazování aktuálního stavu položky

Po spuštění beta testování si aplikace již našla své první uživatele. Aktivnější jsou uživatelé systému iOS, kteří s ní stráví denně průměrně 10 minut. Naopak uživatelé systému Android v ní stráví denně průměrně o 5 minut méně a navíc nejsou denně aktivní. Denní statistiky jsou zobrazeny na obrázku 6.4.



Obrázek 6.3: Komponenta `ActionSheet` byla použita pro systém Android i iOS a vyjížděla po dlouhém podržení titulku seznamu

Obrázek 6.4: První graf znázorňuje používání iOS aplikace za období od 1. do 13. května. Na druhém grafu jsou statistiky aplikace pro Android za stejné období. Lze vidět, že nejaktivnější jsou uživatelé vždy v době zveřejnění nové verze aplikace

6.4 Zveřejnění aplikace verze 1.0

Po odstranění všech nalezených chyb, které byli zjištěny spuštěním beta testování, byla aplikace zveřejněna v obchodě Google Play a Apple App Store. Společně byla spuštěna i propagační webová prezentace[4].

Do aplikace byla implementována služba *Firebase Crashlytics*[5], která v případě pádu aplikace odešle chybové hlášení a pomáhá tak k rychlejšímu vyřešení problému.

[4]Webová prezentace Vitalist: http://getvitalist.com
[5]Firebase Crashlytics: https://firebase.google.com/docs/crashlytics/

Kapitola 7

Závěr

Tato práce popisuje implementaci mobilní aplikace pro systém Android a iOS, díky které mohou uživatele sdílet seznam denně potřebných věcí s ostatními členy domácnosti. Nejprve bylo navrženo uživatelské rozhraní a taky způsob komunikace mobilní aplikace se serverovou aplikací. Následně byla implementována jak serverová, tak mobilní aplikace. Mobilní aplikace byla dále testována na reálných uživatelích. Na základě výsledků testů byla iterativně vylepšována. Aplikace byla zvěřejněna v obchodě Google Play a Apple App Store pod názvem Vitalist. K její propagaci byla vytvořena taky webová prezentace.

Díky výsledné aplikaci lze vytvořit seznam a ten následně plně spravovat a sdílet s dalšími uživateli. Všechna tyto data jsou synchronizována v reálném čase a uživatel má tedy k dispozici vždy nejnovější informace. Při případném pozvání do seznamu se odešle emailová zpráva, a to i uživateli, který ještě nemá vytvořený účet. Po stažení aplikace a následné registraci, bude automaticky přidán do seznamu, do kterého byl pozván. Díky této metodě lze získat nové uživatele. Při pozvání již registrovaného zákazníka je navíc odesílána i notifikace do mobilního zařízení. Notifikace se dále odesílají v případě, že někdo v seznamu změnil stav položky na stav *prázdná*.

Kdybych začínal nyní, dal bych si více záležet na testování uživatelského rozhraní před jeho implementací díky prototypu aplikace. Kvůli chybám v nulté verzi se práce hodně zpomalila, jelikož se uživatelské rozhraní zásadně předělávalo. V budoucnu je potřeba upravit uživatelské rozhraní pro tablety. V plánu je taky určitě přidat offline režim a možnosti nastavení práv kolaborátorů, aby měli například možnost editovat položky v seznamu. Samozřejmostí je iterativní zlepšování na základě nových hodnocení uživatelů. Aplikace byla na obchodech zveřejněna těsně před odevzdáním bakalářské práce a její zhodnocení z pohledu uživatelů bude teprve prováděno.

Literatura

[1] *Cloud Firestore documentation.* [Online; navštíveno 03.05.2018].
URL https://firebase.google.com/docs/firestore/

[2] *Cloud Messaging documentation.* [Online; navštíveno 03.05.2018].
URL https://firebase.google.com/docs/cloud-messaging/

[3] *Don't Block the Event Loop (or the Worker Pool).* [Online; navštíveno 09.05.2018].
URL https://nodejs.org/en/docs/guides/dont-block-the-event-loop/

[4] *Firebase.* [Online; navštíveno 03.05.2018].
URL https://firebase.google.com/

[5] *Good UI - Try Undos instead of prompting for confirmation.* [Online; navštíveno 03.05.2018].
URL https://goodui.org/#8

[6] *Immutable Map.* [Online; navštíveno 03.05.2018].
URL https://facebook.github.io/immutable-js/docs/#/Map

[7] *Redux documentation.* [Online; navštíveno 03.05.2018].
URL https://redux.js.org

[8] *Redux Saga documentation.* [Online; navštíveno 03.05.2018].
URL https://redux-saga.js.org/docs

[9] *Understanding the Node.js event loop.* Říjen 2015, [Online; navštíveno 09.05.2018].
URL
http://abdelraoof.com/blog/2015/10/28/understanding-nodejs-event-loop/

[10] Developer: *Web Worker(s) – Outsource your JavaScript. Medium.com*, Březen 2012.

[11] Evkoski, B.: *React Native: What it is and how it works. Medium.com*, Červen 2017.

[12] Gartner: *Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2017.* Srpen 2017, [Online; navštíveno 02.05.2018].
URL https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[13] Inc., F.: *Flux documentation.* [Online; navštíveno 03.05.2018].
URL https://facebook.github.io/flux/docs

[14] Inc., F.: *React Native documentation.* [Online; navštíveno 03.05.2018].
URL https://facebook.github.io/react-native/docs

[15] Krug, S.: *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability (Web Design Courses)*. New Riders, 2013, ISBN 978-0321965516.

[16] labs, G.: *A Beginners Guide to Redux*. [Online; navštíveno 03.05.2018]. URL http://www.gistia.com/beginners-guide-redux/

[17] Mandrigin, I.: *Optimistic UIs in under 1000 words. Medium.com*, Prosinec 2016.

[18] Pangsakulyanont, T.: *Immutable.js, persistent data structures and structural sharing. Medium.com*, Prosinec 2016.

[19] Wheeler, K.: *Getting To Know Flux, the React.js Architecture. Scotch.io*, Červenec 2016.

Příloha A

Obsah přiloženého CD

- `vitalist-api/` - zdrojové soubory serverové aplikace
- `vitalist-emails/` - zdrojové soubory emailových šablon
- `vitalist-mobile/` - zdrojové soubory mobilní aplikace
- `vitalist-thesis/` - zdrojové soubory této technické zprávy
- `vitalist-web/` - zdrojové soubory webové prezentace
- `thesis.pdf` - technická zpráva ve formátu PDF
- `poster.pdf` - plakát pro prezentování aplikace
- `video.mp4` - video pro prezentování aplikace