Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Vývoj mobilní aplikace v React Native

Diplomová práce

Autor: Kristýna Kamenická

Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Pavel Kříž, Ph.D.

Hradec Králové srpen 2018

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracovala samostatně a s použitím uvedené literatury.

V Hradci Králové dne 13. 8. 2018 Kristýna Kamenická

Poděkování:

Děkuji vedoucímu své diplomové práce Ing. Pavlu Křížovi, Ph.D. za odborné vedení práce a užitečné rady v průběhu jejího zpracovávání.

Anotace

Následující text se zabývá nalezením nového multiplatformního řešení pro existující aplikaci eObchodník. Práce obsahuje teoretickou a praktickou část.
V teoretické části práce je popsán vývoj moderních mobilních aplikací a ostatních technologií používaných k vývoji hybridních aplikací. Dále byly popsány jednotlivé knihovny a nástroje, které byly následovně aplikovány při vývoji.
Praktická část popisuje průběh vývoje nové aplikace od komplexní analýzy, implementací jednotlivých předem určených případů použití až po generování produkční verze aplikace. Dále jsou popsány nástroje a standardy, které byly použity při vlastním vývoji k řešení problémů a k usnadnění vývoje. Práce by měla rovněž pomoci budoucím vývojářům pochopit použité technologie a implementované funkce.

Annotation

Title: Development of mobile application in React Native

Following text describes finding of new cross-platform solution for existing mobile application eObchodnik. The thesis contains main two parts – theoretical and practical.
The theoretical part describes development of modern mobile applications and briefly summarize available technologies for hybrid app development. There is also description of important frameworks, tools and libraries that were used during the development of the new solution.
The practical part describes whole process of development from complex analysis, implementation of specified use-cases to a generating of production version. The thesis should also help future developers to understand used technologies and implemented features.

Obsah

1 Úvod 1
2 Cíl práce 2
3 Moderní vývoj mobilních aplikací 3
3.1 Nativní vývoj 4
3.2 Multiplatformní vývoj 5
4 Hybridní aplikace 8
4.1 Motivace 9
4.2 Principy a přístupy 9
4.3 Úskalí multiplatformního vývoje 10
4.3.1 Výkon a optimalizace 10
5 Dostupné technologie 11
5.1 Native script 11
5.2 Xamarin Platform 12
5.3 Apache Cordova 13
6 React Native 14
6.1 React 14
6.1.1 Komponentární přístup, JSX 14
6.1.1 Props 15
6.1.1 State 15
6.1.2 Práce s DOM 16
6.1.3 Optimalizace ReactJS 18
6.2 Architektura React Native 18
6.1 Nativní knihovny 19
6.1.1 Proces sestavení 20
6.1.2 Varianty sestavení 21

6.2 Redux 22
6.3 Vzhled a stylování 28
6.4 NodeJS 28
7 Stávající řešení 29
7.1 Předpoklady ke změně přístupu 29
7.2 Výběr případů užití 29
7.3 Analýza 31
8 Vlastní řešení 34
8.1 Nastavení prostředí a nástroje 34
8.2 Rozvržení aplikace 34
8.3 Implementace 37
8.3.1 Inicializace projektu 37
8.3.2 Spuštění aplikace 38
8.3.1 Obecné standardy aplikace 40
8.3.2 Vzhled aplikace 41
8.3.3 Navigace 43
8.3.4 Práce s nativními moduly 48
8.3.5 Lokální ukládání dat 48
8.3.6 Přístup k informacím o zařízení 51
8.3.7 Snímání fotografií a post-processing 53
8.3.8 Networking 56
8.3.9 Lokalizace 59
8.3.10 Integrace s Google Maps 59
8.3.11 Odeslání SMS a vytočení telefonního čísla 63
8.3.12 Ramda a Ramda Extension 63
8.4 Optimalizace 64

8.5 Produkční verze aplikace ... 66
8.6 Ladění ... 69
8.6.1 React DevTools ... 72
8.7 Continuous integration ... 72
8.8 Testování ... 73
9 Shrnutí výsledků ... 74
10 Závěry a doporučení ... 75
11 Seznam použité literatury ... 76
Seznam příloh ... 80

Seznam obrázků

Obrázek 1 Základní typy mobilních aplikací ... 4
Obrázek 2 Architektura hybridních aplikací ... 9
Obrázek 3 Procesy vykreslení v ReactJS ... 17
Obrázek 4 Architektura React Native ... 19
Obrázek 5 Use Case Diagram ... 31
Obrázek 6 Struktura aplikace ... 35
Obrázek 7 Struktura store ... 36
Obrázek 8 Data flow diagram ... 37
Obrázek 9 Struktura obrazovek aplikace ... 44
Obrázek 10 Struktura navigačních komponent ... 45
Obrázek 11 Schéma databáze ... 50
Obrázek 12 Povolení použití polohy uživatele (RN Android, RN iOS) ... 52
Obrázek 13 Indikátor aktivity ukládání fotografie ... 55
Obrázek 14 Clustery (nativní Android vlevo, RN vpravo) ... 61
Obrázek 15 Autocomplete s místy podle vzdálenosti ... 62
Obrázek 16 Networking ... 71

Seznam tabulek

Tabulka 1 Možnosti podle typu aplikace ... 7

Seznam grafů

Graf 1 Podíl OS na trhu pro jednotlivé kvartály ... 5

Seznam ukázek kódu

Ukázka kódu 1 Ukázka JSX ... 15
Ukázka kódu 2 Action creator ... 24
Ukázka kódu 3 Reducer ... 25
Ukázka kódu 4 Vytvoření store ... 26
Ukázka kódu 5 Provider ... 26
Ukázka kódu 6 Connect ... 27

Ukázka kódu 7 gradle.build 38
Ukázka kódu 8 Ukázka definice stylu 42
Ukázka kódu 9 Definice barev 42
Ukázka kódu 10 Zdroj bitmapy 43
Ukázka kódu 11 Definice stránky navigace v routes.js 44
Ukázka kódu 12 Definice obrazovek a parametrů 45
Ukázka kódu 13 HOC 46
Ukázka kódu 14 Filtrování skrytých obrazovek 47
Ukázka kódu 15 Přepsání výchozí funkce HW tlačítka 48
Ukázka kódu 16 Model prohlídky 49
Ukázka kódu 17 Načtení pozice zařízení 52
Ukázka kódu 18 NetInfo 53
Ukázka kódu 19 Porovnání nastavení a zdroje připojení aplikace 53
Ukázka kódu 20 Sejmutí fotografie 54
Ukázka kódu 21 Vnoření elementu 56
Ukázka kódu 22 czgDa požadavek 57
Ukázka kódu 23 Čtení binárních dat z lokálního úložiště 57
Ukázka kódu 24 Odeslání prohlídek 58
Ukázka kódu 25 Lokalizace 59
Ukázka kódu 26 Vyhledávání na mapě 62
Ukázka kódu 27 Použití knihovny Reselect 65
Ukázka kódu 28 Konfigurace release 66
Ukázka kódu 29 Konfigurace Reactotron 70
Ukázka kódu 30 Konfigurace pipeline v Bitbucket 73

Pozn.: Pokud není u obrázku či ukázky kódu uveden zdroj jedná se o vlastní zpracování.

1 Úvod

V posledních letech prošla mobilní zařízení z pohledu využitelnosti významnou změnou. Zařízení jsou v mnoha ohledech chápána jako primární zdroj informací pro své uživatele z důvodu snadné dostupnosti. Počet aplikací, které využití takového zařízení značně rozšiřují, v posledních letech několikanásobně vzrostl.

Standardem webových stránek je mít optimalizovanou – responzivní verzi pro uživatele mobilních zařízení. Velmi často se stává, že poměr těchto uživatelů výrazně převyšuje ty, kteří přistupují z osobního počítače. Počet typů operačních systémů používaných napříč mobilními zařízeními se snížil a ustálil. Tento fakt umožnil rozvoj nových přístupů, které jsou konkrétněji zaměřené na tyto platformy – nejčastěji tedy iOS a Android.

Pro poskytovatele služeb a produktů se mobilní aplikace staly nedílnou součástí business strategie. Vytváření aplikace, která by měla být podporována mobilními zařízeními, není snadný úkol. Spektrum technologií a přístupů je široké a vidina možnosti ušetření nákladů spojeného s vývojem lákavá. Vytváření aplikace v mladé technologii může mít v mnoha příkladech nejisté výsledky. Je proto více důležitá příprava a správně uchopená analýza požadavků.

Tato práce si klade za cíl vyzkoušet zcela nové řešení pro již existující aplikaci. Motivací společnosti je hlavně ušetření času stráveného na vývoji a jeho přesunutí do interního týmu.

Práce popisuje možnosti provedení správné analýzy požadavků aplikace a představuje nejvíce používané technologie a principy. Čtenář bude postupně seznámen s principy React Native a dalších technologií, které jsou nezbytné pro správné uchopení dané problematiky. Klíčovou částí práce je popis vlastní implementace jednotlivých požadavků ve výsledné aplikaci a popsání úskalí, kterým bylo při vývoji nutné čelit.

2 Cíl práce

Cílem práce je zanalyzovat stávající řešení nativní aplikace eObchodník a navrhnout nové multiplatformní řešení. Výsledkem pak bude ukázková aplikace, která bude obsahovat a prezentovat funkční řešení vybraných kritických případů užití původní aplikace. Práce bude sloužit jako funkční i technická dokumentace, která bude popisovat řešení jednotlivých částí aplikace a bude sloužit jako primární podklad k případnému dalšímu vývoji.

Práce bude také obsahovat nezávislé zhodnocení všech přínosů a omezení, které by pro objednatele vyvstaly, pokud by se vydal touto cestou vývoje. Finální rozhodnutí o úspěšnosti a vhodnosti řešení bude na obchodním oddělení dané společnosti.

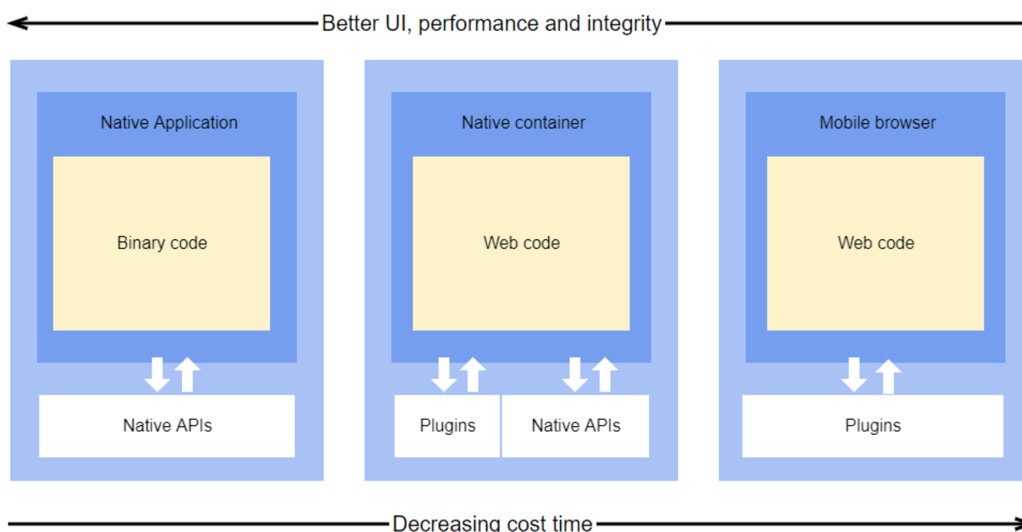
3 Moderní vývoj mobilních aplikací

V posledních letech jsou zadavatelé a vývojáři mobilních aplikací postaveni před několik možností, jak k implementaci požadované aplikace přistupovat. Nalezení optimálního řešení není snadný úkol. Je proto vhodné před samotným vývojem provést analýzu právě těchto možností. Správné rozhodnutí v počátcích může později ušetřit velké množství vynaloženého času a financí. Výsledek této analýzy pak závisí na velkém množství aspektů, jako jsou např.:

- účel aplikace,
- cílová skupina uživatelů,
- způsob distribuce,
- náročnost,
- dostatečná seniorita v týmu.

Zpravidla neexistuje pouze jedno správné řešení, ale výsledek analýzy může pomoci s výběrem nejvhodnějšího způsobu. Typy mobilních aplikací lze v současnosti rozdělit na tři základní kategorie – nativní, hybridní a webové. Jednoduchá schémata jednotlivých architektur jsou znázorněna na obrázku č. 1. Hybridní a webové aplikace se také souhrnně označují jako multiplatformní, protože je možné je vyvíjet pro více platforem najednou. Principy jednotlivých přístupů jsou vysvětleny v následujících podkapitolách.

Vývojové nástroje (SDK) pro jednotlivé platformy poskytují celou řadu nativních API, přes které je možné přistupovat k nativním komponentám zařízení (fotoaparát, senzory pohybu atp.). Možnost využívání těchto API je u různých přístupů rozdílná. Na obrázku č. 1 je znázorněno, že s nativním vývojem zpravidla roste celková rychlost vytvářené aplikace, lepší využitelnost nativních UI prvků zařízení a lepší integrita. Na druhou stranu multiplatformní přístupy přinášejí redukci času stráveného vývojem.

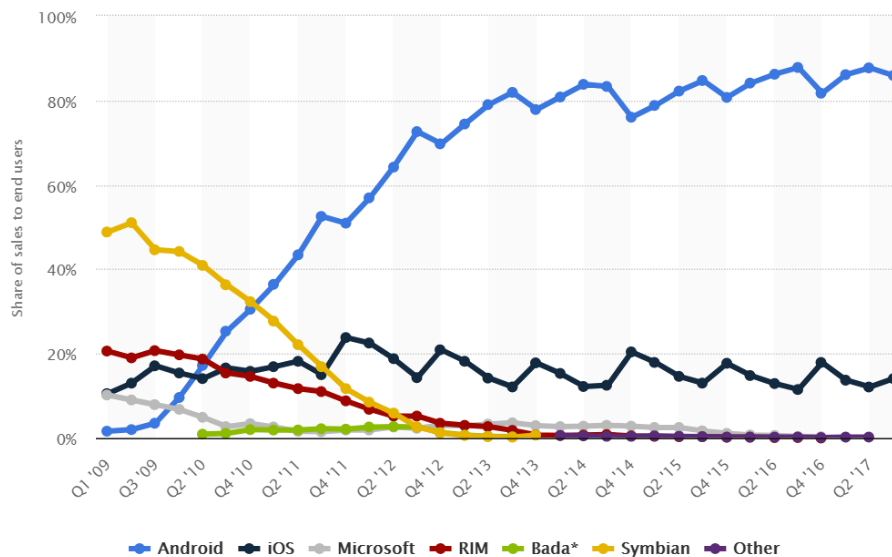


Obrázek 1 Základní typy mobilních aplikací
Zdroj: Vlastní přepracování podle [1] a [2]

K usnadnění analýzy připravované aplikace může také pomoci využití již připravovaného nástroje, jako je například ten představený v následujícím článku [2]. Autoři rozebírají několik pohledů a způsobů, jak získat přehled o úskalích plánované aplikace. Konkrétní analýza potom probíhá v několika krocích. Prvním krokem je vyplnění dotazníku o obecných požadavcích na aplikaci a následné ohodnocení jejich důležitosti (publikace v obchodě, off-line mód apod.). Následně jsou vybrány požadované funkcionality zařízení a senzory. Výsledkem je pak poměrný přehled vhodnosti jednotlivých přístupů

3.1 *Nativní vývoj*

Nativní vývoj je často považován jako standardní přístup k vytváření mobilní aplikace. Jedná se o vytváření na sebe nezávislých aplikací s využitím nástrojů a technologií pro příslušnou platformu. Z grafu č. 1 je patrné, že v současné době dominují trhu chytrých mobilních zařízení dvě platformy – iOS (Apple) a Android (Google Inc.). Aplikace pro tyto systémy jsou zpravidla distribuovány pomocí online obchodů a v současné době jejich počty dosahují několika miliónů.



Graf 1 Podíl OS na trhu pro jednotlivé kvartály
Zdroj: [3]

Hlavní výhodou je kompilace do nativního jazyka spouštěného přímo na zařízení, což zajišťuje rychlost odezvy, která může být u jiných přístupů problematická. Dalším benefitem je i přímý přístup k nativním API a práci s hardwarovými součástmi zařízení. Jedná se pak zpravidla o aplikace využívající společnou serverovou část, se kterou komunikují přes rozhraní. Mnoho společností proto k vývoji přistupuje tak, že velkou část aplikační logiky řeší na serveru a klientská aplikace slouží pouze jako tzv. lehký klient (thin client), spíše tedy k zpracovávání vstupů a zobrazení výsledných dat. Pokud je ale aplikace takové povahy, kdy je nezbytné držet velké množství logiky v klientu, může nastat problém. Vývoj nových částí aplikace a změn pak musí proběhnout na všech podporovaných platformách zvlášť. Z technologického hlediska je pak často potřeba najít vývojáře, kteří se na danou platformu specializují. Tato skutečnost se pak odráží na celkové organizaci projektu a jeho finančních nákladech.

3.2 *Multiplatformní vývoj*

Multiplatformní vývoj, jak už název napovídá, je způsob vytváření jedné aplikace, která je cílena na více platforem. Hlavním benefitem pak je, že je určitá část implementace logiky aplikace společná a zároveň je možné přistupovat k nativním

funkcím[1] konkrétní platformy. Toho lze dosáhnout využitím nástrojů, které se do určité míry o konkrétní interpretaci nativních funkcionalit starají a často poskytují jednotné rozhraní.

Multiplatformní řešení není vhodné pro všechny typy aplikací. Ne zřídka se proto stává, že pokus o takové řešení bývá neúspěšný. Zajímavý a komplexní pohled na tuto problematiku poskytla nedávno společnost Airbnb, která v roce 2016 přešla na multiplatformní přístup u vytvářených mobilních aplikací. Po dvou letech používání React Native však vedení společnosti usoudilo, že touto cestou již v budoucnu jít nechtějí. Tento návrat autor následujícího článku zdůvodňuje hlavně tím, že s použitím RN nebyla společnost schopna dosáhnout svých specifických cílů z pohledu požadavků na aplikaci i organizační struktury celé společnosti. Na druhou stranu dodává, že Airbnb tato zkušenost pomohla porozumět některým úskalím vyvíjených aplikací a přispěla k vyjasnění způsobu budoucího vývoje. [4] Toto je nepochybně důkaz toho, že i přes veškeré benefity, které multiplatformní řešení přinášejí, stále existuje i celá řada problémů, kterým je nutné čelit.

Hybridní aplikace

Mobilní aplikace, které využívají obvykle webové technologie a zároveň do určité míry přistupují k nativním funkcím zařízení, se nazývají hybridní aplikace. K nativním funkcím je zpravidla pak přistupováno pomocí tzv. přemostění (*bridge)*. Stručný přehled dostupných nástrojů a jejich základní principy jsou zpracovány v 7. kapitole. [5]

Webové aplikace a responzivní webové stránky

Za předpokladu, že zamýšlená aplikace bude minimálně využívat nativní zdroje zařízení a není nutné aplikaci distribuovat pomocí obchodu, je responzivní webová aplikace vhodným řešením. Nejčastěji se bude jednat o aplikace sloužící primárně k zobrazení obsahu. Aplikace bude pak jednoduše dostupná přes kterýkoliv webový prohlížeč na příslušné adrese. Aplikace je zpravidla vytvářena pomocí HTML5, JavaScript a CSS. Problém webových aplikací je značné omezení přístupu k nativním

---

[1] Fotoaparát, GPS, apod.

funkcím zařízení, proto nejsou vhodné pro složitější aplikace, pro které je například nezbytné získávat data ze senzorů zařízení. Dalším problémem je nutnost připojení zařízení k internetu, na jehož rychlosti může být celkový výkon aplikace velmi závislý. [6]

Velkou škálu dalších možností přinášejí progresivní webové aplikace (WPA). WPA aplikace je progresivním rozšířením běžné webové aplikace ať už se jedná o *single-page* aplikaci (SPA) nebo *multi-page* stránku. Přináší s sebou možnosti jako ikona na ploše telefonu, *push-notifikace*, off-line mód nebo přístup do lokálního úložiště telefonu. [7]

V tabulce č. 1 je v jednoduchosti znázorněna dostupnost klíčových možností aplikace pro jednotlivé způsoby implementace.

	Dostupnost zařízení (nativní API)	**Rychlost**	**Náročnost vývoje**	**Možná publikace v obchodě[2]**	**Schvalovací proces**
Nativní	Plná	Velmi rychlá	Velmi náročný	Ano	Povinný
Hybridní	Plná	Možná nativní rychlost	Středně náročný	Ano	Podle technologie
Webové	Částečná	Rychlá	Středně náročný	Ne	Žádný

Tabulka 1 Možnosti podle typu aplikace
Zdroj: Vlastní přepracování podle [1]

[2] Obchod s aplikacemi

4 Hybridní aplikace

Vývoj pro více platforem obecně směřuje k tomu najít rovnováhu mezi dvěma základními požadavky – maximalizace jednotného řešení požadavků a dojem koncového uživatele. To znamená snahu ze strany vývoje docílit co největšího společného *codebase*[3], který je znovupoužitelný pro podporované platformy za předpokladu, že jsme schopní koncovému uživateli poskytnout stejně kvalitní zážitek jako s nativní aplikací.

Dojem koncového uživatele je primárním ukazatelem toho, zda je aplikace úspěšná, či nikoliv. Ať už se jedná o rozsáhlou komerční aplikaci nebo jenom jednoduchý nástroj, mělo by být v zájmu vývojáře poskytnout uživateli co nejlepší zážitek po vizuální a funkční stránce. Otázka hodnocení koncového uživatele je často negativně zmiňována právě v kontextu multiplatformního vývoje. Touto problematikou se zabývá následující článek [8]. Autoři analyzovali a srovnávali hodnocení uživatelů nejčastěji používaných aplikací, které přešly z nativního vývoje na hybridní. Většinou se jedná o delší časová období, ze kterých byla data čerpána, ale na druhou stranu počet použitých aplikací v porovnání je nízký, což může vést ke zkreslení výsledků. I přes to se jedná o zajímavý způsob, jak spokojenost uživatelů ověřit. Hodnocení uživatelů bylo klasifikováno a hodnoceno z pohledu výkonu, relevance, bezpečnosti a použitelnosti aplikace. Podle výsledků bylo hodnocení u hybridní verze aplikace pro OS Android spíše negativní, kromě hodnocení bezpečnosti, které vyšlo lépe než u nativní verze. U iOS byly výsledky spíše nekonzistentní a ukazovaly na přínos hlavně z hlediska výkonu a použitelnosti aplikace u hybridní varianty. Podobným problémem se také zabývá i následující práce. [9]

Často je pro obecný pojem multiplatformního vývoje využíván slogan *write-once, run everywhere/anywhere*[4], v kontextu mobilního vývoje však ne úplně koresponduje se aktuálně dostupnými možnostmi. Často jsou při vývoji využity nativní

---

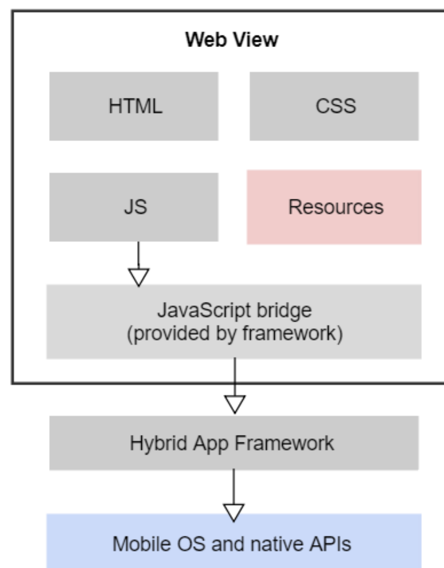
[3] Veškerý zdrojový kód aplikace
[4] Autor – Sun Microsystem (Java) .

API jednotlivých platforem, velikost znovu použitelné části potom závisí na tom, do jaké míry daná platforma poskytuje společná rozhraní.

4.1 *Motivace*

Hlavním motivačním faktorem pro vytváření hybridních aplikací je ušetření času stráveného při vývoji i při rozšiřování a udržování aplikace. Vývoj hybridní aplikace pak může pro objednavatele znamenat výrazné ušetření celkových financí za projekt. Koncovému uživateli nezáleží na tom, jaká technologie byla použita, ale na jeho koncovém dojmu z aplikace, kterou má jednoduše dostupnou. Tento aspekt je jedním z nejvíce diskutovaných – jak poskytnout uživateli stejný bezchybný zážitek jako s nativní aplikací a zároveň ušetřit značné množství financí při vývoji? Zážitek koncového uživatele by tedy měl být co nejméně rušen možnými negativními projevy aplikace.

