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

Analýza a využití platformy Firebase pro vývoj mobilních aplikací

Bakalářská práce

Autor: Kryštof Macek
Studijní obor: ai3

Vedoucí práce: Ing. Barbora Tesařová, Ph.D.

Hradec Králové Duben 2020

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 30.4.2020 *Kryštof Macek*

Poděkování:
Rád bych poděkoval své vedoucí Ing. Barboře Tesařové, Ph.D. za odborné vedení práce a cenné rady, které mi pomohly tuto práci zkompletovat.

Anotace

Tato práce se zabývá platformou Firebase, popisem vybraných služeb a demonstrací implementace některých služeb při vývoji mobilní aplikace.
Práce slouží jako představení platformy a jejích služeb, které může programátor využít. Dále by měla pomoci k pochopení fungování poskytovaných služeb a jejich využití, a také při jejich konkrétní implementaci.

Annotation

Title: Analysis of the Firebase platform and its utilization for mobile app development

This paper summarizes information about the Firebase platform. It lists selected services and demonstrates the implementation of several of those services on the example of developing a mobile aplication.
The work serves as an introduction to the platform specifics, which can be used by any programmer looking to integrate it into their projects. It should facilitate an insight into the services provided by the platform and the possibilities of their implementation.

Obsah

1 Úvod

Vývoj mobilních a webových aplikací je dnes jedním z nejrozšířenějších a nejžádanějších odvětví vývoje software. Optimalizace vývoje, jeho urychlení a usnadnění práce individuálních vývojářů jsou zásadními oblastmi, které jsou vývojáři řešeny.

Téměř každá aplikace vyžaduje pro poskytnutí funkčnosti nějakou formu back-endových služeb běžících na serverech. Poskytnutí těchto služeb však může být problematické vzhledem k jejich implementaci zvláště pro nezávislé vývojáře.

S ohledem na tyto požadavky vznikají takzvané BaaS (Backend as Service) platformy, které se snaží tento problém řešit. Baas platformy využívají trendu cloud computingu a distribučních modelů. To jim umožňuje poskytnout implementaci back-endových služeb na vlastních serverech. Vývojáři mohou tyto služby snadno využít při vývoji vlastních aplikací. Mezi populární Baas platformy patří například Firebase, Amazon Web Services, Parse a Kinvey.

Tato práce se blíže zabývá platformou Firebase. V současnosti Firebase spadá pod firmu Google. Své služby rozděluje na tři hlavní kategorie. V první řadě jsou to služby užitečné při vývoji samotné aplikace. Mezi ně patří například Real-time databáze nebo Ověřování uživatelů. Další kategorií jsou služby, které se snaží o zlepšení kvality výkonu vyvinuté aplikace jako Crashlytics nebo Performance monitoring. Poslední kategorie služeb je zaměřena na zlepšení UX jako Cloud Messaging nebo Google Analytics.

Firebase tedy poskytuje služby užitečné pro každou fázi při vývoji aplikace, urychluje jejich vývoj a usnadňuje práci individuálním vývojářům.

2 Cíl práce

Cílem této práce je přehledně zpracovat a popsat služby platformy Firebase a poskytnout tak základ pro vývojáře, kteří mají zájem platformu využívat ve vlastních aplikacích. Na základě získaných informací budu demonstrovat využití platformy a implementaci vybraných služeb při vývoji mobilní aplikace. Tato aplikace bude umožňovat navázání kontaktů, komunikaci a vyhledávání uživatelů na základě specifických zájmů a geolokace.

3 Metodika zpracování

Bakalářská práce se skládá ze dvou hlavních částí, kterými jsou teoretická a praktická.

K vypracování teoretické části jsem využil literární rešerše, přičemž hlavními zdroji byla oficiální dokumentace platformy Firebase a online články zabývající se problematikou BaaS a platformou Firebase. Teoretická část se skládá ze čtyř kapitol, které poskytují potřebné informace k pochopení platformy a jejích technologií. Zároveň slouží jako podklad pro část praktickou.

První kapitola teoretické části se věnuje úvodnímu představení služby BaaS a platformy Firebase. Popisuje základní funkcionality, způsob kategorizace a cenovou politiku platformy Firebase.

Druhá Kapitola pojednává o problematice správy uživatelů, ověření přístupu a implementaci této služby ve Firebase.

Třetí kapitola se zaměřuje na databáze, představuje základní typy a využívané technologie jako SQL, NoSQL, Real-time a Cloud databáze. Podrobněji se pak zabývá NoSQL Cloud databázemi, které jsou součástí platformy Firebase.

Čtvrtá kapitola teoretické části práce, se zabývá doplňujícími službami platformy Firebase. Obecně se tyto služby zaměřují na optimalizaci aplikace a poskytují aplikacím rozšiřující funkcionalitu.

Praktická část primárně slouží pro demonstraci implementace a využití služeb popsaných v teoretické části. Pro tento účel bude vytvořena aplikace, implementující služby pro správu uživatelů (Firebase autentizace), NoSQL real-time cloud databázi (Cloud Firestore) a rozšiřující služby (Crashlytics, Performance Monitoring, Cloud Messaging a Analytics).

4 Představení služby BaaS a platformy Firebase

V současné době existuje mnoho služeb řadících se mezi takzvané BaaS (Back-end-as-Service) a jednou z těchto služeb je právě platforma Firebase. V této kapitole si představíme služby BaaS, jejich funkcionalitu a způsoby rozdělení. A současně se podíváme, jak mezi ně zapadá platforma Firebase.