4.2 *Principy a přístupy*



Obrázek 2 Architektura hybridních aplikací
Zdroj: Přepracování podle [2]

Přístupy k vývoji hybridních aplikací se liší podle použité technologie. Základní architektura je však pro všechny podobná, viz obrázek č. 2. Jednotlivá řešení poskytují rozhraní pro práci s nativními funkcemi zařízení, které jsou do určité míry společné

pro jednotlivé platformy. Pro vytváření uživatelského rozhraní jsou zpravidla použity stejné technologie jako při vývoji webových aplikací. Definice jednotlivých UI prvků jsou potom povětšinou společné všem podporovaným platformám. Výhodou je možnost standardní distribuce aplikace do obchodů příslušných platforem. Aplikace je nejčastěji spouštěna pomocí nativního kontejneru. Struktura tohoto kontejneru se potom nijak neliší od struktury klasické aplikace psané nativně.[6]

4.3 *Úskalí multiplatformního vývoje*

S multiplatformním vývojem se pojí řada problému, kterým musí vývojář v průběhu vytváření aplikace čelit. Největším problémem je celkový dojem z aplikace z pohledu výkonu a také rozdílnost napříč zařízeními. Dobře zpracovaný přehled obecných problémů s multiplatformním vývojem, který zahrnuje celý proces vývoje, zpracovává následující článek [10].

4.3.1 Výkon a optimalizace

V kontextu multiplatformního vývoje je otázka výkonu často zmiňována. Zpravidla každá technologie poskytuje obsáhlou dokumentaci s doporučeními na implementaci jednotlivých problémových domén aplikace. Pokud jsou tyto postupy dodržovány, nemělo by docházet k fatálním chybám vedoucím k znatelným problémům spojených s úbytkem výkonu aplikace. Jednotlivé problémy lze pak rozdělit do následujících kategorií:

- nedostatek CPU a paměti,
- nadměrná konzumace baterie a čerpání dat. [11]

Obecně platí, že by aplikace měly být již od začátku vytvářeny s ohledem na optimalizaci jednotlivých částí. Z důvodu možného výskytu problémů s výkonem, ale i z prevenčního hlediska lze využít metodu profilace, která dokáže měřit a následně analyzovat jednotlivé úkony aplikace. Nástroj k profilaci se nazývá profiler. Výsledkem profilace je pak přehled o časové náročnosti jednotlivých procesů, zátěži procesoru, využívání paměti apod. Lze poté snadněji vyčíst, v jakých částech má aplikace z pohledu optimalizace rezervy.

Doporučuje se spouštět profilaci na aplikaci běžící na reálném zařízení, nikoliv pouze na simulátoru, protože by výsledky mohly být výrazně zkreslené. *Profiler* je většinou nástroj na obecnou analýzu, existují však i specializované nástroje, které se zaměřují pouze na určitou množinu problémů. Například k analýze konzumace baterie aplikace, která je způsobena zobrazováním prvků na obrazovce telefonu apod. Řešení problémů s výkonem lze shrnout do několika bodů:

- použití profilace k analýze případných problémů,
- optimalizace používaných bitmap,
- korektní management vláken aplikace,
- využívání speciálního garbage collectoru,
- omezit prováděné akce, pokud je aplikace na pozadí. [12]

5 Dostupné technologie

Následující kapitola stručně představuje dostupné technologie pro vývoj hybridních aplikací.

5.1 *Native script*

Jedním ze zajímavých nástrojů pro tvorbu mobilních aplikací napříč platformami je open-source projekt NativeScript (NS), který byl uveden v r. 2015 společností Progress Software Corporation. [13] NS poskytuje řešení pro vytváření mobilní aplikace pro platformy iOS a Android pomocí Angular 2, TypeScript nebo JavaScript. Angular je řešení používané pro front-end webových aplikací. Veškeré informace o NS byly čerpány z oficiální dokumentace. [14]

Nástroje

Výhodou je možnost používání balíčků dostupných prostřednictvím *npm*[5]. Jednotlivé obrazovky aplikace jsou reprezentovány jako stránky pomocí elementu *Page*. Ke tvorbě uživatelského rozhraní (UI) jednotlivých stránek lze využít již hotové komponenty, které NS poskytuje. K ladění aplikace lze uplatnit vývojářské

[5] Správce balíčků pro NodeJS

nástroje prohlížeče Google Chrome. Pro jednotkové testy můžeme použít rozšířené frameworky, jako jsou Jasmine nebo Mocha.

Výkon a optimalizace

K optimalizaci výkonu aplikace lze využít balíčkovací nástroj *(bundler)* jako je například Webpack a nástroj UglifyJS. Dalším optimalizačním prvkem pro Android je V8 snapshot builds, který je součástí engine[6], na němž je NS Android spouštěn. Na základě dokumentace lze zrychlit čas spouštění aplikace až o několik sekund. [15]

Shrnutí

NS je vhodný především pro vývojáře, kteří již mají zkušenosti s vývojem webových aplikací pomocí Angular, CSS a nemají možnost investovat větší množství času do učení nové technologie.

5.2 *Xamarin Platform*

Dalším z již dostupných a poměrně rozšířených frameworků je Xamarin Platform, který vznikl v rode 2011 jako komerční projekt a je od roku 2016 podporovaný společností Microsoft. [16] Xamarin aktuálně podporuje platformy iOS, Android a Windows Phone. Primárně lze použít objektový programovací jazyk C# pro sdílené části a nativní moduly pro Windows Phone, sekundárně pak standardně Swift pro nativní iOS a Java pro specifické části nativního Android. [17]

Nástroje

Důležitým nástrojem, který poskytuje Xamarin, je knihovna Xamarin Forms, která poskytuje unifikované rozhraní k vizuální i funkční implementaci grafického uživatelského rozhraní aplikace (GUI), které je společné napříč platformami. K implementaci jednotlivých prvků UI jsou využívány elementy, které jsou pak následně mapovány na nativní ekvivalent. [18]

[6] V8 Engine vytvořený Google Inc.

Implementace nativních funkcionalit, jako je např. správa událostí v aplikaci, využívání senzorů a využívání multimediálních nástrojů je realizována pomocí rozhraní, která jsou potřeba implementovat zvlášť pro každou platformu. Technologie pak dále poskytuje výhody spojené s použitím programovacího jazyka C# verze 6, poskytování možnosti jednoduše provádět asynchronní operace, možnost vytvářet balíčky NuGet, které lze následně využít napříč platformami, lokalizaci textací aplikace anebo zabezpečení komunikace pomocí vrstvy TSL. Xamarin dále poskytuje možnost využití cloudových a datových služeb např. Microsoft Azure.

Výkon a optimalizace

K optimalizaci výkonu aplikace lze využít nástroje Xamarin Profiler, který dokáže celkově zanalyzovat aplikaci a vybrat problémy, které se vztahují k výkonu aplikace. Xamarin Profiler je součástí vývojového prostředí (IDE) Visual Studio. Dále pak využít návrhový vzor Disposable, který je implementován pomocí rozhraní IDisposable v C#, F# a Visual Basic, jenž dává možnost vynuceného uvolnění alokovaných zdrojů. [19]

5.3 *Apache Cordova*

Tento open source projekt, známý také jako *PhoneGap,* byl uveden společností *Nitobi* a následně zakoupen Adobe, které ho pak v roce 2011 uveřejnila pod nynějším názvem. *Apache* Cordova je řešení, které z pohledu podporovaných platforem nabízí asi nevětší výběr 2 – *iOS, OS X, Android, Blackberry 10, Windows a Ubuntu*. Aplikace jsou vytvářeny pomocí webových technologií HTML5, CSS3 a JavaScript za použití nativních rozšíření. (Cordova Plugins) [20]

Shrnutí

Řešení je především vhodné pro publikaci webových aplikací, které cílí na více platforem ve specifických obchodech s tím, že bude aplikace schopna využívat některé nativní funkce zařízení. Z důvodu velkého množství podporovaných platforem je pak řešení co nejvíce abstrahováno od konkrétní platformy.

6 React Native

React Native (RN) je poměrně nová knihovna, která slouží k vytváření mobilních aplikací pro operační systémy iOS a Android. RN je založen na knihovně ReactJS, která je velmi populární ve vývoji webových aplikací. Pro vývojáře, který s RN začíná, je důležité porozumět základním konceptům ReactJS, které jsou vysvětleny v následujících podkapitolách.

6.1 *React*

React je JavaScript je deklarativní knihovna k vytváření uživatelského rozhraní webových aplikací založená na komponentním přístupu, která byla uvedena v r. 2010 společností Facebook Inc. ReactJS i React Native jsou publikován jako součást Facebook Open Source. [21]

6.1.1 Komponentární přístup, JSX

Jednotlivé stránky aplikace jsou tvořeny pomocí elementů, které jsou do sebe zanořovány, jako tomu je např. u HTML tagů. Pro jednodušší práci s elementy ReactJS poskytuje syntaxi JSX, která je rozšířením běžného formátu JS, avšak její použití není podmínkou. Definice vzhledu a základní zpracování logiky událostí jde nepochybně ruku v ruce, proto je právě JSX způsob, jak přehledně udržet tyto definice na jednom místně. Tyto jednotky kódu jsou pak nazývány komponentami.

Komponenty by měly být zpravidla vytvářeny jako na sobě nezávislé části, které by měly implementovat jasně definovanou část UI a případné aplikační logiky. Obecně jsou rozlišovány dva základní druhy komponent – funkcionální a třídové. Chování a vzhled komponenty zpravidla udávají vstupní data, která se definují jako atributy (*props*) jednotlivých elementů. Ukázka kódu č. 1 ukazuje strukturu jednoduché třídové komponenty, která reprezentuje profil uživatele a zobrazuje základní informace, které přijímá v props.

```
class Profile extends React.Component {
  render () {
     const { name, surname, imageUrl } = this.props;
     return (
       <PersonalInfo name={name} surname={surname}>
           <ProfileImage url={profileUrl} />
       </PersonalInfo>
          )
        }
}
```

Ukázka kódu 1 Ukázka JSX

6.1.1 Props

Props jsou primárním zdrojem dat veškerých komponent. Do vnořených komponent jsou zasílány, stejně jako tomu je u definice atributů HTML elementu. React reaguje na veškeré změny v datech v *props* překreslením komponenty. Je doporučeno komponentě dávat k dispozici pouze ta data, která jsou reálně použita.

Funkcionální komponenty by měly být definovány jako čisté[7] funkce a také by se měl vývojář snažit o maximální zobecnění a přepoužitelnost.

6.1.1 State

Třídové komponenty oproti funkcionálním mohou držet vlastní stav a mají svůj životní cyklus, na jehož jednotlivé fáze lze v komponentě v případě potřeby zareagovat. Důležitá fáze životního cyklu komponenty, jejíž definice je povinná, je metoda `render`. V třídových komponentách je možné definovat konstruktor, který slouží k inicializaci stavu komponenty. Změna stavu komponenty má za následek vyvolání její překreslení. Lze také manipulovat s instancí samotné komponenty pomocí `this`. Stavové komponenty mají své výhody, ale pokud to není nutné, jsou vhodnější komponenty funkcionální.

[7] Pro jeden vstup pokaždé stejný výsledek, nepoužívání *side-effects*

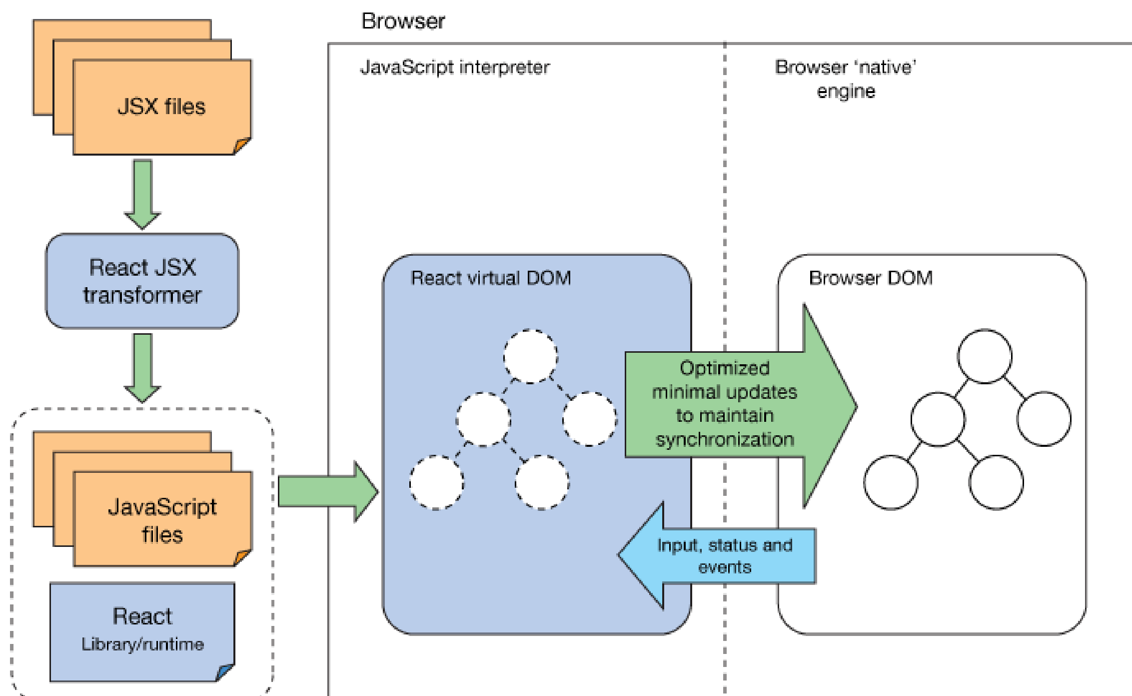
6.1.2 Práce s DOM

Při načítání webové stránky prohlížeč vytváří tzv. DOM (*Document Object Model*). DOM je definovaným standardem (W3C) pro manipulaci s dokumenty stránky. DOM je tvořen stromovou strukturou objektů stránky, kde každý objekt znázorňuje HTML element. Programovací jazyk, jako je např. JavaScript, je pak schopen pracovat s jednotlivými objekty stromu a vytvářet tak dynamické HTML [22].

ReactJS disponuje vlastní optimalizovanou implementací DOM systému pro efektivní práci s elementy, která je nezávislá na použitém webovém prohlížeči. React má své specifické atributy, které fungují odlišně do klasického HTML. Podrobný popis lze nalézt v oficiální dokumentaci. [23] V aplikaci používající ReactJS by měl být zpravidla definován kořenový element, na který bude směrováno jako na výchozí bod aplikace (*entry point*). Tím aplikaci sdělujeme, že s veškerými podřazenými elementy bude manipulovat React DOM.

Každý element musí projít několika fázemi, než se dostane od svého zaregistrování ve stromu až po jeho zobrazení. Pokaždé, když v DOM proběhne jakákoliv změna, musí se element změnit ve stromové struktuře. To, že je DOM organizován právě do stromové struktury, přináší výkonnostní benefity. Tato událost se ale vyvolá i pro všechny zanořené elementy, a to je právě v situaci, kdy aplikace nejvíce ztrácí na výkonu.

ReactJS aplikace provádí build komponentního stromu, se kterým je manipulováno jako s virtuálním DOM. Tento virtuální DOM funguje jako optimalizační článek mezi aplikací a DOM. ReactJS reaguje na změny tím, že si komponenty, které změny vyvolají, označí jako *dirty*. Následně na těchto komponentách probíhá *rebuild*, jehož následkem jsou veškeré tyto komponenty překresleny do *virtualDOM*. Důležitou částí procesu je procházení a porovnávání jednotlivých elementů s DOM a rozhodování, který element se musí překreslit *(Reconciliation process)*. Na obrázku č. 3 je znázorněno schéma celého procesu.



Obrázek 3 Procesy vykreslení v ReactJS
Zdroj: [24]

Změny v jednotlivých elementech jsou způsobeny nastavením nového stavu komponenty nebo obdržením nových *props*. Tento proces porovnávání změn lze optimalizovat využíváním metod životního cyklu komponenty. Pokud takové metody nejsou nalezeny, použije se výchozí rozhodovací proces, který porovná props. ReactJS vyniká svým výkonem právě díky tomuto procesu. [25]

Změny ve struktuře jsou vyvolávány na základě událostí. Jednotlivé události mohou být vyvolány uživatelem i samotným prohlížečem. ReactJS tyto nativní události obaluje vlastní tzv. umělou událostí (*synthetic event*). Funkce, ve kterých se zpravidla tyto události později zpracovávají, obdrží jako argument objekt umělé události. Tento objekt je pak z optimalizačních důvodů dále nulizován a opět použit v následující události. [25] Konkrétní akce, které na jednotlivé události reagují, jsou definovány jako funkce v třídových komponentách anebo vstupují do komponenty pomocí props jako *callback*, který lze chápat jako funkci, jejíž logika je definována mimo komponentu. Provedení takové funkce nastane až ve chvíli obdržení konkrétních parametrů z události.

6.1.3 Optimalizace ReactJS

Primárním problémem s výkonem aplikace psané v ReactJS je ve většině případů špatné uchopení práce s daty a jejich předávání v rámci jednotlivých komponent. Optimalizaci nadměrného překreslování jednotlivých komponent lze provést implementací metody životního cyklu *shouldComponentUpdate*, kde lze implementovat vlastní porovnávací mechanizmus. V případě nutnosti použít třídové komponenty lze místo React.Component použít React.PureComponent, která již tuto metodu implementuje a porovnává *props* pomocí *shallowCompare* (porovnání referencí).

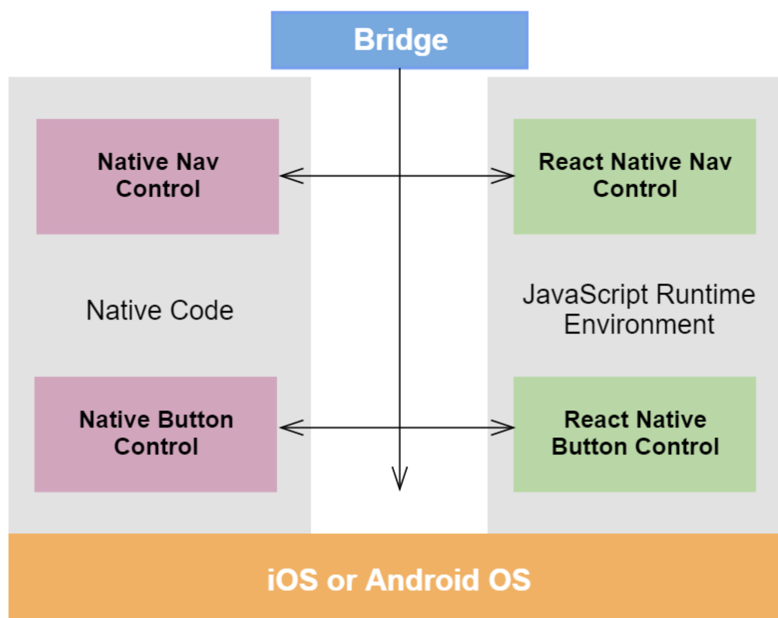
Ke zrychlení vytvářené aplikace je dále obecně doporučeno používat produkční minifikovaný (redukovaný) build. ReactJS je často doporučeno používat společně s balíčkovacím frameworkem např. *Webpack*, který se postará o vytvoření a minifikaci finálního balíčku aplikace. Výsledný balíček aplikačního kódu se nazývá *bundle*.

V neposlední řadě je přínosné věnovat pozornost varováním v konzoli, která se v aplikaci vyskytují. Nemusí to být přímo chyby, které by aplikaci nějak ohrozily, ale je doporučeno je neignorovat a řešit. V opačném případě by časem mohl nastat rozsáhlejší problém. [26]

6.2 *Architektura React Native*

Následující kapitola byla zpracována s pomocí oficiální dokumentace RN a následujícího článku. [27] [28] Aplikace vytvářená v RN poskytuje možnost provádět nativní kód aplikace a zároveň kód napsaný ve skriptovacím jazyku JavaScript na mobilním zařízení. Zařízení používající iOS nebo Android neumí standardně nativně procesovat jazyk primárně využívaný k webovému vývoji. Je proto nezbytné mít k dispozici běhové prostředí, ve kterém bude skriptovací jazyk prováděn. Ve standardním nativním vývoji využívá aplikace primárně jedno vlákno, které se stará o běh aplikace. RN využívá hlavní nativní vlákno v nativním jádru aplikace i vlákno výše zmíněného běhového prostředí *JavaScriptCore*. *JSCore* běží přímo nad operačním systémem zařízení. Příslušná vlákna těchto dvou modulů běží současně, přičemž je umožněna oboustranná komunikace. Obě vlákna na sobě nejsou výkonem závislá. Nemůže proto nastat situace, kdy by náročnější výpočet logiky v JS zbrzdil hlavní nativní vlákno.

Komunikační entita mezi jednotlivými vlákny se nazývá *bridge*. Ve skutečnosti je komunikace prováděna mezi konkrétními komponentami nativního modulu a jejich alternativ v JS modulu tak, jako tomu je na obrázku č. 4. Znamená to tedy, že i když RN umožňuje vytvářet aplikaci v JS, je nezbytné, aby každá část byla podložena korespondující nativní částí, se kterou bude navázána komunikace k zajištění její správné činnosti pro konkrétní podporovanou platformu.



Obrázek 4 Architektura React Native
Zdroj: [28]

6.1 *Nativní knihovny*

Základní RN poskytuje celou řadu nativních modulů pro práci s nativními API obou platforem. Nativní API si lze představit jako rozhraní mezi JS a konkrétním nativním kódem (modulem). Takový modul je pak např. *NetInfo*, který poskytuje informace o připojení zařízení do sítě. RN je primárně koncipován tak, aby v případě potřeby využití jiného API existovala snadná možnost, jak aplikaci o tento modul rozšířit. V současné chvíli je RN komunita natolik rozsáhlá, že ty nejčastěji využívané případy užití jsou vývojářům k dispozici právě prostřednictvím konkrétního nativního modulu. Tyto moduly jsou často vytvářeny jako open-source pod křídly velkých společností, např. react-native-maps od Airbnb [29].

Každý modul, který bude v aplikaci využíván, je potřeba standardně nainstalovat pomocí npm a dále propojit s React aplikací. Linkování lze provést ručně anebo pomocí příkazu `react-native link <název modulu>`. Po provedení příkazu dojde k propojení obou nativních částí aplikace s dříve nainstalovaným modulem. Jednotlivé moduly jsou instalovány standardně do *node_modules*. Správné propojení (link) lze ověřit úspěšným sestavením aplikace, nativní modul by měl být součástí objektu *NativeModules* poskytovaného balíčkem *react-native.* Pro iOS je možné nativní moduly instalovat pomocí CocoaPods. Jednotlivé závislosti jsou potom definovány v souboru Podfile. Tento soubor generuje automaticky příkaz `pod init`. Následně je potřeba provést `pod install`.

6.1.1 Proces sestavení

Proces sestavení aplikace probíhá v několika krocích. Pro spuštění aplikace na zařízení je nejprve potřeba sestavit nativní kontejner, který bude odkazovat na sestavený *bundle* aplikace. Tento *bundle* je potom následovně spuštěn pomocí běhového prostředí *JSCore*.

Za předpokladu, že aplikaci spouštíme za účelem vývoje, je na vývojovém zařízení lokálně spouštěn balíčkovací server *Metro Bundler* pomocí *NodeJS*, který výsledný bundle publikuje tak, aby si jej nativní kontejner měl možnost stáhnout a spustit. [30] Bundle je standardně publikován na adrese *http://localhost:8081/index.android.bundle* (event. ios). Hlavním benefitem při vývoji v RN je právě tento moment, kdy při úpravách JS části aplikace není potřeba nové sestavení nativního kontejneru, ale je pouze nutné stáhnout aktuální bundle. Proces vytváření nového bundle a jeho publikace probíhají automaticky lokálně při každé změně a uložení jakékoliv části JS kódu. Zařízení, které se na server dotazuje, buď čeká na manuální načtení balíčku pomocí možnosti *Reload* anebo v případě zapnuté možnosti *Hot Reload* načítá nový balíček ihned po jeho publikaci. [28]

Sestavení aplikace pro Android, která bude nezávislá na *Metro Bundler*, lze provést tak, že spustíme sestavení projektu přímo pomocí nástroje *Gradle*, který je standardně využíván pro sestavování nativních aplikací pro platformu Android. Sestavení pak probíhá podobně akorát s tím rozdílem, že bundle je přímo součástí

instalované aplikace a nativní kontejner potom odkazuje na lokální soubor v mobilním zařízení. Vytvoření bundle a generování nativního kontejneru lze provádět separovaně (stejně jako ve starších verzích RN). Výsledek takového sestavení je potom instalační soubor apk, který lze standardní cestou do zařízení nainstalovat.

Obdobně tento proces probíhá pro iOS pomocí generování aplikace pro produkční účely (release) v IDE Xcode. Následně je nutné změnit parametr *jsCodeLocation* v *AppDelegate.m*, který definuje cestu k bundle.

Pro Android

Nativní kontejner aplikace je sestavován pomocí nástroje Gradle. Projekt má multimodulární charakter za předpokladu, že jsou využity nativní knihovny (mimo standardní knihovny obsažené v RN). Při sestavení kontejneru jsou postupně sestavovány jednotlivé moduly. RN je vytvářen tak, aby vývojáři maximálně ulehčil konfigurace Gradle projektu. Konfigurační soubor hlavního modulu *build.gradle* je generovaný s veškerým popisem jednotlivých konfiguračních parametrů.
Za předpokladu standardního vygenerování nové aplikace je pouze nutné nastavit cílový adresář, do kterého Gradle pomocí sestavení zkompiluje potřebné soubory a zkomprimuje je do *apk*.

Pro iOS

Kontejner aplikace je sestavován pomocí nástrojů poskytovaných v Xcode. Závislosti na jednotlivé nativní moduly jsou automaticky rozeznány při inicializaci projektu. Sestavení aplikace potom taktéž probíhá pomocí Xcode. Způsob konfigurace se proto výrazně neliší od nativního vývoje pro iOS. IDE automaticky rozeznává projekt psaný v RN a spouští *Metro Bundler* během sestavování. Pokud není připojeno žádné zařízení spouští výchozí iPhone simulátor.

6.1.2 Varianty sestavení

Jak pro vytváření balíčku, ale i pro sestavení nativního kontejneru jsou k dispozici dva základní módy – *debug* a *release*.

Sestavení bundle pro debug mód zabraňuje minifikaci balíčku a nabízí potom v aplikaci vývojářské menu – *Dev Options*, kde je možné procházet jednotlivé elementy

aplikace, nastavit lokální server a další ladící nástroje. Na rychlost aplikace má debug mód sestavení negativní dopad. V případě zvolení možnosti *Debug JS remotely* probíhá spuštění JS balíčku aplikace přímo v prohlížeči Google Chrome, a to pouze tehdy, pokud je zapnutý mód vzdáleného ladění. [31] K aktivování tohoto módu následuje několik kroků, které musí balíčkovací nástroj provést a umožnit tak vývojáři ladit kód pomocí ladicích nástrojů prohlížeče. Prvním signálem je požadavek, který je vyslaný ze strany zařízení o aktivaci. Tento požadavek zpracuje balíčkovací nástroj, což má za následek otevření Google Chrome prohlížeče se záložkou na adrese *http://localhost:8081/debugger-ui* a načtením statického HTML souboru.

Dalším krokem je vytvoření komunikačního kanálu, mezi prohlížečem, balíčkovacím nástrojem a následně i zařízením pomocí *web socket*. Navázání komunikace se zařízením předchází několik korků, ve kterých je nutné vytvořit relaci pro komunikaci. Po vytvoření ověřené relace vyžádá zařízení skript přímo do prohlížeče. Jednotlivá volání, která probíhají na pozadí během procesu vzdáleného ladění, je možné pozorovat v záložce *Network* v *DevTools*. Prohlížeč následně pracuje se skriptem totožně, jako tomu je u webových aplikací. Načte si skript a vloží ho jako součást statického DOM připravené webové stránky. Ve chvíli spuštění tohoto skriptu jsou k dispozici standartní nástroje ladění v prohlížeči. [32]

Gradle poskytuje také tyto dva módy sestavení pomocí *assembleDebug* a *assembleRelease*. Pokud je balíček vytvářen v rámci sestavení kontejneru, je použit stejný mód pro *bundle*. V Xcode jsou tyto módy již přednastaveny.

6.2 *Redux*

Velké množství moderních webových aplikací je vytvářeno jako *single-page-aplication* (SPA). Taková aplikace reaguje na vstupy uživatele tak, že místo vykreslování nových stránek dynamicky překresluje prvky v rámci jediné stránky, čímž zaručuje plynulejší odezvu. Následující kapitola čerpá z oficiální dokumentace Redux [33].

Redux vznikl jako reakce na vzrůstající nároky SPA hlavně z pohledu dat, která musí aplikace za svého „běhu“ držet. Takovými daty mohou být např. informace o přihlášeném uživateli, vytvořené objednávce apod. Při vývoji mobilních aplikací pomocí RN je taktéž vhodné Redux, ke správě stavu aplikace, použít.

Základní koncept *Redux* je poměrně jednoduchý. Veškerá data aplikace jsou společně nazývána stavem aplikace, který je uchováván ve strukturovaném objektu zvaném *store*. Aktuální stav aplikace výrazně ovlivňuje to, jak se aplikace chová a vypadá. S každou změnou v aplikaci je vytvářen stav nový. Důležitou skutečností je to, že ke stavu aplikace můžeme libovolně přistupovat, avšak nikdy ne ho přímo měnit. Jediný způsob, jak změnit stav aplikace je pomocí akcí. Tento koncept nám zajistí přehled o tom, v jakém okamžiku byl stav aplikace změněn a přehled přesné podobě dané změny. V oficiální dokumentaci Redux jsou uvedeny tři základní principy, které knihovnu vystihují:

1. jediný zdroj pravdy *(Single source of truth),*
2. stav je pouze ke čtení *(State is read-only),*
3. změny jsou vytvářeny pomocí čistých funkcí *(Changes are made with pure functions).*

První princip deklaruje, že veškeré informace, ze kterých komponenty čerpají, by měly pocházet pouze z jednoho zdroje – globální stav. To zaručuje přehled o tom, odkud komponenta čerpá svá data, čímž se předchází neočekávanému chování.
Druhý princip je již zmíněná restrikce změn stavu. S každou změnou je vytvořen nový obraz stavu následujícího. Jednotlivé kroky jsou pak uchovány a lze se mezi nimi pohybovat v obou směrech. V praxi k tomu přehledně slouží rozšíření pro vývojářskou konzoli prohlížeče Google Chrome, které dokáže zobrazovat jednotlivé akce v pořadí, v jakém byly vyvolány. Záznamy pak uchovávají informaci o identifikátoru, parametrech vyvolané akce, původním stavu a novém stavu.

Akce

Akce je událost vyvolaná v aplikaci za účelem změny stavu aplikace. Každá akce, jejíž předpis má být vykonán, musí mít specifikovaný jedinečný identifikátor, podle kterého lze pak rozhodnout, jakým způsobem bude stav aplikace změněn. Předpisy

změn jsou definovány v rámci funkce zvané *reducer*. Akce je napříč aplikací poslána pomocí metody *dispatch*. Funkce, která zpracovává vstupy uživatele a volá konkrétní `dispatch`, se nazývá *ActionCreator,* viz ukázku kódu č. 2. Do každé akce lze specifikovat libovolné množství dalších parametrů (v ukázce `userId`).

```
const createUserAction = (id) => {
  // action id
  const CREATE_USER = 'CREATE_USER';
  // action
  return {
   type: CREATE_USER,
   userId: id,
  };
}

const createUser = (dispatch, getState) => {
  const id = getId(getstate());
  dispatch(createUserAction(6));
}
```

Ukázka kódu 2 Action creator

Reducer

Reducer je funkce, která má na vstupu akci, aktuální stav a vrací nový stav. Tento stav je pouze derivát celkového stavu aplikace, který spadá pod příslušný *reducer*. Běžná aplikace jich má několik a jsou kombinovány do výsledné podoby *store* aplikace pomocí funkce *combineReducers*. Rozdělení konkrétního *store* aplikace pak ve většině případů koresponduje s komunikačním rozhraním aplikace, které je nezbytné si správně rozvrhnout jako minimální stromovou reprezentaci stavu aplikace.

Výsledek z *reducer* by měl být snadno predikovatelný a testovatelný, je proto zapotřebí *reducer* psát jako čistou funkci, viz ukázku kódu č. 3. Do každého *reducer* vstupuje aktuální stav aplikace a akce. Objekt akce potom zahrnuje identifikátor a popř. vlastní vstupní parametry. Objekt stavu aplikace, který je na vstupu, by neměl být, jakkoliv mutován. Je vhodné používat funkce s konceptem *immutable*[8].

[8] V případě změny dat navrací celý nový objekt, původní instance zůstává nedotčená

Pokud není k identifikátoru nalezeno příslušné zpracování, vrací *reducer* původní stav aplikace.

```
const initialState = {
     userInfo: {
          authorized: false;
    }
}
function userReducer(state = initialState, action){
     switch (action.type) {
             case CREATE_USER:
                     // nový objekt stavu
                     return ...
             default:
                     return state;
    }
}
```

Ukázka kódu 3 Reducer

Middleware

Dalším užitečnými nástroji Redux jsou *middleware*. Jsou to rozšíření třetích stran, která lze zakomponovat do procesu změn stavu aplikace. Výhodou je skutečnost, že se mohou jednotlivé middleware řetězit a tím jednoduše využít více rozšíření najednou. Takové rozšíření potom pracuje s vyslanou akcí těsně před tím, než ji převezme *reducer*. Pokud je řetězeno několik rozšíření za sebe, každé další potom pracuje s výsledkem přechozí modifikace akce. K nejrozšířenějším *middlware* nepochybně patří podpora logování akcí a aktuálního stavu aplikace, reportování chyb a řízení akcí asynchronně. [33]

Redux poskytuje funkci `applyMiddleware`, která jako argumenty akceptuje jednotlivé middleware a je použita jako druhý argument metody `createStore` při inicializaci store aplikace. Konfigurace a konkrétní využití jednotlivých middleware bude popsáno v praktické části práce.

React Native-Redux

Koncept React-Redux je s použitím Redux v RN totožný. V následujícím odstavci bude popsáno obecné použití kombinace React-Redux, které čerpá z oficiálního dokumentačního souboru React-Redux. [34]

Vytvoření a inicializace stavu aplikace je provedeno pomocí metody `createStore` viz ukázku kódu č. 4.

```
import { applyMiddleware, compose, createStore } from 'redux';
import thunk from 'redux-thunk';
import logger from 'redux-logger';
import { userReducer, orderReducer } from '../reducers';

const reducers = combineReducers({
  userInfo: userReducer,
  orders: orderReducer,
});

export default createStore(
  reducers,
  compose(applyMiddleware(thunk, logger)),
);
```

Ukázka kódu 4 Vytvoření store

Jako první parametr metoda přijímá reducer (popř. kombinace více reducerů) a do druhého nepovinného atributu lze definovat požadované middleware, které se mají aplikovat. V ukázce kódu č. 4 jsou použity dva různé reducery a middleware *redux-thunk* a *redux-logger*. Výsledek je pak předán do komponenty `Provider` do atributu `store` viz ukázku kódu č. 5. Provider, je součástí balíčku *react-redux*.

```
export default () => {
  return (
    <Provider store={store}>
      <Root />
    </Provider>
  );
};
```

Ukázka kódu 5 Provider

Tato komponenta zajišťuje, že budou mít podřazené komponenty přístup ke stavu. *Root* je kořenová komponenta aplikace. Za předpokladu, že chceme mít přístup ke stavu aplikace ze všech komponent, je potřeba tuto komponentu tímto elementem obalit.

Jak už bylo zmíněno, aplikace psaná v ReactJS je členěna do komponent. V kombinaci s Redux jsou pak specifikovány dva základní druhy komponent – prezentační komponenty a kontejnery. Jejich rozdíl spočívá v tom, zda komponenta přistupuje přímo ke globálnímu stavu aplikace (kontejner přistupuje, komponenta nikoliv).

Zdrojem dat prezentační komponenty jsou pouze její *props* nebo *state*. Kontejner má navíc přístup k *dispatch* a stavu aplikace pomocí metody *connect*. *Connect* funguje jako *high-order-function* (dekorátor), který požadovaná data nebo *dispatch* namapuje na *props* vnořené komponenty, viz ukázku kódu č. 6. Kontejnery potom slouží jako zdroje dat pro zanořené prezentační komponenty.

```
const mapDispatchToProps = dispatch => ({
  ...bindActionCreators(
    { createUser },
    dispatch,
  ),
});

const mapStateToProps = state => ({
// without selector, just illustrative
  userName: state.userInfo.userName,
});

export default connect(mapStateToProps, mapDispatchToProps)(Profile);
```

Ukázka kódu 6 Connect

Metoda `connect` může mít více parametrů, pro běžné účely postačí dva – `mapStateToProps` a `mapDispatchToProps`. Prvním parametrem je funkce, která definuje, že daná komponenta bude odebírat stav aplikace (stane se z ní tzv. *subscriber stavu*). Vstupními parametry `mapStateToProps` jsou aktuální stav aplikace a vlastní *props (state, ownProps).* Tato funkce bude pak provolána pokaždé, když bude stav aplikace jakýmkoliv způsoben změněn. Přímo v metodě pak bude vyhodnoceno, zda je potřeba danou komponentu překreslit, či ne. Metoda následně vrací objekt, který se předá do *props* obalované komponenty. V ukázce kódu č. 6 je potom konkrétně do props přidáno `userName`, se kterým komponenta může dále pracovat. Konkrétní mapování hodnoty se označuje jako selekce.

MapDispatchToProps slouží k mapování akce jako objektu obalená funkcí s *dispatch*, který je následně přístupná v *props* k přímému provolání. Využívá k tomu metodu *bindActionCreators* (*react-redux*), která veškeré vstupní *action creatory* provolá s *dispatch* a při jejich následném vyvolání poskytne tuto funkci jako vstupní parametr společně se stavem aplikace, jako tomu je v ukázce kódu č. 2.

Metoda *connect* přijímá další dva argumenty, které jsou používány spíše zřídka. *MergeProps* příjmá *props* připojené komponenty včetně *props* z předešlých dvou funkcí a poskytuje možnost dalších modifikací props, které vstoupí do komponenty. Dalším argumentem jsou pak *options*, do kterých je možné definovat specifické požadavky na komponentu. Např. *areStatesEqual* specifikuje způsob porovnání aktuálního a nového stavu aplikace. [35]

6.3 *Vzhled a stylování*

K sestavování jednotlivých obrazovek a jejich následnému stylování React Native používá Flexbox, který se primárně využívá k webovému vývoji. Implementace Flexbox pro RN má však několik rozdílů, jako je např. odlišná výchozí hodnota pro orientaci vnořených prvků (*flex-oriantation*). Flexbox je použit hlavně z důvodu schopnosti přizpůsobení různým rozlišením koncových zařízení. V oficiální dokumentaci lze nalézt přehledný seznam veškerých atributů a jejich vysvětlení. K seznámení se s Flexbox lze využít mnoho dostupných tutoriálů, které mohou značně pomoci správnému sestavování komponent. [36]

Každá komponenta poskytovaná balíčkem *react-native* přijímá parametr style, díky němuž je možné specifikovat požadovaný styl komponenty. Některé mají i atributy pro přímou specifikaci hodnoty (nejčastěji barvy). Jednotlivé atributy mají zpravidla stejné pojmenování, jako tomu je ve standardním CSS, s tím rozdílem, že víceslovné atributy nejsou odděleny pomlčkou, ale psány v *camelCase*.

6.4 *NodeJS*

Velkým přínosem RN je správa JS modulu pomocí NodeJS, což přináší jednoduchost a rychlost. NodeJS je sestavovací nástroj, který slouží pro aplikace psané v JavaScript. Projekt se skládá ze závislostí dostupných prostřednictvím *node package manager* (npm).

Vývojář má k dispozici všechny balíčky dostupné na npm. Po inicializaci projektu je vygenerována základní struktura včetně *package.json* s definovanými závislostmi pro RN. Instalaci samotného projektu je možné provést pomocí *npm install* nebo *yarn*.

7 Stávající řešení

Aplikace eObchodník je nativní mobilní aplikací, která je psána pro platformy Android a iOS. Aplikace slouží pojišťovacím agentům primárně k pořizování fotodokumentace na místě pojistné události a také jako informační podpora.

7.1 *Předpoklady ke změně přístupu*

V současné době je aplikace vyvíjena zvlášť dvěma externími týmy pro jednotlivé platformy. Komunikace a zadávání požadavků musí probíhat separovaně, což není pro společnost ideální z časového i finančního hlediska.

Společnost se zaměřuje primárně na vývoj portálových a webových aplikací se zajištěním podpory, které mají na starosti interní týmy i externí dodavatelé. V roce 2016 započal aktivní vývoj v interním týmu na společném frameworku, který úspěšně slouží k unifikaci front-end částí nového vývoje v rámci různých projektů. Tento framework byl pro firmu revoluční zejména pro použité technologie a celkovou změnu pohledu na vývoj z hlediska kvality a přepoužitelnosti. I přes obavy spojené se zavedením nových technologií byl start projektu úspěšný. Se vzrůstající oblíbeností technologií pro multiplatformní vývoj, a především zájmem o ReactJS, vznikla myšlenka, zda by bylo možné urychlit vývoj mobilních aplikací firmy tím, že by se vývoj přesunul do interního týmu bez potřeby zaměstnání velkého počtu vývojářů zaměřených na nativní platformy. React Native byl jednou z možností, jak otestovat, zda by tato cesta byla reálnou hlavně z pohledu požadavků aplikací.

Jako vhodný kandidát byla vybrána aplikace eObchodník, která i mimo standardních požadavků mobilní aplikace má i řadu nároků na nativní funkcionality, které jsou klíčové pro její použitelnost. Tím vznikl nezávazný projekt pro vytvoření aplikace pomocí React Native. Primárně se jednalo pro platformu Android s tím, že by vývoj měl být minimálně závislý na specifikaci jednotlivých platforem.

7.2 *Výběr případů užití*

S vedením společnosti byl dohodnut rozsah vytvářené aplikace se specifikací požadavků, které by mohly ve větší míře odhalit problémy, kterým by se muselo při vývoji čelit. Finálně byly vybrány 4 hlavní funckionality aplikace, které se zdály být

jako nesložitější z hlediska logiky a integrace s okolními systémy a službami. Jednotlivé funkcionality byly rozděleny na několik konkrétních případů užití. Z obrázku č. 5 je patrné, do jaké míry vybrané případy pokrývají celkovou funkcionalitu aplikace. Ty případy, které byly v rámci nové aplikace implementovány, jsou vyznačeny zelenou barvou. V následujících podkapitolách jsou vysvětleny jednotlivé hlavní funkce.

Prohlídka vozidla

Prohlídka majetku slouží obchodníkovi k pořízení fotodokumentace potřebné k uzavření smlouvy pro pojištění vozidla v případech, ve kterých je doložení fotografií nezbytnou součástí. Obchodník musí vyplnit číslo smlouvy, které má přiděleno, a pořídit jednotlivé fotografie z předepsaných úhlů. Fotodokumentace může být uložena do stavu „rozpracovaná“ a později „doplněna“. Po dokončení jsou jednotlivé fotografie odesílány na službu, která spravuje digitální archiv a správu veškeré dokumentace k pojistkám. Odeslané prohlídky jsou potom zobrazeny v záložce *Historie*.

Prohlídka majetku

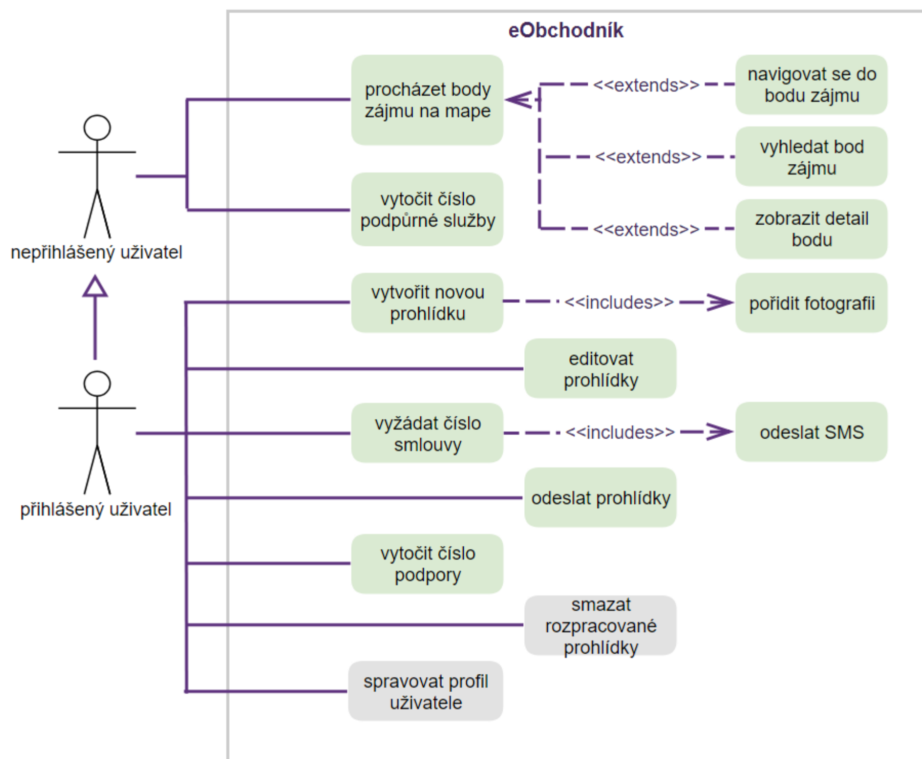
Prohlídka majetku funguje obdobně jako prohlídka vozidla akorát s rozdílem požadavků na pořizované fotografie.

Služby na mapě

Služby na mapě poskytují obchodníkovi veškeré informace ohledně poboček České pojišťovny, czechPOINTů, notářů, bankomatů a dostupných hotspotů. Obchodník má možnost zobrazit detail pobočky s rozšiřujícími informacemi a přímo kontaktovat zodpovědnou osobu. Jednotlivá místa lze vyhledávat podle adresy a lze se nechat do nich dovést prostřednictvím navigace. Vyhledávané body se řadí v závislosti na poloze uživatele.

Podpůrné akce

Další funkce, které aplikace poskytuje, je vyžádání čísla smlouvy, kontaktování podpůrné linky a helpdesku. Do této kategorie patři i možnost nastavení typu připojení do sítě, které bude aplikace používat.



Obrázek 5 Use Case Diagram

7.3 *Analýza*

V době počátku projektu byla primární základna hardware společnosti především na platformě Windows, proto byla aplikace nejprve vyvíjena primárně pro Android. V průběhu vývoje byly potom jednotlivé funkcionality testovány i pro iOS na simulátoru poskytovaného vývojovým prostředím Xcode. Od externího dodavatele byl poskytnut projekt pro nativní Android s tím, že postrádal veškerou dokumentaci, kormě ukázky komunikace se službou spravující digitální archiv. Na základě vybraných případů užití bylo nemalé množství času věnováno právě analýze funkcionalit původní aplikace a zaznamenávání všech překážek, které by při vývoji mohly nastat.

Jednotlivé případy užití kombinují aplikační logiku, zpracování požadavků obchodníka i práci s nativními komponentami zařízení. Jednotlivým požadavkům předcházela náročná část, jak novou aplikaci uvést do stavu, kdy je možné začít implementovat konkrétní požadavky. Primárním úkolem také je do maximální míry

udržet vzhled aplikace a rozložení korespondující s původním řešením. Je to hlavně z toho důvodu, že původní rozložení bylo konzultováno a schváleno UX specialisty.

Z obrázku č. 5 je zřejmé, že k veškerým funkcionalitám kromě služeb na mapě je přístup povolen pouze přihlášenému uživateli. Ověření uživatele probíhá pomocí odeslání testovacího dotazu na službu digitálního archívu czgDa. Služba komunikuje pomocí protokolu Simple Object Access Protocol (SOAP), který podporuje přenos dat pomocí XML elementů. [37] Je tedy potřeba řešit autorizaci pomocí SOAP rozhraní a ukládání údajů o uživateli do databáze. Uživatele bude možné ověřit i ve stavu off-line, bude proto nutné zařadit bezpečností mechanizmy, které tento proces zajistí. Ověřovací mechanizmus si bude držet přihlášeného uživatele po dobu 30 minut. Po uplynutí této doby bude uživatel odhlášen a bude vyžadováno nové ověření. Uživatel má dále možnost nastavit si preferovaný způsob připojení aplikace do sítě, které zabraňuje tomu, aby se např. objednávky odeslaly přes mobilní data. Bude možné přepínat mezi více uživateli v rámci jedné aplikace.

Prohlídka majetku a vozidla

V rámci tohoto požadavku bude potřeba řešit problémy spojené s používáním nativního fotoaparátu zařízení, zápisem na lokální úložiště zařízení a následným zpracováním obrazových dat. Každá fotografie bude komprimována a doplněna o vodotisk s aktuálním časem a polohou uživatele. Fotografie bude možné upravovat a v případě potřeby znovu vyfotografovat. Bude tedy potřeba přistupovat k aktuální poloze uživatele. Bude nutné vytvořit uživatelské rozhraní, které bude odezvou odpovídat nativní aplikaci.

Jednotlivé prohlídky je možné uložit do rozpracovaného stavu a umožnit tak dodatečnou editaci. Obchodník má dále možnost procházet již odeslané prohlídky. Bude potřeba vhodně implementovat lokální databázi, která bude uchovávat požadovaná data pro uživatele.

Služba pro archivaci digitálního materiálu komunikuje pomocí protokolu SOAP. Bude tedy nezbytné vhodně implementovat dané komunikační rozhraní. K veškeré komunikaci byl přiložen testovací projekt pro software SoapUI, kde byly definovány jednotlivé ukázky požadavků na službu.

Služby na mapě

Služby na mapě nevyžadují přihlášení uživatele a poskytují informace o bodech zájmu na mapě. Největším úskalím v tomto případě bude práce s mapou a vypořádání se s velkým množstvím informací, které bude potřeba zobrazit. Jednotlivé typy bodů má uživatel možnost zobrazit aktivováním příslušné vrstvy na mapě. Pro některé typy se množství bodů pohybuje v řádech tisíců, bude proto nutné nalézt řešení, které povede k přehlednému a hlavně rychlému zobrazení.

Další funkcí je vyhledávání na mapě podle adresy či názvu bodu na mapě. Výsledky vyhledávání se přehledně řadí podle konkrétní vzdálenosti do seznamu a po kliknutí je uživatel na mapě přenesen na daný bod.

Data o pobočkách jsou veřejně poskytována Českou pojišťovnou pomocí webové služby, která využívá architektonický princip *Representational State Transfer* (REST) a komunikuje pomocí formátu json. [38]. Odpověď má velikost přibližně 6 MB. Zdrojem ostatních dat je v původní aplikaci soubor s koncovkou *.db, který je s každým vydáním aplikace pravděpodobně exportován přímo z databáze.

Při inicializaci mapy je automaticky vyhledána poloha uživatele, do které je pohled mapy přesunut. Celkově by práce s mapou měla být rychlá hlavně proto, že obchodník často požaduje takové informace během pohybu.

Podpůrné akce

Dalšími rozšiřujícími funkcemi aplikace jsou vyžádání čísla smlouvy a kontaktování podpory. Bude potřeba vyřešit komunikaci a využívání ostatních aplikací v zařízení, aby došlo k automatickému vytočení čísla a odeslání předem sestavené SMS. Uživateli bude dále zobrazena přehledná nápověda k aplikaci.

8 Vlastní řešení

Následující kapitola popisuje nové vlastní řešení aplikace od analýzy, přes implementaci až po generování produkční verze a optimalizaci.

8.1 *Nastavení prostředí a nástroje*

K samotnému vývoji bylo využito několik pracovních prostředí. K části vývoje v JavaScript bylo použito Visual Studio Code (VSCode) a následně IntelliJ IDEA. K přechodu k druhé možnosti došlo hlavně z důvodu jednoduchého spouštění testů aplikace a jejich ladění pomocí integrované podpory a snazší práce s git. VSCode měl na druhou stranu výhodu ve své rychlosti a jednoduchosti. K práci s nativní částí kódu Android bylo využito Android Studio, a to hlavně pro přehledný log připojeného zařízení.

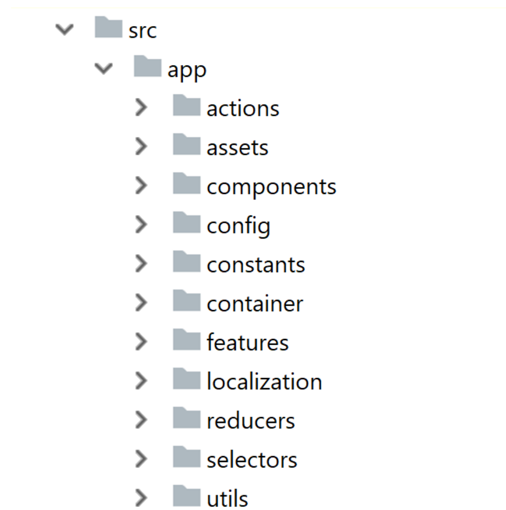
Pro usnadnění vývoje bylo využito několik nástrojů, které měly na starosti automatické opravování chyb a správné formátovaní výsledného kódu. Prvním z nich byl nástroj ESlint, který zvýrazňuje chyby vytvořené v jazyce JavaScript verze EcmaScript a nabízí automatickou opravu. To vede vývojáře k tomu psát přehledný a srozumitelný kód. V prostředí VSCode pracuje ESlint velice rychle a jednotlivá upozornění se zobrazují téměř instantně během psaní. V IntelliJ IDEA je při používání kontrola o něco zpožděna, což může v některých chvílích vývojáře zmást. ESLint je primárně vytvořen k použití z příkazové řádky což dává jednoduchou možnost jeho uplatnění např. v rámci automatizovaných testů . ESlint lze snadno nakonfigurovat pomocí pravidel, které se zadávají do vygenerovaného souboru *.eslintrc.json*. Pro tento projekt byla jako primární zdroj pravidel využita konfigurace od Airbnb, která je dostupná při inicializaci. ESlint poskytuje také automatickou opravu drobných chyb, kterou lze spustit pomocí příkazu *lint –fix*.

Dalším nástrojem byl JS, CSS a HTML formáter Prettify, který se vypořádá i s JXS syntaxí a zajišťuje tak jednotný formát pro soubory aplikace.

8.2 *Rozvržení aplikace*

Společná část aplikace má standardní strukturu aplikace psané v ReactJS s tím rozdílem, že obsahuje některé RN specifické konfigurace, viz obrázek č. 6. Jednotlivé

funkcionality aplikace byly navrženy tak, aby byla aplikační logika oddělena od zobrazovací.



Obrázek 6 Struktura aplikace

Veškerá aplikační logika aplikace je obsažena v adresáři `actions`. Zde je adresářová struktura podle typu objektu, kterého se dané funkcionality týkají *(inspection.js, photo.js,...).* Z akcí je potom přistupováno do dalších modulů aplikace, jako jsou `features` a `utils`. Jednotlivé akce jsou volány na základě události aplikace nebo z jiné akce.

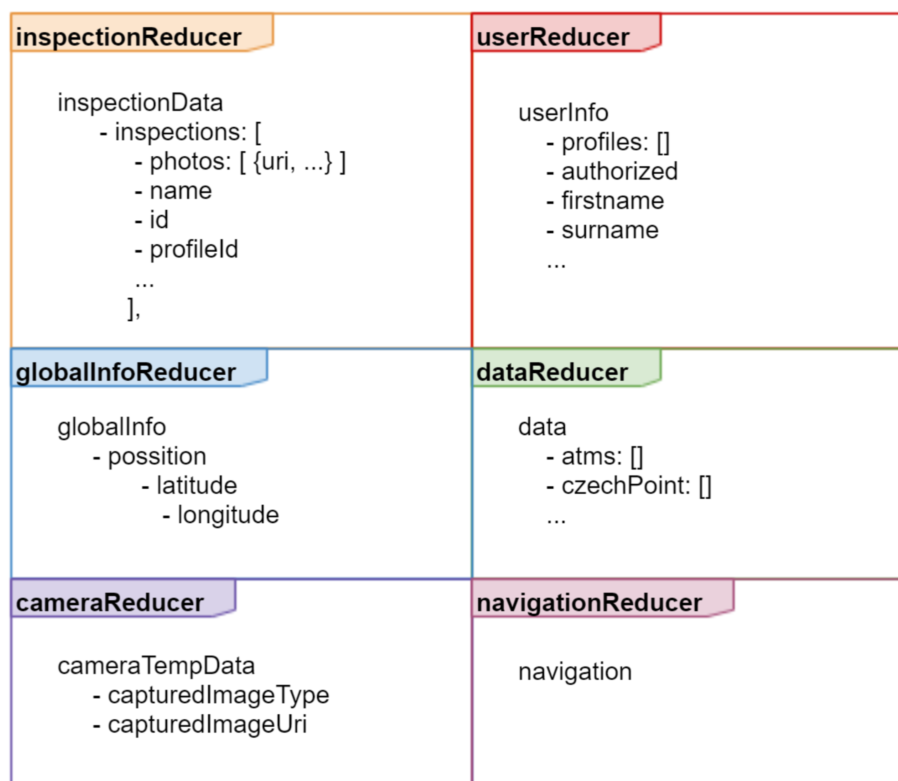
Veškeré obrazové zdroje aplikace jsou obsaženy v adresáři `assets`. Zde se pak nacházejí i data bodů zájmů, která jsou součástí aplikačního balíku ve formátu *json*. Tato data se automaticky inicializují při sestavení *store* aplikace při startu.

Veškeré UI prvky jsou obsaženy v adresářích `components` a `containers` podle toho, zda daná komponenta přímo přistupuje ke stavu aplikace. V `containers` jsou potom definovány veškeré stránky aplikace pod adresářem `pages`. Jednotlivé komponenty jsou rozděleny do adresářů podle svého účelu. Složitější komponenty mají zpravidla samostatný adresář a menší komponenty, např. různá tlačítka, jsou pod adresářem `Buttons`.

Konfigurace navigace, Redux, Reactotron a další jsou obsaženy v adresáři `config`.

Statické hodnoty používané v aplikaci jsou definovány pod adresářem `constants`. Důležitou částí aplikace je adresář `features`, který obsahuje veškeré funkcionality pro práci s nativními moduly. Je zde implementován přístup do lokálního úložiště, funkce pro čtení a zápis do lokální databáze Realm, komunikační rozhraní pro REST a SOAP a další doplňkové funkcionality jako vkládání vodoznaku do fotografie.

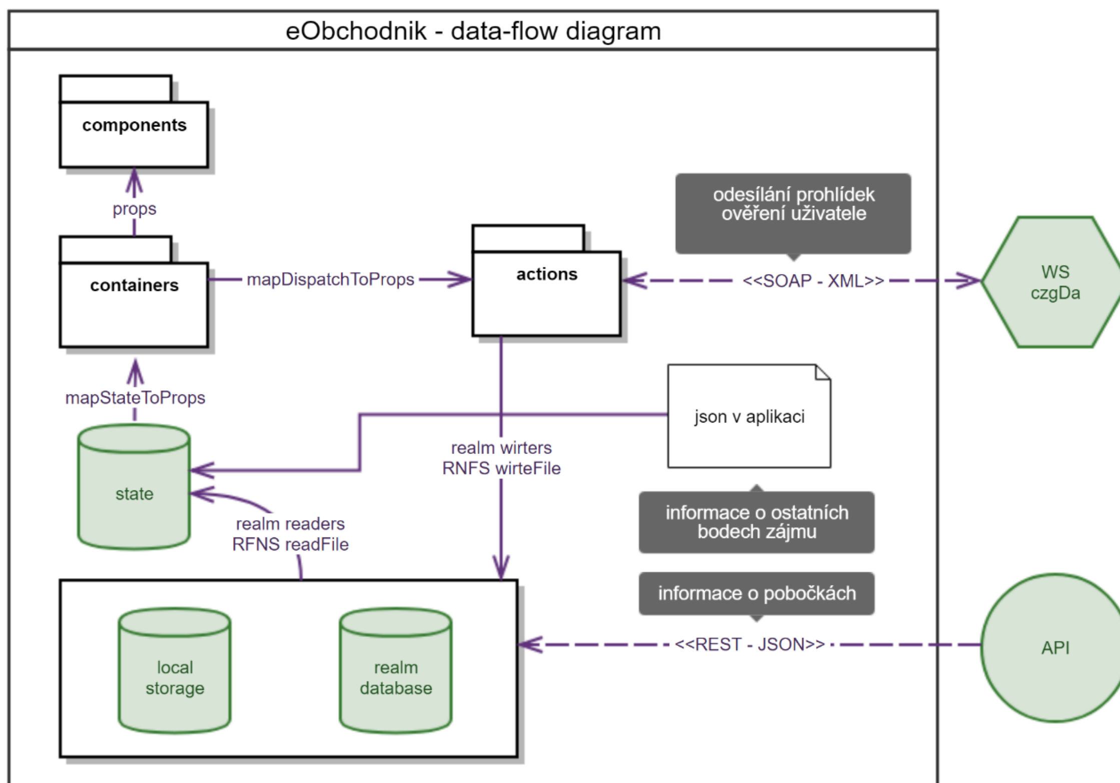
Funkcionality jako formátery, zabezpečovací mechanismy a další podpůrné funkce jsou implementovány pod adresářem `utils`. Předpisy jednotlivých akcí jsou definovány v adresáři `reducers`. Každý Reducer má na starosti určitou část store tak, jako tomu je na obrázku č. 7.



Obrázek 7 Struktura store

Pro konzistentní chování aplikace bylo nutné analyzovat veškeré zdroje dat a jejich tok aplikací. Následně byla navržena taková struktura, která bude intuitivní a snadno rozšiřitelná. Vytvořená struktura toku dat aplikací včetně externích zdrojů je znázorněna v obrázku č. 8. Veškerá data, se kterými aplikace pracuje, jsou z primárního zdroje – *state*. Databáze Realm a vnitřní úložiště potom spravují ta data,

která musí mít aplikace uložená i po vypnutí. Při inicializaci aplikace se potom veškeré potřebné informace na pozadí nastavují přímo do state tak, aby s nimi následně aplikace mohla pracovat. Data jako informace o uživateli a prohlídky jsou z databáze načtena až za předpokladu, že je uživatel ověřen. Podrobný popis jednotlivých částí je zpracován v následující kapitole *Implementace*.



Obrázek 8 Data flow diagram

8.3 *Implementace*

Následující kapitola podrobně popisuje implementaci jednotlivých částí aplikace, které bylo potřeba v aplikaci vyřešit. Samotné implementaci případů použití předcházela implementace velkého množství dílčích funkcí, které pro jejich správnou funkcionalitu byly nezbytné.

8.3.1 Inicializace projektu

Projekt byl založen jako standardní NodeJS projekt. RN poskytuje nástroj k automatickému vygenerování základny aplikace. Balíček *create-react-native-app*,

je dostupný prostřednickým npm. Před samotným vygenerováním projektu je zapotřebí mít nainstalovaný NodeJS. Balíček lze poté nainstalovat globálně pomocí parametru *–g* a pak ve zvoleném adresáři spustit pomocí příkazu *create-native-project <NázevProjektu>*. Následně je vygenerován a nainstalován základní projekt pomocí *npm install*, který je připraven k prvnímu spuštění na zařízení. Pro instalaci byl kvůli své rychlosti používán nástroj *yarn* místo *npm*.

V projektu jsou automaticky vygenerovány nativní části pro android a iOS, které jsou propojeny se společnou částí aplikace. V ukázce kódu č. 7 lze vidět konkrétní propojení *gradle* projektu s RN a definování *entry point* aplikace.

```
project.ext.react = [
        entryFile: "index.js"
]

apply from: "../../node_modules/react-native/react.gradle"
```

Ukázka kódu 7 gradle.build

8.3.2 Spuštění aplikace

Existuje více možností, jak vytvořenou aplikaci sestavovat a spouštět. V projektu bylo pracováno postupně se dvěma. První z možností je využít nástroj Expo, který přináší celou řadu nástrojů usnadňujících vývoj aplikace pro RN. Expo je sestavovací nástroj vytvořen přímo pro RN, který se stará o vytváření balíčku aplikace, publikaci a jednoduchý náhled v zařízení. Expo poskytuje aplikaci Expo Client, která je dostupná jak na Google Play, tak na App Store. Po spuštění aplikace pomocí příkazu *react-native-scripts start* je standardně spuštěn balíčkovací nástroj, ale navíc od vystavení balíčku na lokálním serveru poskytuje jednoduché řešení, jak si balíček do zařízení stáhnout. Po sestavení aplikace je do příkazové řádky vytvořen QR kód, jehož naskenováním z Expo Client se balíček automaticky stáhne a spustí za předpokladu, že je spuštěn lokální server a zařízení připojeno do stejné sítě. V prostředí je pak tato adresa serveru uložena a je možné ji znovu spustit bez nutnosti dalšího skenování později. Změny se reflektují ihned.

Expo poskytuje i několik rozšíření. Jedním z nich je možnost sestavení pro *release*, tím vystaví aplikaci v rámci veřejné sítě internet. Pomocí QR kódu pak mohou např.

testeři ověřovat funkce aplikace. Proces testování a nasazování nové verze s opravou chyb se tam může několikanásobně zrychlit. Dalším benefitem je možnost využití serveru pro *push notifikace* implementované aplikace (notifikace ze strany serveru do zařízení). Vývojář se tak nemusí starat téměř o nic. Žádné přihlašovací údaje, generování klíčů a podobně. Není potřeba pracovat s Android Studio ani Xcode.

I přes veškeré své benefity nebyla nakonec tato možnost využita. Je tomu tak u většiny řešení, která rapidně usnadňují části procesu vývoje. Problém je s konkrétními a specifickými požadavky na aplikaci, jako je např. práce s nativními moduly aplikace. V těch Expo nedává vývojáři velkou svobodu. V implementované aplikaci bylo nutné spolupracovat hned s několika nativními moduly, a proto by řešení přineslo více problémů než užitku. I přes to je však Expo velmi povedeným nástrojem, který může ušetřit značné množství práce strávené řešením problémů ohledně sestavování a publikace aplikace. Pro jednodušší aplikace je jeho použití více než vhodné. S Expo bylo tedy nadále pracováno pouze jako s nástrojem, který poskytuje prostředí pro testování a spouštění menších částí aplikace k rámci ladění *(snippets).*

Nakonec byl využit standardní režim spouštění aplikace, a to pomocí lokálního balíčkovacího nástroje Metro Bundle v kombinaci s Gradle, který je sestavovací nástroj využívaný pro nativní aplikace platformy Android. Ke standartnímu vývoji je tedy vhodné mít nainstalované Android Studio, které díky integraci s Gradle nabízí přehledný nástroj, pomocí kterého lze jednotlivé Gradle procesy spouštět. V začátcích vývoje před sestavováním release verze bylo však Android Studio využíváno především jako prostředí ke kontrole výpisu logu zařízení a pro kontrolu správného sestavení po přidání nového nativního modulu. Bylo tomu tak hlavně proto, že některé chyby aplikace hlavně při práci s nativními moduly nelze odhalit jinak než sledováním již zmíněného výpisu konkrétního zařízení. Takové chyby nejsou reflektovány do konzole, kterou poskytuje Google Chrome.

Standardní spuštění aplikace na zařízení lze provést pomocí příkazu *react-native run-android (run-ios)*, který automaticky spustí lokální server a sestavení aplikace pomocí Gradle (nástrojů Xcode). Za předpokladu, že máme spuštěný emulátor nebo

připojené reálné zařízení, bude aplikace na konci procesu do zařízení nainstalována. Tato aplikace je spustitelná pouze v případě, že je na příslušné adrese k dispozici balíček. Vytvořený nativní kontejner je potřeba instalovat tehdy, pokud se jedná o první instalaci aplikace nebo změny implementace nativních modulů, kdy je nutné kompletní sestavení aplikace. V ostatních případech stačí pouze vystavit aktuální balíček a z aplikace se na něj nasměrovat. Nasměrování na konkrétní adresu lze provést ve vývojářském menu aplikace. Při bezdrátovém připojení je nutné nastavit lokální adresu počítače a port, na kterém je spuštěný lokální server.

Při spouštění na reálném zařízení může nastat několik problémů, které se pojí především s typem použitého zařízení. V případě potíží s rozeznáním připojeného zařízení je potřeba problém řešit standardně, jako tomu je u nativního vývoje. Je nutností mít na zařízení odblokovaný a zapnutý vývojářský režim a mít správně nainstalován příslušný Android SDK. Tato příprava je jednoduší s pomocí Android Studia, kde lze nástroje jednodušeji stáhnout. V případě potřeby je možné použít vestavěný emulátor, který během testu základních funkcionalit pracoval obstojně.

Pro iOS bylo nezbytné použití Xcode, který automaticky rozezná z konfigurace *<název projektu>.xcodeproj*, že se jedná o RN projekt a lze skrz něj sestavovat aplikaci. Na výběr má několik simulátorů, na kterých je možné aplikaci testovat. Je nezbytné mít jednotně pojmenovaný projekt v package.json a první parametr metody `AppRegistry.registerComponent('<název>', () => App);`. Dále by se na cestě adresáře, kde je projekt umístěn, neměly vyskytovat speciální znaky. V tomto konkrétním případě způsoboval problém i znak „`_`“.

8.3.1 Obecné standardy aplikace

Při vytváření jednotlivých komponent byl kladen důraz na to, aby byly používány převážně bez-stavové komponenty. S vnitřním stavem komponent bylo pracováno pouze v těch případech, kdy by uchování potřebných hodnot v globálním stavu aplikace nemělo význam pro žádné jiné komponenty (index aktivní záložky v TabLayout). Pro přehlednost implementace je každá komponenta definovaná v identicky pojmenovaném adresáři, kde jsou vyčleněny styly a v případě potřeby

další doplňující funkce, které by svou přítomností v komponentě ubíraly na přehlednosti. Každá komponenta má striktně definované typy props pomocí PropTypes, které do komponenty mohou vstupovat. Každý atribut je okomentován dokumentačním komentářem. Veškeré textace komponent jsou vyčleněny a importovány pomocí lokalizačního modulu.

Komponenty jsou rozděleny do dvou hlavních adresářů podle toho, zda využívají přímo Redux, standardně potom do adresářů pojmenovaných `containers` a `components`. Součástí adresáře `containers/pages` jsou definované veškeré komponenty jednotlivých stránek aplikace. Veškerá data, která tyto komponenty používají a předávají dál do vnořených komponent, jsou čtena ze stavu aplikace pomocí standardního vývojového vzoru *react-redux* pomocí metody *mapStateToProps*. Kvůli optimalizačním důvodům jsou předávána pouze minimální potřebná data, která jsou získávána pomocí selektorů. Rozhodovací logika je prováděna výhradně uvnitř selektoru, nikoliv uvnitř metody *mapStateToProps*. Příklady správného použití selektorů budou v praktické části aplikace.

8.3.2 Vzhled aplikace

Samotná implementace aplikace započala analýzou jednotlivých obrazovek a vyčleněním komponent. Bylo dbáno na maximální parametrizaci a na zachování původního vzhledu. Veškeré bitmapové zdroje, textace a barvy byly převzaty z původní aplikace.

Nedílnou součástí vytváření vzhledu je způsob, jak budou komponenty stylovány. Každá stylovaná komponenta má pro větší přehlednost veškeré styly vyčleněny do samostatného souboru ve svém adresáři – *styles.js*. K samotné definici bylo využito rozšíření *React Native Extended StyleSheet,* které poskytuje možnosti jako globální proměnné, matematické operace a mnoho dalšího přímo v definici stylu. Definice stylu komponenty je vytvořena pomocí metody `create` a má strukturu objektu, kde každý klíč určuje identifikaci daného stylu, který bude následně použit na konkrétním elementu, viz obrázek č. 8.

```
const PADDING = 10;
const styles = EStyleSheet.create({
  text: {
    color: '$white',
    paddingBottom: PADDING,
    paddingTop: PADDING,
    borderBottomColor: '$cpLightBlueButton',
    borderBottomWidth: 0.5,
  },
 });
```

Ukázka kódu 8 Ukázka definice stylu

Výsledným produktem předchozí funkce, který je exportován ze souboru, je objekt s atributem `text` s již doplněnými finálními hodnotami. Stačí tedy daný soubor importovat a na požadované komponentě definovat atribut *style* např. na hodnotu *styles.button*.

Veškeré barvy použité v aplikaci jsou jednotně definovány v souboru *colorVariables.js* v adresáři *constants*. K tomu, aby bylo možné používat jednotlivé barvy v definicích stylů po celé aplikaci, je nezbytné při inicializaci aplikace (nejlépe v kořenovým index.js) provést příkaz `EStyleSheet.build(colors);`, kde vstupní hodnota je výchozí produkt souboru *colorVariables.js* viz ukázku kódu č. 9.

```
const colors = {
  $white: '#ffffff',
  $whiteBlue: '#d5eaf6',
  ...
  $red: '#ff0000',
};
export default colors;
```

Ukázka kódu 9 Definice barev

Názvy jednotlivých proměnných je nezbytné pojmenovávat se znakem „$“ na začátku. Definicí barev tímto způsobem lze získat dobrý přehled o použitých barvách a jednoduchý přístup bez nutnosti dohledávání přesné hodnoty.
Veškeré obrazové zdroje aplikace byly definovány v adresáři *assets/images*. Jednotlivé obrázky jsou importovány do společného souboru pomocí *require('./xxx.png');* a nadále exportovány jako jeden objekt. Pro specifikaci zdroje je pak zpravidla do jednotlivých komponent zasílán řetězec s názvem daného klíče v objektu. Veškeré

komponenty využívající bitmapu používají nativní komponentu `Image`, která jako parametr source obdrží konkrétní obrázek, jako tomu je v následující ukázce.

```
<Image
  source={prop(imageName, images)}
  resizeMode="cover"
/>
```

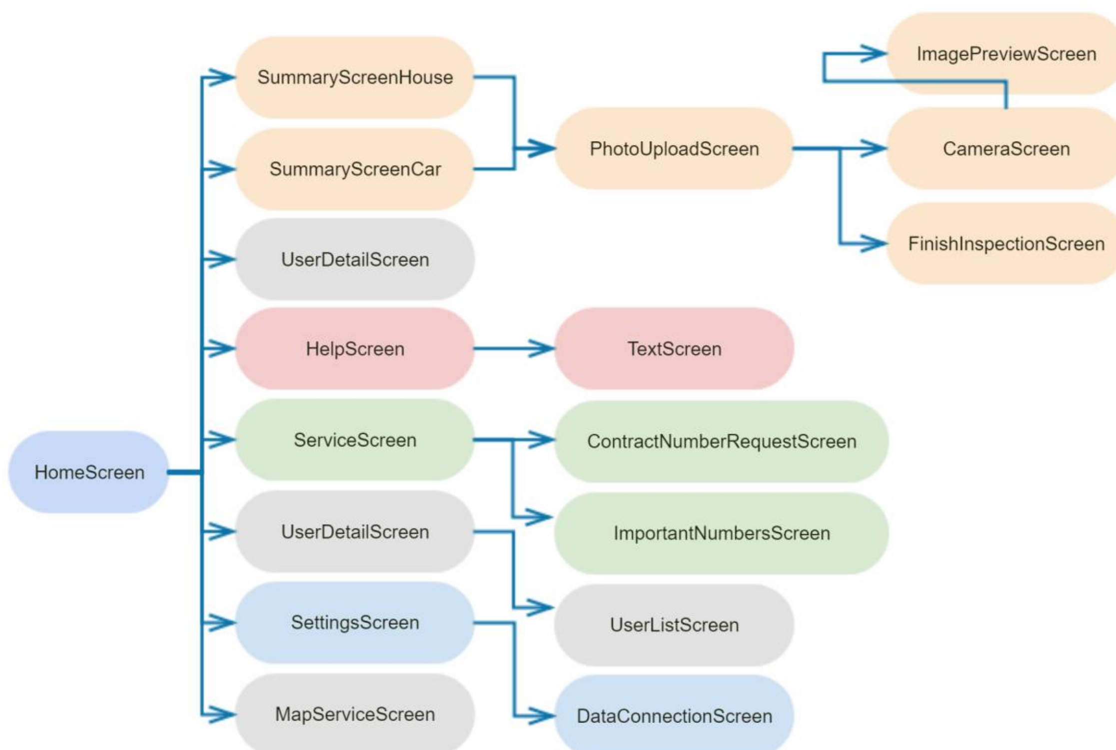
Ukázka kódu 10 Zdroj bitmapy

8.3.3 Navigace

K navigaci mezi jednotlivými obrazovkami aplikace byla využita knihovna *React-navigation,* která poskytuje řešení právě i pro aplikace RN. Knihovna poskytuje tři základní druhy navigací – *Stack navigator, Tab navigator* a *Drawer navigator*, které lze vzájemně kombinovat. Požadavkem na navigaci aplikace bylo zachování původního rozložení včetně výsuvného bloku s rychlou navigací. Výsledná navigace byla sestavena tak, aby vyhovovala následujícím požadavkům:

1. Možnost přístupu do postranní navigace odkudkoliv.
2. Podpora přechodových animací.
3. Zachování historie a funkce zpět.
4. Kontrola ověření uživatele při přechodu na určité obrazovky.

Primární navigací byla implementována *DrawerNavigation,* ve které byly definovány obrazovky první úrovně aplikace. Pro jednoduchou aplikaci s bočním panelem lze využít pouze tuto navigaci bez větších problémů a výsledek bude uspokojivý. Problém ale nastává v případě, kdy má nějaká část aplikace několik obrazovek jdoucích po sobě a je potřeba poskytnout uživateli možnost jít zpět pouze o jednu úroveň. DrawerNavigation počítá pouze s první úrovní, a proto neuchovává žádnou historii, potom každý krok zpět znamená, že se aplikace vrátí na výchozí obrazovku dané navigace definovanou v jako `initialRouteName`. Přechody v obrazovkách první úrovně také nepodporují úpravu stylu přechodu na další obrazovku.
Pro sekce obou typů prohlídek, služeb, nápovědy a nastavení bylo potřeba udělat vnořený navigátor. Finální struktura navigace a jednotlivých obrazovek vypadá potom jako na obrázku č. 9.



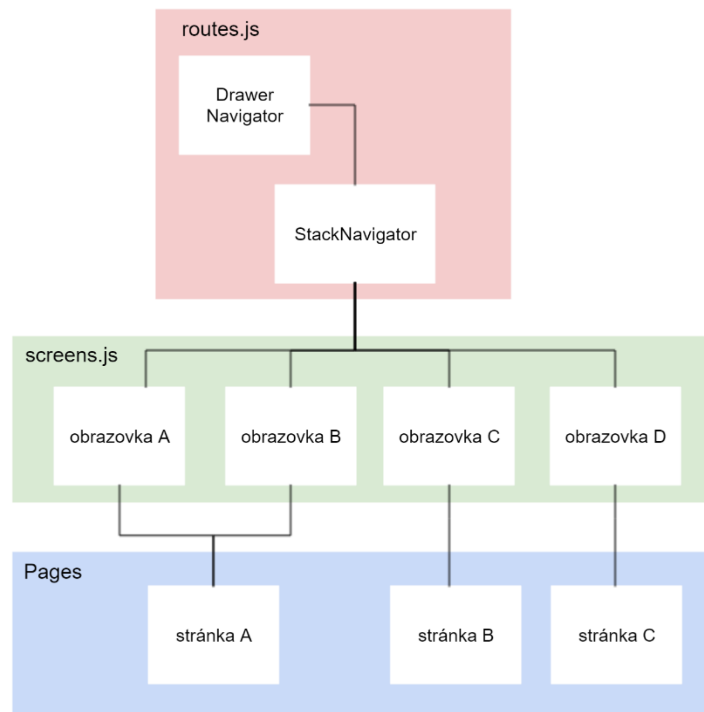
Obrázek 9 Struktura obrazovek aplikace

V prvním sloupci předchozího obrázku jsou obrazovky, které byly součástí hlavního *DrawerNavigator*. Pro každou obrazovku, ze které lze pokračovat dále, byl vytvořen další navigátor *StackNavigator*. Každý záznam obrazovky v *DrawerNavigator* je definován tak jako v následující ukázce kódu č. 11. Do parametru `screen` lze přiřadit komponentu obrazovky nebo další navigátor. Pokud je navigováno do vnořeného navigátoru, je zobrazena výchozí obrazovka daného navigátoru.

```
SettingsStack: {
  screen: SettingsStack,
  navigationOptions: {
    drawerLabel: () => 'Nastavení',
  },
},
```

Ukázka kódu 11 Definice stránky navigace v routes.js

Takto je celá struktura definována v souboru `routes.js`. Konkrétní obrazovky jsou importovány ze souboru `screens.js`, kde každá definuje to, jaká komponenta se má vykreslit. Struktura navigačních komponent je znázorněna na obrázku č. 10.



Obrázek 10 Struktura navigačních komponent

Více obrazovek může odkazovat na stejnou stránku. Například pro prohlídky vozidla jsou dvě rozdílné obrazovky využívající stejnou komponentu, viz ukázku kódu č. 12.

```
export const SummaryScreenCar = props =>
  <SummaryPage {...props} recordType={inspectionTypes.CAR} />;
export const SummaryScreenHouse = props =>
  <SummaryPage {...props} recordType={inspectionTypes.HOUSE} />;
```

Ukázka kódu 12 Definice obrazovek a parametrů

Při navigaci mezi obrazovkami v rámci jednoho navigátoru lze předávat parametry. U přechodu do jiného navigátoru však tato možnost není, a proto je nutné předat parametry staticky rozlišením dvou různých typů obrazovek.

Pro odlehčení zápisu jednotlivých obrazovek jsou využívány dvě *high-order-component* (HOC) *BasicPage.js* a *PageWithTopNavigation.js*. HOC funguje podobně jako již zmíněná *high-order-function* s tím rozdílem, že komponentu zpravidla obaluje další komponentou. Není potom nutné definovat pokaždé stejnou komponentu explicitně uvnitř stránky. Většina obrazovek má definovaný jednotný vzhled pozadí (*BasicPage.js*). Obrazovky, které využívají horní navigační panel pouze k zobrazení

ikony pro vysunutí postranního panelu a popisku, používají `PageWithTopNavigation`, kde parametr `label` udává zobrazovaný popisek a `component` předává konkrétní komponentu stránky, viz ukázku kódu č. 13. Ostatní obrazovky mají kvůli specifickým požadavkům komponentu `TopNavigation` definovanou přímo uvnitř stránky.

```
const PageWithTopNavigation = (props) => {
  const { navigation, label, component } = props;
  return (
    <View style={{ height: '100%' }}>
      <TopNavigation navigation={navigation} label={label} />
      {React.createElement(component, props)}
    </View>
  );
};
```

Ukázka kódu 13 HOC

Jednotlivé stránky potom odebírají data ze stavu aplikace a definují strukturu konkrétní obrazovky. Samotná konfigurace navigace do aplikace je zajištěna pomocí několika kroků:

1. Přidání reduceru pro navigaci, který nastavuje aktuální stav navigace na základě odchycené akce a metody *Navigator.router.getStateForAction.*
2. Nastavením navigátoru jako vnořenou komponentu kořenové komponenty aplikace, který bude spravovat jednotlivé cesty aplikace (routes).
3. Mapováním parametru `navigation` do každé komponenty obrazovky.

Předaný parametr `navigation` nese metodu `navigate`, pomocí které je možné vyvolat akci navigace na specifikovanou obrazovku nebo akci zpět. Předávání dat mezi jednotlivými obrazovkami bylo zajištěno pomocí parametrizace. Parametry jsou zpravidla předávány jako druhý argument metody `navigate` nebo samostatně pomocí metody *setParams*. Je tomu tak například u procesu vytvoření objednávky, kdy je potřeba předat id vybrané objednávky pro správné zobrazení dat na následující obrazovce. Je výhodou, že se původní parametry obrazovky uloží do jejího stavu a jsou znovu předány v případě zpětného pohybu.

Na první úrovni navigace bylo potřeba definovat i takové obrazovky, které nebudou zobrazeny v postranní navigaci, ale bude do nich přistupováno jiným způsobem, např. tlačítky na úvodní obrazovce. V `navigationOptions` lze definovat mimo jiné i název obrazovky pomocí parametru `drawerLabel`, který bude v postranním panelu zobrazen uživateli. Jako výchozí popisek je jinak zobrazen název dané cesty. V případě stránky, která má zůstat skryta, byl nejprve popisek vyplněn jako *null*. Takto byl ale pro každou položku připraven kontejner, na který byly aplikovány styly. To mělo za následek aplikování rozdělovníku i na skryté cesty. Byla proto vytvořena vlastní komponenta tvořící obsah navigačního panelu, která jednotlivé cesty filtruje, viz následující ukázku kódu.

```
const visibleItems = ['HomeScreen', 'SettingsScreen', 'HelpScreen'];
const removeHiddenItems = item => contains(item.key, visibleItems);

const getFilteredAndStyledItems = ({ items, ...other }) => (
  <View style={styles.container}>
    <View style={styles.line} />
    <DrawerItems
      activeBackgroundColor={colors.$cpBlue}
      activeTintColor={colors.$white}
      inactiveTintColor={colors.$white}
      itemStyle={styles.item}
      items={filter(removeHiddenItems, items)}
      {...other}
    />
  </View>
);
```

Ukázka kódu 14 Filtrování skrytých obrazovek

V metodě `getFilteredAndStyledItems` je zachycen parametr nesoucí jednotlivé položky navigace a podle výše specifikovaných cest pak profiltrován. Jsou zde přímo aplikovány i styly položek navigace a barvy.

Bylo potřeba upravit funkci hardwarového tlačítka pro zpětnou navigaci. Výchozí funkce měla za následek minimalizaci celé aplikace, což nebylo žádoucí. Pomocí následující ukázky kódu lze přepsat chování daného tlačítka.

```
import { BackHandler } from 'react-native';

BackHandler.addEventListener(
'hardwareBackPress', () => goBack(dispatch, nav)
);
```

Ukázka kódu 15 Přepsání výchozí funkce HW tlačítka

Navigace je často zmiňovaných problémem z pohledu výkonu hybridních aplikací, za předpokladu doporučených postupů bylo dosaženo téměř totožné rychlosti a odezvy. Na novějších zařízeních nebyl znát žádný rozdíl. U těch méně výkonných animace přechodů občas nepůsobily plynule. Tento problém ale občas nastával při testování původní aplikace, ale s menším počtem výskytů.

8.3.4 Práce s nativními moduly

K dílčím funkcím aplikace je využito několik nativních modulů třetích stran. V momentě, kdy se aplikace rozšiřuje o další moduly, mohou nastat komplikace ohledně konfliktů jednotlivých knihoven. Byly použity výhradně ty moduly, které podporují Android i iOS.

8.3.5 Lokální ukládání dat

Aplikace využívá několik způsobů, jak ukládá lokální data aplikace. Prvním způsobem je již zmíněné držení globálního stavu aplikace pomocí Redux, který řídí celkové chování a zobrazení dat v aplikaci.

Samotná struktura store byla navržena tak, aby udržovala minimální potřebná data přehledně a jednoduše v logických celcích. Celý store je potom pod správou následujících reducerů:

- `inspectionReducer` – veškerá data prohlídek,
- `userInfoReducer` – informace o přihlášeném uživateli,
- `globalInfoReducer` – obecná aktuální data jako poloha uživatele,
- `cameraReducer` – pomocná data při pořizování snímků,
- `navigationReducer` – správa stavu navigace,
- `dataReducer` – data, která jsou zobrazovaná na mapě.

Realm

V několika situacích bylo potřeba uchovávat určitá data lokálně v zařízení. Jednalo se o informace o uživatelských profilech a jednotlivých prohlídkách. Ze všech dostupných řešení byla vybrána knihovna Realm Database, která vznikla jako alternativa k SQL Lite. Realm podporuje celou řadu programovacích jazyků a poskytuje také cloudové řešení. Pomocí Realm Studio lze snadno nahlížet do uložených dat v aplikaci. Realm poskytuje řešení RealmJS pro aplikace psané pomocí jazyka JavaScript. Realm vytvořil přímo podporu pro aplikace psané v RN. Pro použití je nutné nainstalovat balíček *realm* pomocí npm a dále propojit nativní aplikace s tímto nainstalovaným modulem v *node_modules*. Lze použít nástroj link a poté důkladně kontrolovat podle dokumentace, že je veškerá konfigurace v pořádku. Po úspěšné instalaci je možné začít Realm používat, není potřeba žádná další konfigurace v rámci aplikace.

V první řadě byl definován model datové vrstvy, kdy každému ukládanému objektu je potřeba definovat model s názvem, primárním klíčem, jeho strukturou a příslušnými datovými typy. V následující ukázce kódu č. 16 je definován model prohlídky. U každého atributu lze definovat jeho povinnost pomocí *?. V případě, že povinný atribut není při vytvoření nastaven, dojde k zamítnutí takového zápisu dat. Následující schéma odkazuje na schéma jiné z důvodu potřeby ukládání fotografií k příslušné prohlídce v předem specifikované struktuře. K usnadnění zápisu a čtení dat byl navržen model tak, aby konkrétní části stavu aplikace a ukládaných objektů měly shodnou či velmi podobnou strukturu.

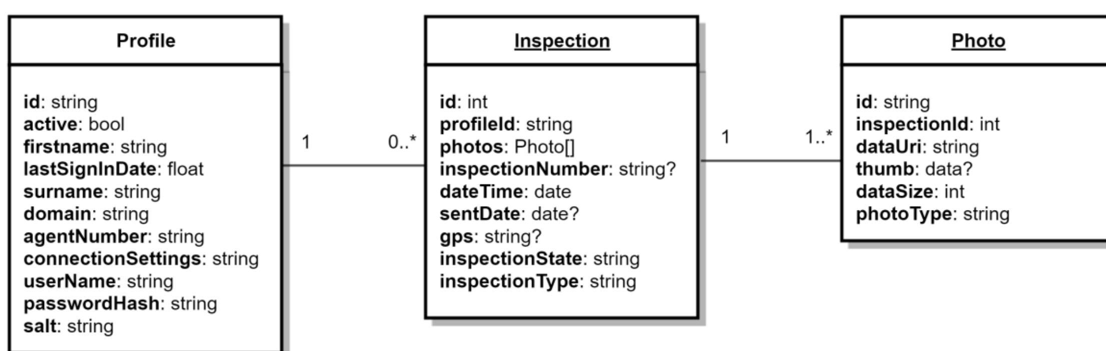
```
export default {
  name: 'Inspection',
  primaryKey: 'id',
  properties: {
    id: 'int',
    profileId: 'string',
    photos: 'Photo[]',
    inspectionNumber: 'string?',
    dateTime: 'date',
...
  },
};
```

Ukázka kódu 16 Model prohlídky

Struktura ukládaných dat je znázorněna v obrázku č. 11. Kombinace těchto modelových objektů se nazývá schéma a je vytvářeno pomocí následujícího řádku:

```
new Realm({ schema: [Profile, Photo, Inspection] }).
```

S touto instancí je pak pracováno napříč aplikací. Pro zápis záznamu do databáze je využíváno funkce `Realm.create`, případně `Realm.update`. Funkce určené k zápisu dat do databáze jsou definovány v rámci schématu do jednoho soboru pojmenovaném jako *wirter (*inspectionWriter*) v balíčku *features/realm/writers*. Metody pro vytváření a modifikaci je nutno obalit do zapisovací relace `Realm.write`. Pro náročnější zápisy do databáze bylo využito asynchronních funkcí, které svůj výsledek publikovaly po dokončení do stavu aplikace.



Obrázek 11 Schéma databáze

Lokální úložiště

K zápisu do lokálního úložiště byl využit modul *react-native-file-systém (RNFS)*, který poskytuje několik užitečných nástrojů, které jsou na platformě nezávislé. Výhodou je to, že vývojář nemusí specifikovat místo uložení pro každou platformu zvlášť. RNFS si tyto proměnné drží, na základě typu zařízení je potom nutné pouze specifikovat typ umístění, se kterým chceme pracovat (dokumenty, stažené soubory apod.).

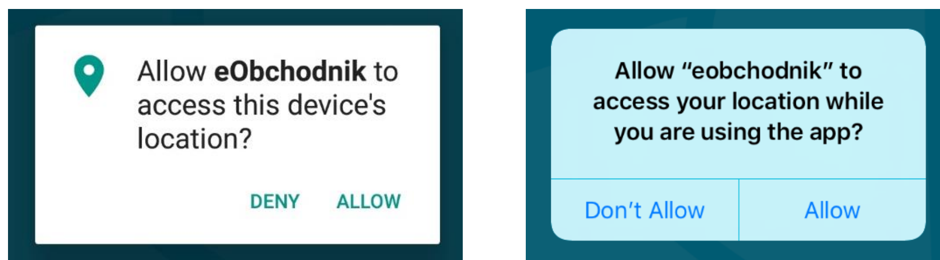
Původní myšlenka byla taková, že by byl Realm využit na veškerá data, která aplikace spravuje i po ukončení. Problém ale nastal v případě, kdy byla potřeba načíst

z databáze větší objem dat a výsledek publikovat ve stavu pomocí *dispatch*. I přes to, že samotné čtení bylo implementováno asynchronně a trvalo okolo 1 vteřiny, aplikace přestala pracovat tím způsobem, že bylo hlavní vlákno aplikace dále nepoužitelné. Výsledek operace byl vypsán jako úspěšný pomocí *redux-logger,* ale aplikace již dále nebyla funkční. Po několika neúspěšných pokusech o nápravu byla nakonec data ukládána jiným způsobem. Jednalo se konkrétně o uložení sejmutých dat fotografie, které byly komprimovány a ukládány pomocí kódování *base64*. Po bližším prozkoumání ukládání dat sejmuté fotografie bylo zjištěno, že bude postačující uložit pouze odkaz na fotografii, která je ukládána automaticky do lokálního úložiště telefonu a binární data načíst až před samotným odesláním.

Objekt *Photo* tedy v databázi drží pouze odkaz na finální fotografii v lokálním úložišti, tím se inicializace dat při spuštění aplikace rapidně zrychlila. Stejný problém nastal v případě práce s daty o pobočkách, kde je nutné stahovat data o velikosti 6 MB. V aplikaci jsou pak data ukládána do souboru formátu *txt* na lokální úložiště do dokumentů, kde jsou v případě exspirace přepisována.

8.3.6 Přístup k informacím o zařízení

K přístupu k citlivým informacím o uživateli je nutné stejně jako u nativního vývoje pro Android žádat výslovné povolení od uživatele viz obrázek č. 12. Jedná se o aplikace používající SDK 23 a vyšší. Je nezbytné tato povolení specifikovat v souboru *AndroidManifext.xml*. Pro iOS v *Info.plist*. Samotná implementace žádosti o povolení je zpravidla již součástí využívaného nativního modulu (např. *react-native-camera*). Pokud i přes to je potřeba explicitně zažádat povolení pro danou službu, je možné tak učinit přímo z JS kódu pomocí asynchronní akce a není nutné zasahovat do kódu nativního. Přehledně je tento proces propsaný v oficiální dokumentaci. V případě iOS má uživatel omezený čas, kdy musí potvrdit, jinak je automaticky daná funkce zakázána.



Obrázek 12 Povolení použití polohy uživatele (RN Android, RN iOS)

V případě aplikace eObchodník bylo nezbytné přistupovat hned k několika informacím. Pro správnou funkci služeb na mapě bylo nutné přistupovat k aktuální poloze uživatele. Uživatel má pak možnost navigace a přehledné zobrazení bodů zájmů seřazených podle vzdálenosti vzhledem k jeho poloze. Samotná poloha je načítána při zapnutí aplikace a ukládána do globálního stavu aplikace pro pozdější použití. Poloha zařízení je získávána pomocí Geolocation API, které k odhadu aktuální polohy využívá více zdrojů současně. API je globálně dostupné v aplikaci pomocí polyfilu `navigator.geolocation`. Před samotným voláním bylo potřeba definovat povolení ACCESS_FINE_LOCATION v AndroidManifest.xml. Samotné navrácení polohy z *getCurrentPosition* probíhá asynchronně. Metodu lze parametrizovat a tím ovlivnit přesnost a rychlost. Kvůli časové optimalizaci byl parametr *enableHightAccuracy* ponechán na *false*. V případě, že volání proběhne v pořádku během nastaveného timeoutu 20 sekund, nastavíse poloha do globálního stavu, viz ukázku kódu č. 17.

```
loadPosition() {
  navigator.geolocation.getCurrentPosition(
    ({ coords }) => {
      const { latitude, longitude } = coords;
      this.props.updatePositionInGlobalInfo({ latitude, longitude });
    },
    console.log('Err.'), { timeout: 20000, maximumAge: 1000 },
  ...
```

Ukázka kódu 17 Načtení pozice zařízení

V případě chyby nebo překročení limitu je volání vyhodnoceno jako chybné. Poloha je pro aplikaci nezbytná také z důvodu pořizování snímků během prohlídky. V případě, že fotografie nezahrnuje údaj o poloze v čase pořízení, nemůže být později použita jako validní podkladový materiál ke smlouvě.

Dále byla získávána informace o stavu připojení zařízení. Uživatel aplikace má v nastavení vybrat možnost, jaký typ připojení bude preferovat hlavně kvůli nechtěné spotřebě velkého množství mobilních dat. V případě, kdy se typ připojení neshoduje s nastavením, nebude možné odeslat prohlídku a ani se nebudou stahovat aktuální data o pobočkách. V takovém případě dojde k upozornění uživatele, který bude mít možnost své nastavení změnit. Informace o typu připojení je získávána z modulu *NetInfo*, který je součástí balíčku *react-native*. `NetInfo` umožňuje přidání *listeneru* na událost změny připojení. Tato změna je pak propagována do state aplikace pomocí akce `handleConnectivityChange` viz ukázku kódu č. 18.

```
NetInfo.addEventListener('connectionChange',
  netInfo => dispatch(handleConnectivityChange(netInfo)),
);
```

Ukázka kódu 18 NetInfo

Před každým voláním po síti je potom porovnáno aktuální připojení, a to co si uživatel zvolil v nastavení aplikace pomocí metody `matchSettings`, viz ukázku kódu č. 19. V případě, že uživatel nastavil možnost použití Wi-Fi a dat pouze s upozorněním, bude zobrazeno upozornění pomocí komponenty *Alert*.

```
export const matchSettings = (connectionType, connectionSettings) => {
  switch (connectionType) {
    case 'wifi': {
      return true;
    }
    case 'cellular': {
      if (connectionSettings === WIFI) {
        return false;
      } else if (connectionSettings === WIFI_DATA) {
        return true;
      } else if (connectionSettings === WIFI_DATA_WITH_NOTIF) {
        sendingWithDataAlert();
        return true;
...
```

Ukázka kódu 19 Porovnání nastavení a zdroje připojení aplikace

8.3.7 Snímání fotografií a post-processing

K zaznamenávání dokumentů bylo potřeba snímat fotografie a zajistit tak komunikaci mezi nativním fotoaparátem a aplikací. K tomuto účelu byl použit nativní modul

react-native-camera[9], který poskytuje řešení pro snímaní fotografií a zaznamenávání videa pro obě platformy.

Samotné implementaci předcházela standardní instalace nativní knihovny včetně propojení s aplikací a vyřešení konfliktů podpůrných knihoven z modulu *com.google.android.gmc*. Následně musela být definována povolení v AndroidManifest.xml k zaznamenávání snímků a ukládání do lokálního úložiště. Pro iOS je nutné definovat parametr *Privacy – Camera Usage Description*, kde musí být z bezpečnostních důvodů vysvětleno, k čemu aplikace fotoaparát používá.

Modul poskytuje komponentu *RNCamera*, která umožňuje velké množství parametrizace funkcionalit, jako jsou automatické zaostřování, blesk nebo zdroj dat. Samotné sejmutí fotografie je provedeno kliknutím na tlačítko spouště, které vyvolává asynchronní akci poskytovanou modulem `takePictureAsync`. Modul poskytuje několik způsobů, jak data o sejmuté fotografii získat. Kvůli optimalizaci ukládaných dat, byla použita možnost, kdy bude fotografie na pozadí uložena a navrácena bude pouze cesta jejího umístění, která bude potom uložena do databáze k příslušné prohlídce, viz ukázku kódu č. 20.

```
takePicture = () => {
  if (this.camera) {
    this.camera
      .takePictureAsync()
      .then((data) => {
        const { uri } = data;
        console.info(`image captured - ${uri}`);
        this.props.saveImage(uri, this.props.imageType);
      })
      .catch(err => console.info(err));
  }
};
```

Ukázka kódu 20 Sejmutí fotografie

Výsledek pořízení fotografie je potom publikován do stavu aplikace, kde je uložena adresa uri a typ pořízené fotografie (např. tachometr vozidla). Komponenta *CameraPage* zobrazuje komponentu *Camera* pouze za předpokladu, že jsou tyto dva údaje

[9] https://github.com/react-native-community/react-native-camera

ve stavu aplikace prázdné. Po jejich naplnění pomocí metody `saveImage` potom proběhne překreslení s náhledem fotografie.

Problém nastával v případě, kdy si uživatel pořízenou fotografii nevybral a fotografii snímal znovu. Původní fotografie zůstala potom uložena v lokálním úložišti. Bylo proto nutné u každého smazání anebo opětovného sejmutí původní fotografii mazat pomocí funkce *unlink.*

Kvůli optimalizačním účelům byla data následně komprimována a rozlišení fotografie zmenšeno pomocí modulu *react-native-image-resizer*. Kvalita výsledného obrazu byla zmenšena na 80 %, což mělo výrazný vliv na rychlost sejmutí fotografie. Při tomto procesu bylo potřeba opět smazat původní fotografii. Stejně tomu bylo i u přidání vodoznaku pomocí knihovny *react-native-marker.* K přidání vodoznaku byla využita metoda `markerText`, kde bylo potřeba specifikovat umístění, odsazení a barvu textu. Výsledný text byl potom složen z aktuálního času a polohy.

I přes optimalizační úkony bylo snímání fotografie problematické, a to hlavně v rychlosti post-processingu vybrané fotografie. Aplikace proto odkáže zpět na obrazovku s přehledem fotografií, kde se k příslušnému poli zobrazí indikátor aktivity a až potom následně sejmutá fotografie viz obrázek č. 13.

Obrázek 13 Indikátor aktivity ukládání fotografie

8.3.8 Networking

V aplikaci bylo nezbytné vyřešit komunikaci s okolními službami kvůli získávaní dat i k jejich odesílání. Prvním případem byla komunikace se službou czgDa, která slouží jako digitální archiv pro veškeré dokumenty. V tomto konkrétním případě pak pro jednotlivé fotografie pořízené při prohlídkách vozidel a majetku. Služba komunikuje výhradně pomocí protokolu SOAP. Bylo proto nutné vytvořit jednoduchý způsob, jak z dat vygenerovat požadovanou XML strukturu zprávy. Jednotlivé požadavky musí mít bezpečností náležitosti.

Veškeré funkcionality spojené se SOAP komunikací jsou implementovány v adresáři *features/soap*. Jsou vytvářeny dva typy požadavků, první pro ověření a druhý k samotnému odesílání dokumentů.

První metoda je volána při každém přihlášení uživatele za předpokladu, že je zařízení ve stavu online. Samotné ověření uživatele při vytváření profilu probíhá pomocí zaslání testovací obálky s autorizačním řetězcem specifikovaným v hlavičce. Podle odpovědi je potom rozhodnuto, zda je daný uživatel v evidovaný v systému či nikoliv. Za předpokladu, že se přihlašovací údaje shodují, je uživateli vytvořen profil v aplikaci, ke kterému se později budou vázat vytvořené prohlídky.

Obě metody využívají společnou implementovanou metodu *appendChildToEnvelope*, která vytváří vnořené elementy dle parametrů. Metoda může přijímat pět parametrů, z toho dva jsou volitelné. Parametry, které budou nastaveny jako atributy daného elementu, jsou definovány v poli, nad kterým následně *map* aplikuje na element *setAttribute*. Samotné vnoření parametru probíhá pomocí metody *appendChild,* viz ukázku kódu č. 21.

```
export const appendChildToEnvelope = (envelope, parentElement, tagName,
content, attrs = []) => {
  const childElement = envelope.createElement(tagName);
  map(({ value, name }) => childElement.setAttribute(name, value), attrs);
  if (isNotNil(content)) {
    const textNode = envelope.createTextNode(content);
    childElement.appendChild(textNode);}
  parentElement.appendChild(childElement);
  return childElement;
};
```

Ukázka kódu 21 Vnoření elementu

```
const envelope = new DOMParser().parseFromString(envelopeTag);
const rootElement = envelope.documentElement;
appendChildToEnvelop(envelope, rootElement, soapHeader);
const body = appendChildToEnvelop(envelope, rootElement, soapBody);
const inConfig = prop(__, configByCompany(profile.domain));
const importDocs = appendChildToEnvelop(envelope, body, 'importDoc', unde-
fined, [
  { name: 'xmlns', value: nsImport201 },
  { name: 'xmlns:s', value: nsSoapEnv },
]);
const msg = appendChildToEnvelop(envelope, importDocs, 'msg', undefined, [
  { name: 'msgVersion', value: inConfig('msgVersion') },
  { name: 'msgSourceSystem', value: inConfig('msgSourceSystem') },
...
]);
appendChildToEnvelop(envelope, msg, 'msgParams');
const msgData = appendChildToEnvelop(envelope, msg, 'msgData');
appendChildToEnvelop(envelope, msgData, 'userLogin', inConfig('userLo-
gin'));
appendChildToEnvelop(envelope, msgData, 'userGroup', inConfig('user-
Group'));
...
```

Ukázka kódu 22 czgDa požadavek

Postupné vytváření dokumentu je popsáno v ukázce kódu č. 22. K vytvoření základní obálky je použit DOMParser. U každého elementu je pak provolána funkce `appendChildToEnvelope`. Existují dva různé typy konfigurace dokumentu, záleží na doméně uživatele. Doplnění správných parametrů pro danou společnost zaručuje metoda `inConfig`. Dále jsou plněny dynamické parametry jako číslo obchodníka, číslo smlouvy a data fotografií.

```
const inConfig = prop(__, configByCompany(profile.domain));
```

Jednotlivé fotografie jsou ve chvíli odesílání asynchronně načítány z lokálního úložiště v kódování *base64* a následně přímo plněny do atributu `binaryData` elementu nesoucího konkrétní přílohu dokumentu ,viz ukázku kódu č. 23.

```
export async function readFile(inspection, photo, dispatch) {
  const binary = await rnfs.readFile(photo.dataUri.substring(7),
'base64');
  dispatch(updateBinaryForPhoto(inspection, photo, binary));
  console.log(`binary writen to state for photo ${photo.id}`);
}
```

Ukázka kódu 23 Čtení binárních dat z lokálního úložiště

Data prohlídky jsou dále uchovávána v paměti zařízení v záložce historie pro pozdější možnost náhledu. V případě, že jsou data aplikace smazána , je nutná nová registrace a uživatel již nemá přístup k minulým prohlídkám.

Samotný požadavek je odeslán pomocí *fetch API (EcmaScript6)*. Jako *body* je nastavován právě vygenerovaný dokument. *Fetch* je provolán asynchronně a následně je parsována odpověď, v tomto případě pomocí metody `text()`. V případě chyby bude uživateli zobrazeno upozornění a bude mu umožněno provést opakované odeslání. Po odeslání se vrací číslo záznamu v digitálním archívu, se kterým není dále pracováno, ale funguje jako signalizace úspěšného odeslání dokumentu, viz ukázku kódu č. 24.

```
export async function fetchAsyncText(url, body) {
  const response = await fetch(url, body);
  const data = await response.text();
  return data;
}

return fetchAsyncText(WS_BASE_URL, {
  method: 'POST',
  headers: {
    Accept: ACCEPT_ENCONDING_DEFLATE,
    'Content-Type': 'text/xml;charset=UTF-8',
    Authorization: credentials,
  },
  body,
}).then((response) => {
...
```

Ukázka kódu 24 Odeslání prohlídek

K získávání dat o pobočkách bylo opět použito *fetch API* s tím rozdílem, že odpověď byla pasována pomocí `json()`. Jednotlivé záznamy jsou před uložením do lokálního uložiště upraveny tak, aby měly stejnou strukturu jako ostatní data o bodech zájmu, je tomu tak z důvodu usnadnění následné manipulace. Z takto upravené odpovědi je vytvořen soubor *.txt, který je následně uložen spolu s datem stažení. Toto datum je pak ověřováno při každém příchodu do obrazovky mapy, a pokud je starší 24 hodin, jsou data stažena znovu. Odpověď REST API bohužel v tomto případě neposkytuje verzi ani jiný identifikátor, podle něhož by bylo možné určit aktuálnost již stažených dat. Problémem je i to, že z celkové velikosti cca 6 MB je reálně využito pouze asi

5 % dat. V případě mobilních aplikací to může být z pohledu objemu použitých dat velký nedostatek. To ale není problém použité technologie, ale spíše celkového konceptu aplikace, jehož změny nejsou předmětem této práce.

8.3.9 Lokalizace

Aplikace v současné chvíli podporuje pouze jednu jazykovou mutaci, avšak byla navržena tak, aby bylo možné její snadné rozšíření. Veškeré textace aplikace jsou přehledně definovány na jednom místě a pomocí informace o zařízení je rozhodnuto, která textace, ve které mutaci bude použita. Konkrétní jazyk je použit na základě nastaveného jazyka zařízení. K zjištění takové informace byl použit nativní knihovna *react-native-device-info*, která poskytuje celou řadu informací o zařízení, konkrétně se jedná o metodu `getDeviceLocale`, která vrací kód jazyka (pro iOS a Android vrací v jiném formátu). Rozeznávání jazyka a vyhledání textace je znázorněno v ukázce kódu č. 25.

```
import { getDeviceLocale } from 'react-native-device-info';
const defaultLocale = 'cs';
const isDefined = contains(__, keys(helpPage));

const messages = {
  helpPage,
  ...,
};

const getLanguageLocale = locale => (isDefined(locale) ? locale : default-
Locale);

export default (code) => {
  const splited = split('.', code);
  const locale = getDeviceLocale().slice(0, 2);
  return path([splited[0], getLanguageLocale(locale), splited[1]])(mes-
sages);
};
```

Ukázka kódu 25 Lokalizace

Konkrétní získání textace využívá metodu `getMessage`, která má jako parametr název souboru a textace oddělené tečkou, např.: `getMessage('homePage.services')`.

8.3.10 Integrace s Google Maps

Pro podporu zobrazení map byl využit nativní modul *react-native-maps* vytvořený společností Airbnb. Jako poskytovatel konkrétních dat bylo, stejně jako v původní

aplikaci, využito služeb Google Maps. Nativní modul byl zvolen hlavně kvůli velké komunitě, která k vývoji přispívá.

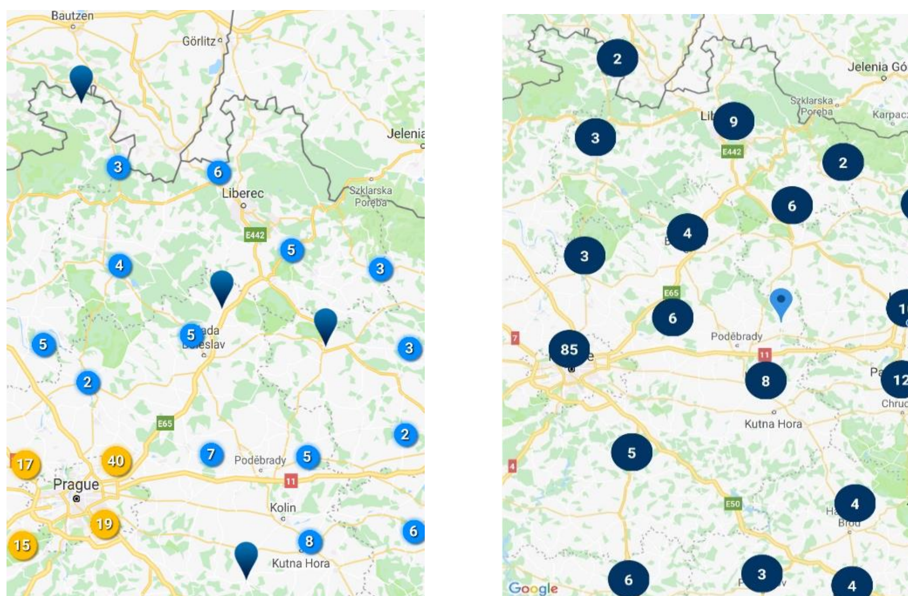
Před samotným použitím bylo nutné založit projekt v Google Cloud Platform a vybrat a povolit potřebná API. Při vytváření projektu je nutné zadat kontaktní a bankovní údaje. Každý měsíc má projekt k dispozici 200 $, které lze využívat bez nutnosti doplácení. Je také možné nastavit limity, aby nedošlo k nečekanému překročení této částky. Tento problém, ale většina aplikací nemusí řešit. Následně je potřeba vygenerovat bezpečnostní klíč pro použití pro Android i iOS. Pro Android je potřeba klíč vložit do souboru *AndroidManifest.xml*. Pro iOS potom do *AppDelegate.m*. Pro iOS je dále vyžadována odlišná konfigurace závislostí pomocí *specifikace* v Podfile.

Jedním z největších problémů bylo zobrazit velké množství dat rychle a přehledně. Bylo potřeba zajistit slučování jednotlivých bodů zájmu v určitém momentě do clusterů. U každého klusteru potom zobrazit počet bodů, které byly sloučeny. Nejprve bylo vytvořeno vlastní řešení, které slučování úspěšně řešilo, problémem byl však výkon, kdy každé překreslení trvalo příliš dlouho. Rozhodování o tom, zda dva body spadají do klusteru bylo vypočteno na základě vzdálenosti a toho, zda se ikony začínají překrývat. Po několika pokusech o optimalizaci, která měla pouze zanedbatelný efekt, bylo převzato již hotové řešení *react-native-map-clustering*[10]. Rozšíření, které poskytuje upravenou komponentu *MapView*, jež poskytuje systém pro klasifikaci dat, bylo upraveno a dále optimalizováno pro konkrétní požadavky aplikace. Jednotlivé klustery jsou spočteny v případě, že přiblížení přesáhne určitý stupeň a rozměry mapy splňují definované podmínky. Výpočet je potom prováděn asynchronně a na základě výsledku jsou do mapy překreslovány jednotlivé body. Pro konkrétní nalezení klusterů je využito knihovny Supercluster od Mapbox[11]. Spočtení předchází inicializace SuperCluster pomocí konstruktoru a nahrání jednotlivých

[10] https://github.com/venits/react-native-map-clustering
[11] https://github.com/mapbox/supercluster

bodů pomocí metody `load`. Metoda `getClusters` přijímá hranice mapy a úroveň zoom.

Vzhled byl upraven tak, aby korespondoval s původní aplikací. V případě, že se jedná o samostatný bod, má uživatel možnost být do bodu automaticky navigován za použití aplikace Google Maps. Zobrazení bodů na mapě je znázorněno na následujícím obrázku.



Obrázek 14 Clustery (nativní Android vlevo, RN vpravo)

Při použití *react-native-maps* se u některých zařízení používající OS Android objevoval problém s vykreslením ikon pro navigaci do aktuální polohy a navigace po kliknutí na bod zájmu. Problém byl způsoben tím, že se komponenta mapy po kliknutí na bod nepřekreslila a tím ikona navigace nebyla zobrazena. Jako dočasné řešení byla přidána funkce, která při kliknutí na bod na mapě změní velikost mapy tak, aby došlo k překreslení. Změna velikosti není pro uživatele viditelná.

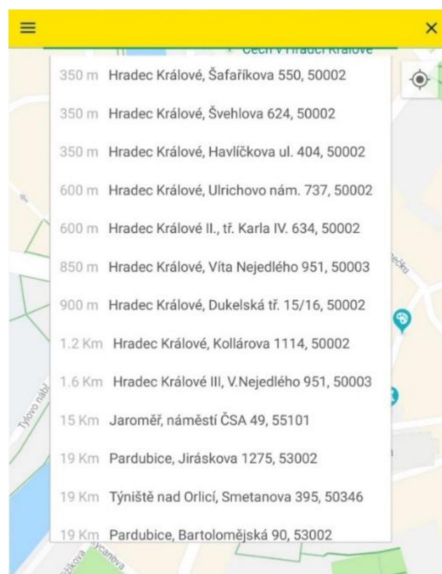
Pro snadné vyhledávání byla využita komponenta Autocomplete z balíčku *react-native-autocomplete-input*, která zvládá rychle zobrazovat velké množství dat díky postupnému načítání. Při každé změně vstupu byla data o bodech zájmu filtrována na základně výsledku z metody *filterContains* a následně seřazena, viz ukázku č. 26.

```
const filterContains = curry((query, { address, street }) => {
  const textToSearch = address || street;
  if (!textToSearch) {
    return false;
  }
  if (!query) {
    return true;
  }
  return textToSearch.toLowerCase().indexOf(query) !== -1;
});

filterOptions() {
  this.setState({
    options: filter(filterContains(this.state.query), this.props.dataBy-
Name),
  });
}
```

Ukázka kódu 26 Vyhledávání na mapě

Metoda musela porovnávat jak parametr `street` i parametr `address`, protože data v rámci stejného typu byla nekonzistentní. Výsledné položky jsou potom zobrazeny v rozbalovacím poli a po kliknutí je mapa přesunuta na místo bodu viz obrázek č. 15. Pole je aktivováno kliknutím na ikonu lupy tím, že je na komponentě *Autocomplete* vynucen *focus*. Tento krok měl výraznou prodlevu, která se při produkčním sestavení aplikace již neprojevovala.



Obrázek 15 Autocomplete s místy podle vzdálenosti

React-native-maps je rozsáhlý modul, který nabízí velké množství funkcionalit jejichž použití může být občas problematické. V průběhu vývoje aplikace byla většina problémů opravena a komponenta funguje obstojně i přes velké množství dat.

8.3.11 Odeslání SMS a vytočení telefonního čísla

Pro vytočení čísla podpory a odeslání SMS pro vyžádání čísla smlouvy byl využit modul *react-native-communications*, konkrétně potom metoda *text* a *phonecall.* U obou platforem je aplikace pouze přesměrována do příslušné aplikace a samotné odeslání nebo vztočení musí provést uživatel. Pro iOS je navíc ještě nutno nastavit do *Info.plist* položku *LSApplicationQueriesSchemes* a definovat „*tel*“ jako povolené *query*.

8.3.12 Ramda a Ramda Extension

Ramda je funkcionální knihovna jazyka JavaScript založená na konceptu, kdy je požadovaného výsledku dosaženo pomocí používání čistých funkcí[12], který jde ruku v ruce se vzorem *immutable*[13]. Největší výhodou je především možnost řetězení jednotlivých funkcí. Pro vývojáře, kteří používají primárně deklarativní k programování, může být funkcionální programování s použitím knihovny Ramda z počátku složitější. Po překonání této fáze však vývojář ocení množství kódu, kterého je ušetřen. To má za následek také přehlednost a jednoduchou editaci. Velice zdařilým rozšířením je knihovna *Ramda-extenstion*, která navazuje na koncept Ramda a nabízí celou řadu dalších zajímavých funkcí věčně podrobné dokumentace. [39]

Veškerá funkce v Ramda se drží konceptu point-free, to přináší další výhodu v řetězení vytvářených funkcí. Knihovna byla v aplikaci použita hlavně z důvodu úspory napsaného kódu a přehlednosti prakticky ve všech částech. Její přínos byl oceněn hlavně při psaní selektorů. Nevýhoda, kterou přechod funkcionálního přístupu přináší, je především v nevhodné kombinaci s deklarativním přístupem anebo použití

[12] Funkce, jejíž výsledek závisí čistě na vstupu.
[13] Návrhový vzor, který nepřipouští jakoukoliv modifikaci používaných objektů.

funkce v případě, kdy není potřeba a lze snadno vyřešit např. ternárním operátorem.

8.4 *Optimalizace*

Následující kapitola popisuje metody, které byly použity k optimalizaci vyvíjené aplikace. Oficiální dokumentace RN uvádí několik bodů, které je doporučeno dodržovat k dosažení optimální rychlosti aplikace. Je důležité si uvědomit, že veškeré funkcionality je z hlediska výkonu potřeba testovat s vypnutím vzdáleného ladění. Za tohoto předpokladu bude docíleno reálné odezvy testovaného zařízení. V opačném případě může být výkon pozitivně i negativně zkreslen využitím běhu aplikace ve vlákně prohlížeče, nikoliv samotného zařízení. Ideální je spouštět aplikaci k testu pomocí příkazu: `react-native run-android --variant=release`.
Důležité je vyvarovat se jakéhokoliv logování pomocí *console.** v rámci produkčního balíčku. Tento problém řeší plugin *babel-plugin-transform-remove-console.* Výrazné zpomalení také způsobovalo použití middleware *redux-logger*, který způsobil zpomalení odezvy aplikace u každé změny stavu přibližně o 5 s, čímž se aplikace stala netestovatelnou. Tento middleware bylo potřeba před testováním výkonu nebo generováním produkční verze zcela deaktivovat, protože předešlý plugin tento problém neřeší.

K výkonnostním propadům může docházet i za předpokladu, že do komponent vstupují *props*, které komponenta nutně nepotřebuje a tím je způsobeno její nadměrné překreslování. U menších komponent je doporučeno testovat jednotlivé případy a optimalizovat správným konkrétnějším selektorem nad daty. U rozsáhlejších komponent, kde je na vstupu velké množství props, je vhodné použít metodu *componentWillUpdate* nebo přehlednější řešení pomocí knihovny *Recompose* a HOC *onlyUpdateForKeys*. Toto řešení je vhodné především v případě, kdy je využíváno komponent třetích stran a nelze tak jednoduše optimalizovat přímo v komponentě (bez nutnosti fork).

Reselect

Jediný způsob, jak přistupovat k informacím ze stavu aplikace z komponent, je mapování stavu na *props* (*mapStateToProps*). V aplikaci bylo přistupováno ke stavu na

mnoha místech, a proto bylo potřeba nalézt vhodný nástroj, který bude čtení maximálně optimalizovat. Řešením bylo použití jednoduché knihovny *Reselect*, která nabízí způsob, jak předejít ztrátám rychlosti aplikace způsobených právě těmito dotazy na stav pomocí optimalizační metody *memoize*[14]. Taková funkce je pak nazývána selektorem. Jednotlivé selektory lze řetězit, a proto je vhodné rozdělit jednotlivé dotazy na obecné a specifické. Reselect poskytuje funkci *createSelector*, která jako parametr přijímá pole jiných selektorů, kde je každý selektor provolán se vstupním parametrem celého nového selektoru – nejčastěji *state*. Reselect pracuje pouze s částí stavu aplikace, kterou nutně potřebuje pro vyhodnocení svého výsledku. Přepočet pak probíhá pouze v případě, že ke změnám stavu právě v této části skutečně dojde. Pokud se data nemění, vrátí funkce předchozí již vypočtený výsledek. Zpravidla se selektory vytvářejí v samostatném souboru, který se váže k dané části stavu. Např. *userInfoSelectors.js* bude obsahovat všechny dotazy na strom uchovávající informace o uživateli.

```
const profileEq = propEq('profileId');
const isCar = propEq('inspectionType', 'CAR');
const isHouse = propEq('inspectionType', 'HOUSE');
const isFinished = propEq('inspectionState', inspectionStates.FINISHED);

export const getInspectionData = createSelector(identity, propOr({}, 'in-
spectionData'));

export const getInspections = createSelector(
  [getInspectionData, getActiveUserId],
  (inspectionData, userId) =>
    o(filter(profileEq(userId)), propOr([], 'inspections'))(inspec-
tionData),
);

export const getCarInspections = createSelector(
  [getActiveUserId, getInspections],
  (id, inspections) => filter(isCar, inspections),
);
```

Ukázka kódu 27 Použití knihovny Reselect

Ukázka kódu č. 27. znázorňuje praktický příklad použití knihovny *reselect*. Pro zobrazení prohlídek pro automobil je potřeba načíst prohlídky pouze pro přihlášeného

[14] Za předpokladu opakovaného volání funkce se stejnými parametry navrací výsledek z cache

uživatele a podle typu. Selektor *getCarInspection* bude přepočten pouze tehdy, pokud se změní *profileId* právě přihlášeného uživatele anebo jeho prohlídky. To znamená, že pokud uživatel bude opakovaně zobrazovat obrazovku s výpisem prohlídek bez zakládání nových, nebude nutné klást na aplikaci další výpočetní nároky.

Tímto způsobem byly v aplikaci implementovány veškeré dotazy na data v metodách *mapStateToProps*.

Recompose

Další přínosným nástrojem z pohledu optimalizace výkonu, ale i velikosti codebase je knihovna Recompose. Ta poskytuje již implementované HOC podle nejčastěji používaných vzorů v *ReactJS*.

8.5 *Produkční verze aplikace*

Před generováním produkční verze aplikace pro Android je potřeba vytvořit zabezpečený klíč, kterým aplikace bude podepsána. Tento krok je nezbytný pokud má být finální aplikace publikována pomocí online obchodu Google Play.

Klíč lze vygenerovat pomocí nástroje *keytool* anebo přímo vytvořit v Android Studio. Vygenerovaný klíč je podle oficiální dokumentace platný 10 000 dní. Po vygenerování klíče byla vytvořena konfigurace *signingConfigs* pro *release*, viz následující ukázku kódu.

```
signingConfigs {
    release {
        if (project.hasProperty('EOBCHODNIK_RELEASE_STORE_FILE')) {
            storeFile file(EOBCHODNIK_RELEASE_STORE_FILE)
            storePassword EOBCHODNIK_RELEASE_STORE_PASSWORD
            keyAlias EOBCHODNIK_RELEASE_KEY_ALIAS
            keyPassword EOBCHODNIK_RELEASE_KEY_PASSWORD
        }
    }
}
```

Ukázka kódu 28 Konfigurace release

Jednotlivé proměnné jsou potom definovány v souboru *gradle.properties.* Samotné sestavení lze spustit pomocí `gradlew assembleRelease`.

Velikost výsledného APK aplikace generovaná pro platformu Android je 15,1 MB. Po minifikaci výsledného balíčku pomocí příznaku *minifyEnabled* se velikost zmenšila na 13,7 MB, což se stále více nežli původní aplikace, která má 6,7 MB. Po profilaci výsledného APK, kterou lze provést pomocí nástroje v Android Studio, bylo zjištěno, že největší část zabírají knihovny *.so, tedy nativní moduly, kde knihovna Realm zabírá zhruba polovinu z této velikosti, tedy téměř 5 MB. Další možností optimalizace je generovat APK separátně pro jednotlivé CPU. Potom je možné snížit velikost APK na velikost v rozsahu 7–9 MB. Rozdíl velikostí aplikací po instalaci již není tak markantní. RN aplikace zabírá 32,1 MB a původní aplikace 31,2 MB.

Problémy produkční verze

Problémy, které se vyskytly při testování produkční verze aplikace, byly spojeny s překladačem Babel, který se stará o překlad použitého EcmaScript do zpětně kompatibilních verzí a konkrétně pak té, kterou umí zpracovat dané JS běhové prostředí dané verze RN. Překlad některých předpisů neprobíhá korektně, což následně způsobovalo pád produkční verze aplikace bez jakékoliv signalizace o chybě. Konkrétně se potom jednalo o operátor **, který byl zpětně nahrazen *mod*. Dalším skrytým problémem byl dynamický řetězec v předpisu *required* z knihovny MomentJS, při použití předem definovaného *locale*. Chyby v knihovnách lze vyřešit vlastním *fork*, v tomto konkrétním případě by stačilo explicitně importovat danou lokalizaci. Předpis *import* je potom přeložen do *required* se statickým řetězcem, kdy původní již nemusí být provolán.

U chyb tohoto typu se zpravidla těžko odhaluje, co je způsobilo. Je proto vhodné mít vypnutou minifikaci při sestavení aplikace a mít zapnutý jakýkoliv logovací nástroj, např. *Logger* z Android Studio nebo nativní aplikaci *LogCat* dostupnou na Google Play[15]. Výskyt těchto problémů je především u knihoven, které jsou používány primárně pro webový vývoj, z toho důvodu vznikají i verze pro RN, které eliminují syntax, která problémy způsobuje.

[15] https://play.google.com/store/apps/details?id=com.nolanlawson.logcat&hl=cs

Dalším problémem, který se vyskytnul při generování produkční verze aplikace, byla přítomnost pomocných souborů, které generuje modul pro práci s databází Realm. To potom způsobovalo zastavení procesu *app:bundleReleaseJsAndAssets*, takže sestavení nedoběhlo ani po 30 minutách (standardní doba cca 8 minut). Jedná se o soubory, které slouží k prohlížení aktuálního stavu databáze v RealmStudio a další dočasné soubory generované při instalaci aplikace. Po smazání vše proběhlo již v pořádku. Řešení předcházelo několikanásobné mazání adresáře a čistého sestavení.

Dalším problémem, jehož řešení zabralo poměrně velké množství času, bylo duplicitní kopírování *class* souborů do sestavovaného adresáře aplikace, konkrétně pak do *intermediates/classes/release*. Tento problém není neobvyklý, pokud aplikace používá více nativních modulů. V takovém případě pak může docházet ke konfliktům tříd modulů třetích stran, které nativní moduly používají v různých verzích. Zpravidla s každou přidanou knihovnou byl řešen tento problém opakovaně. Gradle nenabízí nástroj, který by příčinu takového konfliktu automaticky odhalil, zvláště když se jedná o multi-modulární projekt. Bylo potřeba ručně vypisovat závislosti příkazem *gradlew <název modulu>:dependencies*. Výpis závislostí je potom nutné analyzovat a konfliktní moduly extrahovat u daného nativního modulu přímo v aplikačním *build.gradle*.

Zvláštní problém nastával při sestavování aplikace k produkčním účelům. Při každém sestavení Gradle sestavuje nejdříve moduly, na kterých je hlavní modul závislý, následně generuje *bundle,* a nakonec sestavuje a komprimuje finální instalační soubor apk. Problém byl způsoben kopírováním veškerých zkompilovaných *.class souborů nativních knihoven duplicitně do cílového build adresáře aplikace. Samotné kopírování je prováděno při produkčním sestavení Gradle procesem *transformClassesWithDexForRelease*, který je spouštěn pro každý modul. Nejprve tedy byly soubory kopírovány postupně se sestavením jednotlivých modulů, běh potom skončil s chybou pro první soubor *class*, který byl na cestě *intermediates/classes/release*. To mělo za následek zavádějící chybovou hlášku, která poukazovala na chybu spojenou s modulem *react-native-maps*, která signalizovala konflikt právě tohoto modulu s některým z ostatních, což nebyla

pravda. K odhalení chyby pomohlo prostudovat podrobný výpis podprocesů celého sestavení, které vedlo k odhalení zdroje duplicit. Po mnoha pokusech o nejjednodušší řešení byl nakonec nalezen způsob, jak tomuto problému zabránit a aplikaci sestavit bez chyb pomocí následujících příkazů.

```
gradlew clean
gradlew assembleRelease -x app:transformClassesWithDexForRelease
```

Před každým sestavením je nutné vygenerovaný adresář smazat a až následně spustit vlastní sestavení bez daného podprocesu pro hlavní modul *app*.

8.6 *Ladění*

Ladění RN aplikace je kombinací využíváním nástrojů pro webový i nativní vývoj. V současné době již existují i nástroje vytvořené přímo pro použití s RN. Následující nástroje nebyly využívány vždy najednou, ale spíše podle potřeby a povahy řešeného problému.

Redux-Logger

K výpisu jednotlivých vyvolaných akcí a jejich obsahu byl použit middleware Redux-Logger. Middleware lze nainstalovat pomocí *npm/yarn* a standardně použít pomocí funkce *applyMiddleware*. Po vyvolání akce je zobrazen stav před akcí, název akce s parametry a výsledný stav. Jak už bylo výše zmíněno, tento *middleware* musel být zcela odstraněn před generováním produkční verze aplikace.

Remote-Redux-Devtools

Pro využívání rozšíření pro Redux přímo v prohlížeči Google Chrome lze využít nástroj *remote-redux-devtools*. Balíček poskytuje funkci *devToolsEnhancer*, která musí být aplikována při volání *createStore* před aplikací middleware k poskytnutí přístupu ke stavu přímo v prohlížeči.

Reactotron

Reactotron je jednoduchý nástroj vytvořený k ladění aplikací využívající ReactJS a RN. Obsahuje několik rozšíření, kterými lze výrazně ulehčit vývoj aplikace.

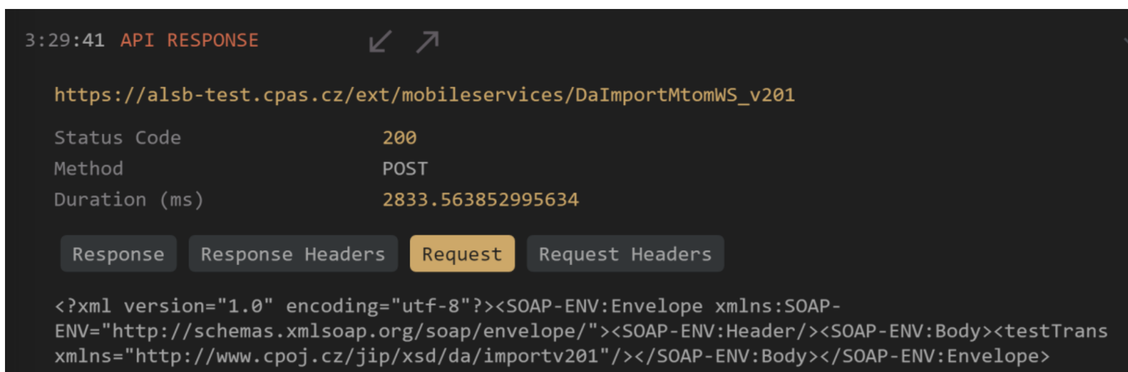
Pro propojení Reactotron s aplikací je nutno vytvořit konfiguraci. První část konfigurace obsahuje informaci o IP zařízení a číslo portu, které zajišťují správné připojení přes komunikační kanál k odchytávání událostí. Další částí konfigurace je specifikace konkrétních nástrojů, které si vývojář přeje používat. Pro konkrétní ukázku konfigurace viz následující ukázku kódu.

```
Reactotron.configure({
  name: 'eObchodnik',
  port: 9090,
  host: 'localhost',
})
  .use(reactotronRedux()).use(networking()).useReactNative().connect();
```

Ukázka kódu 29 Konfigurace Reactotron

V tomto projektu bylo využito dvou rozšíření, a to *networking* a *reatotronRedux*. Networking lze využít k logování veškeré síťové komunikace v aplikaci. Lze navíc definovat volání, která budou ignorována podle typu obsahu nebo url. Samotný log pak vypadá jako na obrázku č. 16. Jsou zobrazeny základní informace o volání, hlavičky a odpověď serveru. Podobně jako tomu je v nástroji Networking v Chrome DevTools, který není možné pro RN použít.
ReactotronRedux slouží pro odchytávání akcí vyvolaných Redux. Nástroj zobrazí přehledně jednotlivé akce v pořadí, v jakém byly vyvolány, se základními informacemi o názvu akce, parametrech a výsledném stavu. Velkým přínosem je možnost akci znovu zavolat anebo provolat vlastní akci s libovolnými parametry dle potřeby a sledovat výsledek v aplikaci.

Výhodou je také možnost uložení stavu aplikace a pozdějšího nahrání, která poskytuje urychlení ladění problémů určité situace, jež v aplikaci nastane bez potřeby opakovat kroky, které jí předcházely. Většinu času si však vývojář bez tohoto nástroje vystačí, pokud využívá *Redux logger middleware*.



Obrázek 16 Networking

Jest

K testování chování komponent a dalších funkcionalit aplikace byla použita JS knihovna Jest. Každá testovaná komponenta má ve svém adresáři adresář *__tests__* s příslušným testem. Jest poskytuje možnost snadno otestovat vzhled komponent, co se týče struktury, a zároveň aplikační logiku aplikace.

Ke snadnému testování byla rozšiřující logika komponent vyčleněna do separátních souborů zpravidla pojmenovávaných jako **-utils*. To zajišťovalo přehlednost a snadnější testovatelnost jednotlivých částí aplikace. Praktickou možností je použití funkce *watch*, kdy dochází ke spouštění požadovaných testů po změně souborů, jejichž funkce jsou testovány. K urychlení vývoje mnoha částí aplikací byl pak aplikován TDD přístup.

Složitější struktury a většina komponent byly testovány pomocí snapshot za použití *shouldMatchSnapsot*. Jest automaticky generuje výslednou strukturu. V případě dalšího testování a případné neshodě přehledně vypisuje rozdíl ve výsledku. Pokud je změna požadována, lze snapshot aktualizovat pomocí stisknutí klávesy *u*. Jednotlivé funkce aplikační logiky byly rozděleny tak, aby byl vždy jasný a testovatelný výsledek.

Výhodou použitých IDE je podpora právě pro Jest a spouštění jednotlivých testů přímo v prostředí včetně módu *debug*. Spuštění testů lze také provést přímo z konzole pomocí příkazu *jest*. Výhodou je rychlost a velká škála funkcí, které Jest poskytuje.

8.6.1 React DevTools

K procházení a ladění aplikace z pohledu DOM byla použita samostatná aplikace React DevTools, která přináší stejné benefity jako stejnojmenné rozšíření do prohlížeče Google Chrome při vývoji standardní aplikace v ReactJS. Důležitým nástrojem, a to především z hlediska optimalizace, je *Highlight Updates*, který zvýrazňuje ty komponenty, u nichž v dané chvíli dojde k překreslení. Lze pak snadno odhalit možné výkonnostní problémy v aplikaci. I přes to, že u mobilních aplikací není zpravidla na jedné obrazovce vykreslen takový počet elementů, jako tomu je u jednostránkových webových aplikací, je doporučeno i tyto malé problémy řešit. Konkrétní řešení bylo popsáno v kapitole *Optimalizace*.

8.7 *Continuous integration*

Projekt využívá verzovacího nástroje git hostovaného webovou aplikací Bitbucket. Kromě standartních východ, které git přináší, je zde také podpora vytváření dokumentace projektu ve Wiki a zakládání Issues, které lze následně provázat s každou změnou kódu a zajistit tak lepší zpětný přehled o provedených změnách.

Přínosným a při vývoji důležitým nástrojem jsou Pipelines, kterým je možné snadno zajistit proces CI celého projektu. Bitbucket poskytuje již nakonfigurované základní obrazy nástroje Docker, které je možné použít. V tomto konkrétním případě byl využit obraz pro NodeJS projekt stejné verze používané na lokálním zařízení – 8.6.0. Jednotlivé kroky, které jsou spouštěny, byly specifikovány do souboru *bitbucket-pipelines.yml* v kořenovém adresáři projektu. Samotná konfigurace obsahuje sestavení aplikace, spuštění testů Jest a nakonec kontrolu kódu pomocí ESLint. Automatické spuštění tohoto procesu probíhá po každé aktualizaci kódu v projektu a probíhá pro každou větev kódu zvlášť. V případě neúspěšného procesu odešle Bitbucket e-mail s bližší specifikací chyby.

```yaml
image: node:8.6.0

pipelines:
  default:
    - step:
        caches:
          - node
        script:
          - cd react_native_app/eObchodnik && npm i
          - npm run test
          - npm run lint
```

Ukázka kódu 30 Konfigurace pipeline v Bitbucket

8.8 *Testování*

Jak už bylo výše zmíněno, aplikace byla v současné době testována pro platformu Android na různých zařízeních pro iOS tomu bylo pouze na simulátoru zařízení typu iPhone. Většina problémů s výkonem, která u méně výkonných zařízení nastávala, se v menší míře vyskytovala i u nativní aplikace. Celkový dojem hybridní aplikace byl uspokojující a ve většině případů nebyl poznat rozdíl. Pro platformu iOS bylo řešení testováno konkrétně pro iPhone 7 a 8, kde byla odezva aplikace velmi rychlá. Porovnání vzhledu jednotlivých komponent je přehledně znázorněno v příloze A.

V případech, kdy byla explicitně zmíněna různorodost pro jednotlivé platformy, byl tento problém ošetřen. Většinou se odlišnost týkala platformy Android než iOS. Pro iOS musely být některé definice stylů striktnější a definovat více parametrů.

9 Shrnutí výsledků

Práce shrnuje moderní přístupy k vývoji mobilních aplikací a popisuje principy jednotlivých řešení. Každý přístup má svá pro a proti. O vhodnosti konkrétního řešení může do určité míry rozhodnout i to, jak je ke konkrétnímu vývoji přistupováno.

Z analýzy původní aplikace vyvstalo několik kritických bodů, se kterými se bylo nutné v průběhu implementace vypořádat. Jednalo se převážně o práci s nativními funkcemi na zařízeních a komunikací s okolními službami. Velká část vývoje byla věnována optimalizaci jednotlivých částí aplikace tak, aby bylo dosaženo co nejlepšího výsledku.

Při řešení jednotlivých případů užití byla velmi nápomocná komunita okolo React Native, která za poměrně krátkou dobu zvládla vyprodukovat velké množství kvalitních řešení běžně používaných funkcí. Za předpokladu, že by jednotlivé přepoužité nativní moduly nebyly k dispozici, nebylo by nejspíš toto řešení časově výhodnější než nativní verze. Proto je právě komunita nepochybně jednou z nejsilnějších stránek RN.

Některé z problémů při vývoji vycházely ze seznamování se s novou technologií. Časové náklady, které byly spojeny s učením se nové technologii, by tedy byly ušetřeny při dalším vývoji, či za předpokladu, že by vývojář již takové zkušenosti měl. Na druhou stranu velkým přínosem byla znalost principů ReactJS a Redux.

Práce prezentuje možný přístup využívání jednotlivých nativních modulů. Z pohledu vývojáře pro nativní Android je velkým přínosem uchopení definice jednotlivých UI prvků a možnost používání CSS. Velkým přínosem je také rychlost vývoje aplikace spojená s možností Hot Reload. Kombinace JS a funkcionálního přístupu knihovny Ramda mělo za následek výraznou redukci často opanovávaného kódu, jako tomu bývá u nativního Android.

Podařilo se vytvořit funkční aplikaci s multiplatformním řešením pro jednotlivé případy použití. Aplikace byla psána tak, aby bylo možné její snadné rozšíření. Tato práce ukázala to, že multiplatformní cesta pomocí React Native je pro aplikace České pojišťovny nepochybně vhodná.

10 Závěry a doporučení

React Native je stále se rozvíjející řešení pro mnoho mobilních aplikací, pomocí kterého lze často ušetřit velké množství času věnovaného implementací pro více platforem. Pro rozhodnutí, zda je toto řešení vhodné pro konkrétní aplikaci, existují různé nástroje, které mohou pomoci k přehledu o současné situaci a podpoře jednotlivých funkcionalit.

Úskalím může být to, že React Native je mladá knihovna, která bohužel stále obsahuje větší množství chyb. Jednotlivé nativní moduly, které byly v průběhu vývoje používány, byly proto často upravovány a přizpůsobovány konkrétním požadavkům. Na druhou stranu má React Native stále rostoucí skupinu příznivců, kteří do společné komunity přispívají. Problémy, které se v průběhu objevovaly, byly při finalizaci práce z velké části opraveny. S vývojem RN jsou postupně představována řešení, která by byla pro určité části aplikace vhodnější. Tento fakt poukazuje na to, že má aplikace z pohledu optimalizace v některých částech rezervy.

Aplikace eObchodník je středně velkou aplikací, pro kterou bylo nalezeno uspokojivé multiplatformní řešení založené na technologii React Native. V případě, že by se aplikace dále významně rozrůstala o složitější funkcionality, které by vyžadovaly zásah do nativních částí aplikace, by mohlo být multiplatformní řešení lehce kontraproduktivní.

V nové aplikaci eObchodník je nutné dále doplnit některé dílčí funkcionality, které nebyly předmětem této práce. Tyto funkcionality nejsou kritické. Při dodržení daných standardů aplikace by neměl nastat moment, kdy výsledek bude nepoužitelný.

I přes velkou skupinu odpůrců je React Native velmi povedená knihovna, která i nadále bude mít významnou pozici ve světě multiplatformních přístupů k vývoji mobilních aplikací.

11 Seznam použité literatury

[1] KINIK, Dicle Cagla. *Mobile App Development Choosing Between Web Native And Hybrid* [online]. 2018 [Citace 2018-07-02]. Dostupné z: https://appsamurai.com/mobile-app-development-choosing-between-web-native-and-hybrid

[2] MOHAMED, LACHGAR a ABDALI ABDELMOUNAÏM. Decision Framework for Mobile Development Methods. *International Journal of Advanced Computer Science and Applications* [online]. 2017, **8**(2). ISSN 21565570. Dostupné z: doi:10.14569/IJACSA.2017.080215

[3] STATISTA. *Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2018* [online]. 2018 [Citace 2018-08-02]. Dostupné z: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[4] PEAL, Gabriel. *React Native at Airbnb* [online]. 2018 [Citace 2018-07-01]. Dostupné z: https://medium.com/airbnb-engineering/react-native-at-airbnb-f95aa460be1c

[5] ANGULO, Esteban, Xavier FERRE, Felix APPIAH, Joseph K PANFORD, Cross-platform APPS, Author CARLOS, Sirvent MAZARICO, Marc ARMGREN, Marc ARMGREN, Eleanor BASH, Nader BOUSHEHRINEJADMORADI, Vinod GANAPATHY, Santosh NAGARAKATTE, Liviu IFTODE, Salma CHARKAOUI, Zakaria ADRAOUI, El HABIB BENLAHMAR, Shauvik Roy CHOUDHARY, Isabelle DALMASSO, Soumya Kanti DATTA, Christian BONNET, Navid NIKAEIN, Lisandro DELIA, Nicolas GALDAMEZ, Pablo THOMAS, Leonardo CORBALAN, Patricia PESADO, Jared DICKSON, Wafaa S. EL-KASSAS, Bassem A. ABDULLAH, Ahmed H. YOUSEF, Ayman M. WAHBA, D I FH, Norbert HABERL, Mona Erfani JOORABCHI, Mohamed ALI, Ali MESBAH, Olivier LE GOAER, Sacha WALTHAM, Guangtai LIANG, Jian WANG, Shaochun LI, Rong CHANG, Virgin MEDIA, Virgin MEDIA, Average UK, Gebremariam MESFIN, Tor Morten GRØNLI, Dida MIDEKSO, Gheorghita GHINEA, OFCOM, Manuel PALMIERI, Inderjeet SINGH, Antonio CICCHETTI, C. P. RAHUL RAJ, S. B. TOLETY, Paruj RATANAWORABHAN, Benjamin LIVSHITS, Benjamin G. ZORN, Shauvik ROY CHOUDHARY, Mukul R. PRASAD, Alessandro ORSO, Dat Do TEXAS a Corpus CHRISTI. Multi-Platform Mobile Application Development Analysis. *05/08/2015* [online]. 2015, **44**(February), 642–645. ISSN 09758887. Dostupné z: doi:10.1016/j.csi.2015.08.004

[6] LARSSON, Thimmy a Jonas WEDIN. *A comparison between cross- compiler and native development for mobile applications*. B.m., 2017. b.n.

[7] MAJCHRZAK, Tim A, Andreas BIØRN-HANSEN a Tor-Morten GRØNLI. Progressive Web Apps: the Definite Approach to Cross-Platform Development? *Proceedings of the 51st Hawaii International Conference on System Sciences* [online]. 2018, 5735–5744. Dostupné z: doi:10.24251/HICSS.2018.718

[8] MERCADO, Iván Tactuk, Nuthan MUNAIAH a Andrew MENEELY. The impact of cross-platform development approaches for mobile applications from the user's perspective. In: *Proceedings of the International Workshop on App Market Analytics - WAMA 2016* [online]. 2016. ISBN 9781450343985. Dostupné z: doi:10.1145/2993259.2993268

[9] APPS, Cross-platform Mobile. Studying the Perceived Quality Consistency of Cross-Platform Mobile Apps. 2017, (May).

[10] AJIT KUMAR, N, K T Hari KRISHNA a R MANJULA. Challenges and Best Practices in Mobile Application Development. *Imperial Journal of Interdisciplinary Research* [online]. 2016, **2**(12), 2454–1362. ISSN 2454-1362. Dostupné z: doi:10.1145/3093241.3093245

[11] SHASHIDHARA, Sunilbagunji. *Mobile Performance Profiling* [online]. 2013 [Citace 2018-04-01]. Dostupné z: https://www.mindtree.com/blog/mobile-performance-profiling

[12] DEVELOPERS, Google. *Profile your app performance* [online]. 2018 [Citace 2018-06-01]. Dostupné z: https://developer.android.com/studio/profile/

[13] ELLIOT, Ian. *I Programmer - NativeScript 1.0.0 Released* [online]. 2015 [Citace 2018-01-14]. Dostupné z: http://www.i-programmer.info/news/167-javascript/8561-nativescript-100-released.html

[14] ADOBE. *Native Script Documentation* [online]. 2018 [Citace 2018-01-11]. Dostupné z: https://docs.nativescript.org/

[15] CORPORATION, Progress Software. *How to Build NativeScript Apps That Start Up Fast* [online]. 2018 [Citace 2018-01-20]. Dostupné z: https://docs.nativescript.org/best-practices/startup-times

[16] ALTEXSOFT. *The Good and The Bad of Xamarin Mobile Development* [online]. 2017 [Citace 2018-01-10]. Dostupné z: https://www.altexsoft.com/blog/mobile/the-good-and-the-bad-of-xamarin-mobile-development/

[17] MICROSOFT. *Xamarin Documentation* [online]. 2018 [Citace 2018-01-21]. Dostupné z: https://docs.microsoft.com/en-gb/xamarin/

[18] HAMEDANI, Mosh. *Xamarin Forms: Build Native Cross-platform Apps with C#* [online]. 2018 [Citace 2018-02-05]. Dostupné z: https://www.udemy.com/xamarin-forms-course/

[19] MICROFSOFT. *IDisposable Interface* [online]. 2018 [Citace 2018-01-25]. Dostupné z: https://msdn.microsoft.com/en-gb/library/system.idisposable(v=vs.110).aspx

[20] APACHE. *Cordova: Documentation* [online]. 2018 [Citace 2018-01-05]. Dostupné z: https://cordova.apache.org/docs/

[21] FACEBOOK. *Facebook Open Source* [online]. 2018 [Citace 2018-07-01]. Dostupné z: https://opensource.fb.com/#frontend

[22] W3SCHOOLS. *DOM* [online]. 2018 [Citace 2018-06-20]. Dostupné z: https://www.w3schools.com/js/js_htmldom.asp

[23] INC., Facebook. *DOM Elements* [online]. 2018 [Citace 2018-06-01]. Dostupné z: https://reactjs.org/docs/dom-elements.html

[24] LI, Sing. *React: Create maintainable, high-performance UI components* [online]. 2015 [Citace 2018-08-01]. Dostupné z: https://www.ibm.com/developerworks/library/wa-react-intro/index.html

[25] KURIAN, Gethyl George. *How Virtual-DOM and diffing works in React* [online]. 2017 [Citace 2018-04-02]. Dostupné z: https://medium.com/@gethylgeorge/how-virtual-dom-and-diffing-works-in-react-6fc805f9f84e

[26] FACEBOOK. *Optimizing Performance* [online]. 2018 [Citace 2018-04-01]. Dostupné z: https://reactjs.org/docs/optimizing-performance.html

[27] FACEBOOK. *React Native Docs* [online]. 2018 [Citace 2018-05-20]. Dostupné z: https://facebook.github.io/react-native/docs/getting-started

[28] HEARD, Pete. *React Native Architecture : Explained!* [online]. 2017 [Citace 2018-06-04]. Dostupné z: https://www.logicroom.co/react-native-architecture-explained/

[29] AIRBNB. *React Native Maps* [online]. 2018 [Citace 2018-02-01]. Dostupné z: https://github.com/react-community/react-native-maps

[30] FACEBOOK. *Metro Packanger* [online]. 2018 [Citace 2018-05-02]. Dostupné z: https://facebook.github.io/metro/docs/en/getting-started

[31] FACEBOOK. *JavaScript Environment* [online]. 2018 [Citace 2018-06-01]. Dostupné z: https://facebook.github.io/react-native/docs/javascript-environment.html

[32] GHIASSY, Shaheen. *Deep Diving React Native Debugging* [online]. 2015 [Citace 2018-07-24]. Dostupné z: https://medium.com/@shaheenghiassy/deep-diving-react-native-debugging-ea406ed3a691

[33] REDUXJS. *Redux* [online]. 2018 [Citace 2018-02-02]. Dostupné z: https://redux.js.org

[34] REDUX. *Usage with react* [online]. 2018. Dostupné z: https://redux.js.org/basics/usage-with-react

[35] REDUXJS. *React-Redux* [online]. 2018 [Citace 2018-04-01]. Dostupné z: https://github.com/reduxjs/react-redux/blob/master/docs/api.md

[36] FACEBOOK. *React Native Flexbox* [online]. 2018 [Citace 2018-06-15]. Dostupné z: https://facebook.github.io/react-native/docs/flexbox

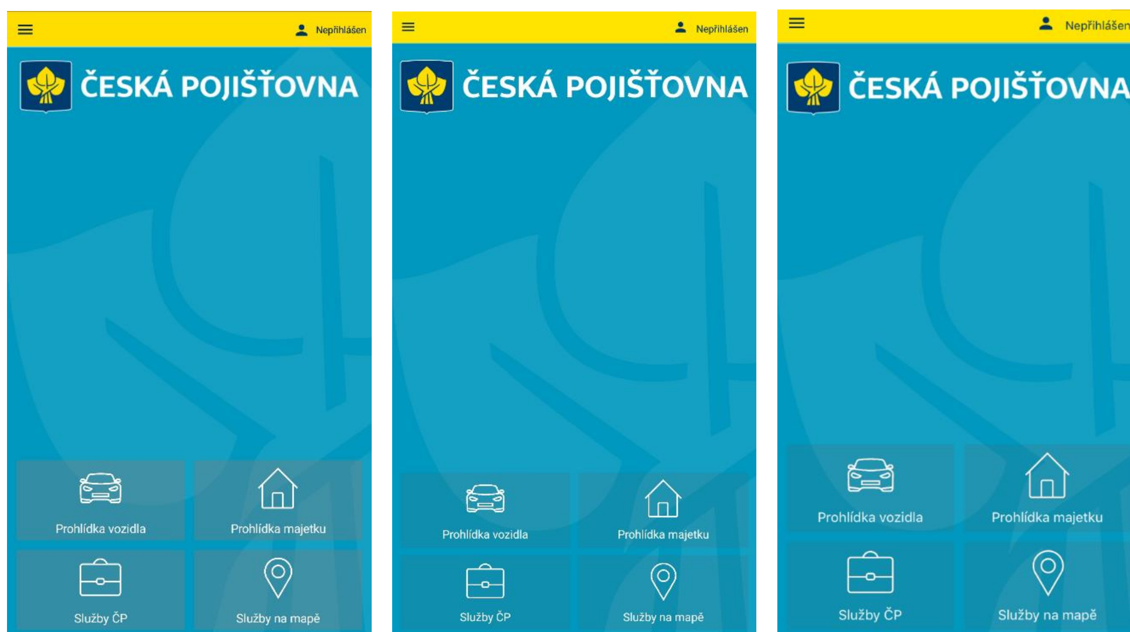
[37] W3SCHOOLS. *XML Soap* [online]. 2018 [Citace 2018-04-02]. Dostupné z: https://www.w3schools.com/xml/xml_soap.asp

[38] ALEX RODRIGUEZ. *Restful* [online]. 2015 [Citace 2018-05-01]. Dostupné z: https://www.ibm.com/developerworks/webservices/library/ws-restful/

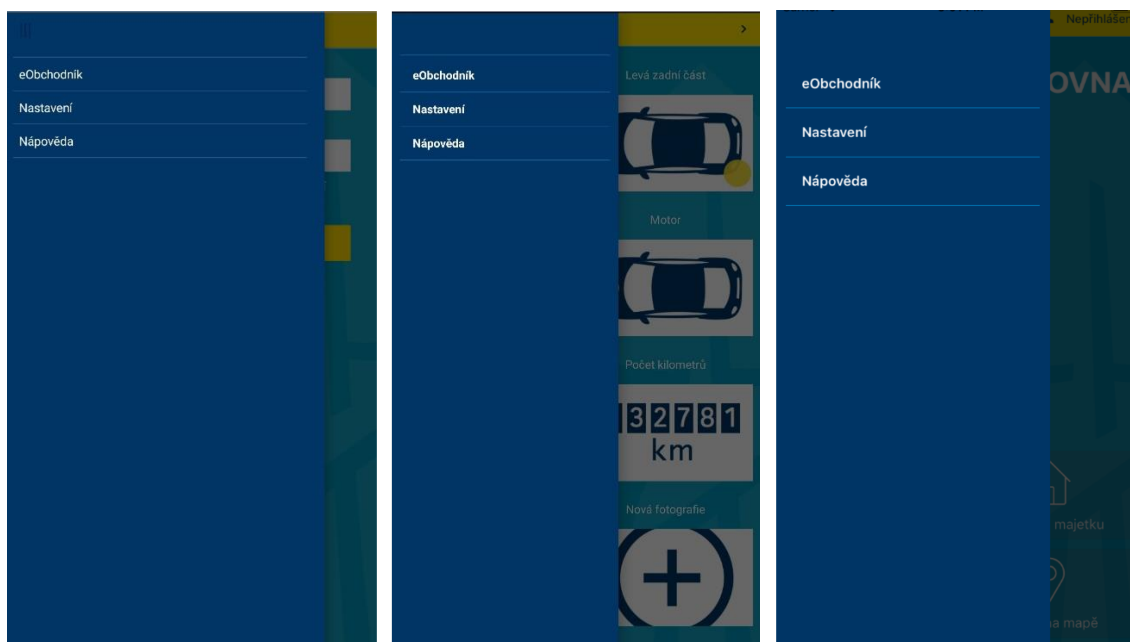
[39] KONRÁDY, Tomáš, Lukáš SULÍK a Jiří BRÁDLE. *Ramda Extension* [online]. 2017 [Citace 2017-12-10]. Dostupné z: https://github.com/tommmyy/ramda-extension

Seznam příloh

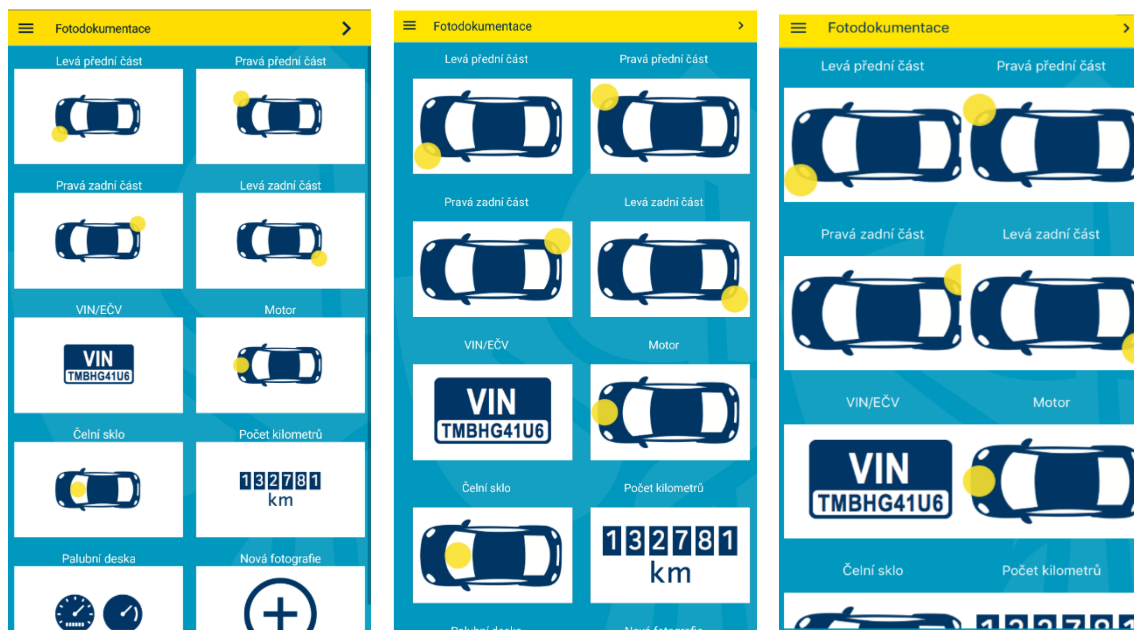
A. Snímky obrazovek aplikace
B. Obsah kompaktního disku



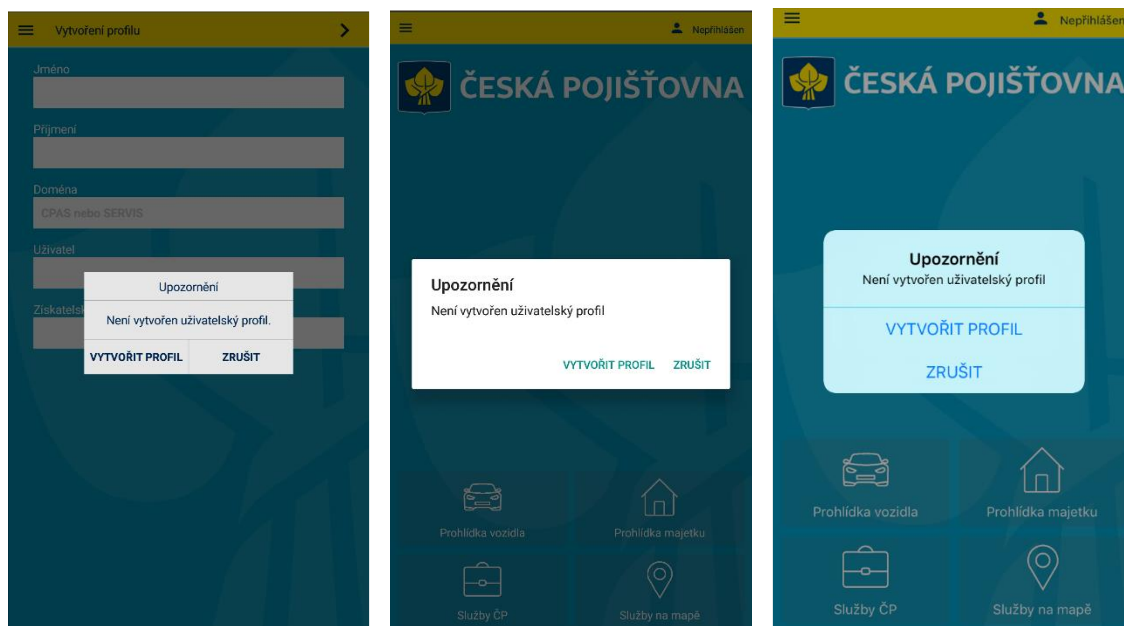
Úvodní obrazovka aplikace (nativní Android, RN Android, RN iOS)



Boční menu aplikace (nativní Android, RN Android, RN iOS)

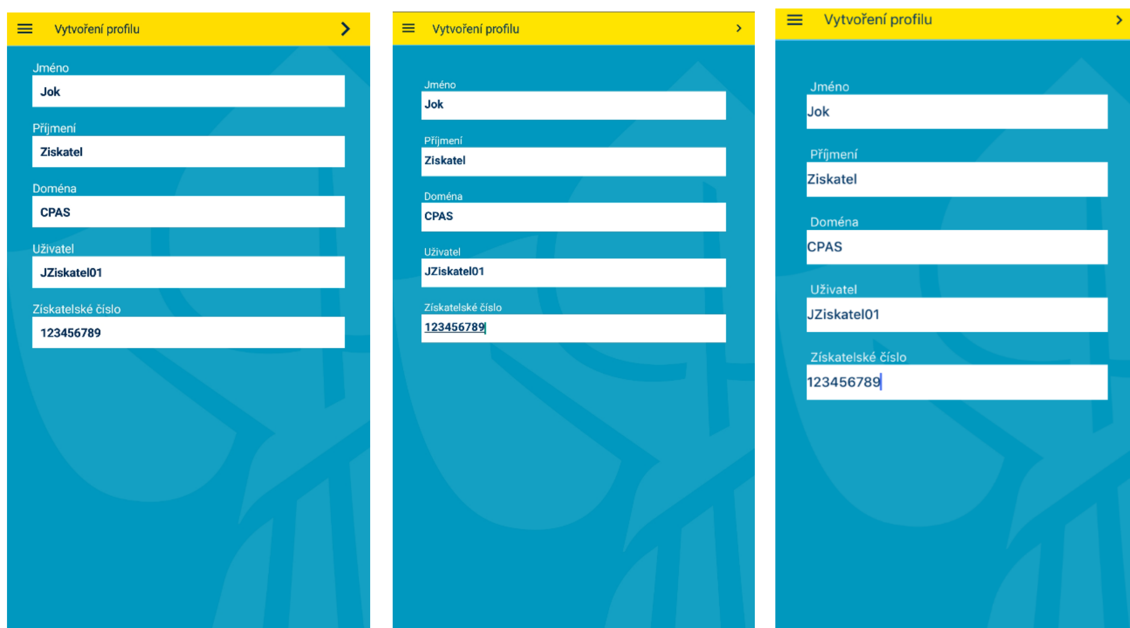


Požadované fotografie pro prohlídku vozidla (nativní Android, RN Android, RN iOS)

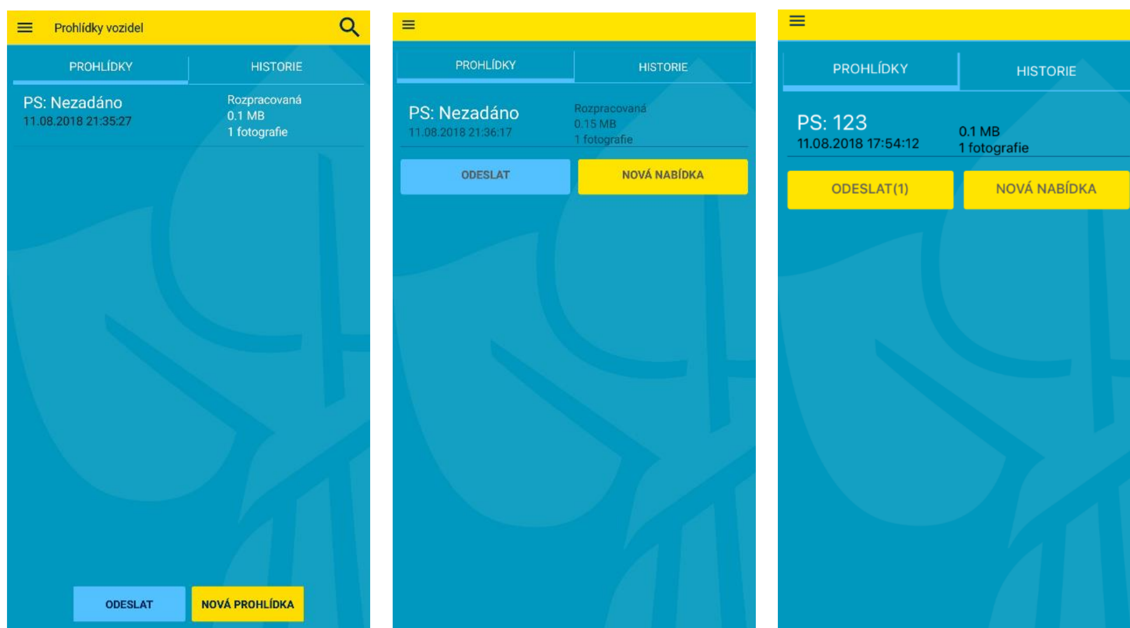


Alert (nativní Android, RN Android, RN iOS)

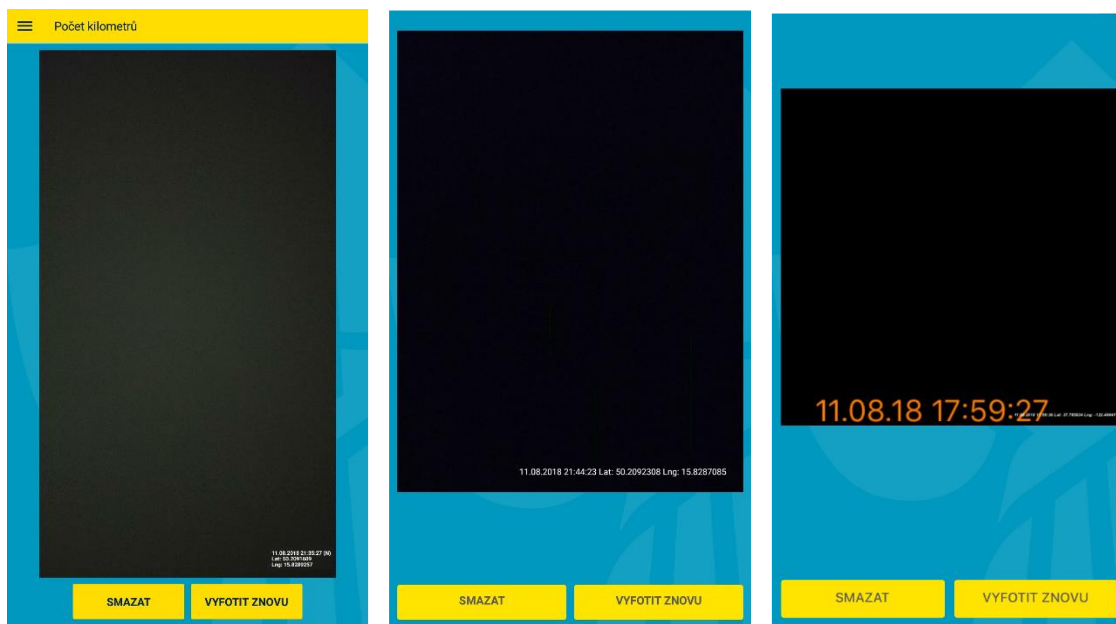
Komponenty tlačítka (nativní Android, RN Android, RN iOS)



Formuláře (nativní Android, RN Android, RN iOS)

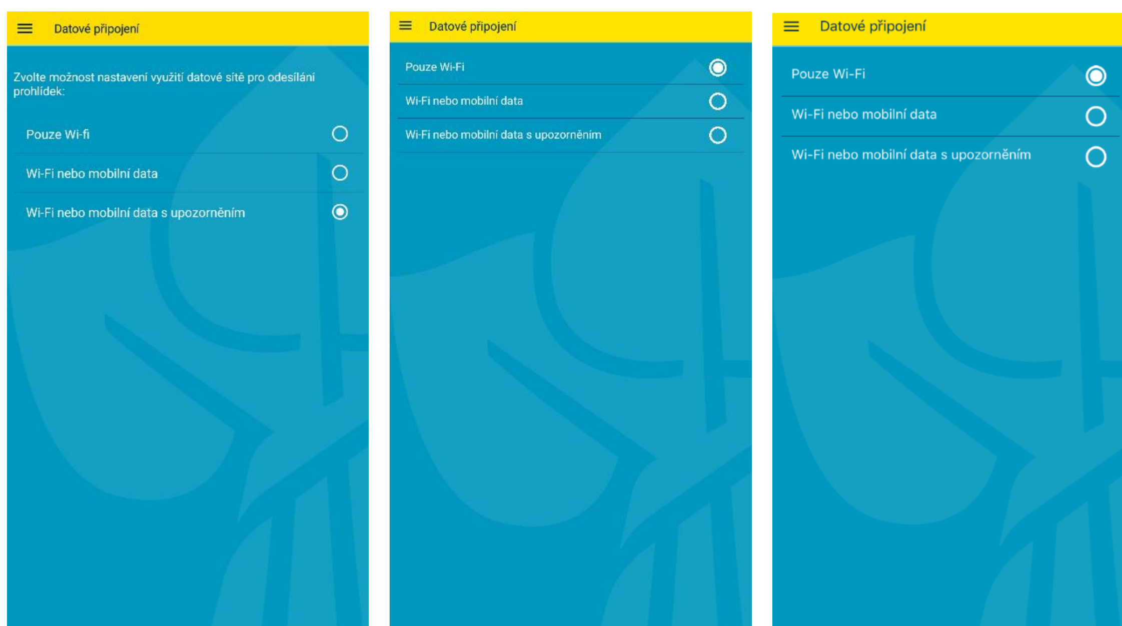


Přehled prohlídek (nativní Android, RN Android, RN iOS)



Náhled sejmuté fotografie (nativní Android, RN Android, RN iOS)

Nápověda ke službám na mapě (nativní Android, RN Android, RN iOS)



Nastavení připojení (nativní Android, RN Android, RN iOS)

Popis obsahu kompaktního disku

Součástí práce je přiložený kompaktní disk se zdrojovými kódy a vygenerovaný instalační soubor aplikace pro Android.

Spuštění aplikace pro vývoj (Android)

Před samotným spuštěním aplikace je potřeba nainstalovat NodeJS s verzí 8 a vyšší. Následně nainstalovat yarn.

```
npm i -g yarn
```

Nejprve je nutné nainstalovat všechny závislosti a poté spustit aplikaci.

```
cd ./eObchodnik && yarn && rect-native run-android
```

Následně by měl být spuštěn lokální balíčkovací server a sestavení aplikace. Pro automatické nainstalování na zařízení (nezbytné u prvního spuštění) je nutné mít připojeno testovací zařízení (může být i emulator).

Při následovných testech je pouze možné spustit aplikaci a ze zaříjení se na balíček nasměrovat pomocí možnosti v Dev Settings -> Debug server host & port for service zadáním informace ve tvaru

```
ip:port např. 10.0.0.3:8080
```

Mobilní a hostující zařízení je nezbytné připojit ke stejné síti. Pozn.: Na síti *eduroam* se nepodařilo bezdrátové připojení uskutečnit.

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2017/2018

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bc. Kamenická Kristýna	Velká 166, Hradec Králové - Pouchov	I1500689

TÉMA ČESKY:

Vývoj mobilní aplikace v React Native

TÉMA ANGLICKY:

React Native Mobile Application Development

VEDOUCÍ PRÁCE:

Ing. Pavel Kříž, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl: Cílem práce bude zanalyzovat stávající řešení nativní aplikace a navrhnout nové multiplatformní řešení s použitím React Native. Výsledkem pak bude ukázková aplikace, která bude obsahovat a prezentovat funkční řešení vybraných kritických případů užití původní aplikace. Práce bude sloužit i jako technická dokumentace, která bude popisovat řešení jednotlivých částí a bude sloužit jako primární podklad k případnému dalšímu vývoji.

Osnova:
Úvod
Multiplatformní vývoj, dostupné technologie
Popis stávajícího řešení
Návrh a implementace vlastní řešení v React Native
Výsledky a testování
Závěr

SEZNAM DOPORUČENÉ LITERATURY:

1) https://facebook.github.io/react-native/
2) MERCADO, Iván Tactuk, Nuthan MUNAIAH a Andrew MENEELY. The impact of cross-platform development approaches for mobile applications from the user's perspective. doi:10.1145/2993259.2993268
3) KOS, Anton, Sašo TOMAŽIČ a Anton UMEK. Evaluation of smartphone inertial sensor performance for cross-platform mobile applications. doi:10.3390/s16040477
4) NILSSON, Samuel. Implementation of a Continuous Integration and Continuous Delivery System for Cross-Platform Mobile Application Development [online].

Podpis studenta: .... Datum: 10.5.18

Podpis vedoucího práce: .... Datum: 10.5.18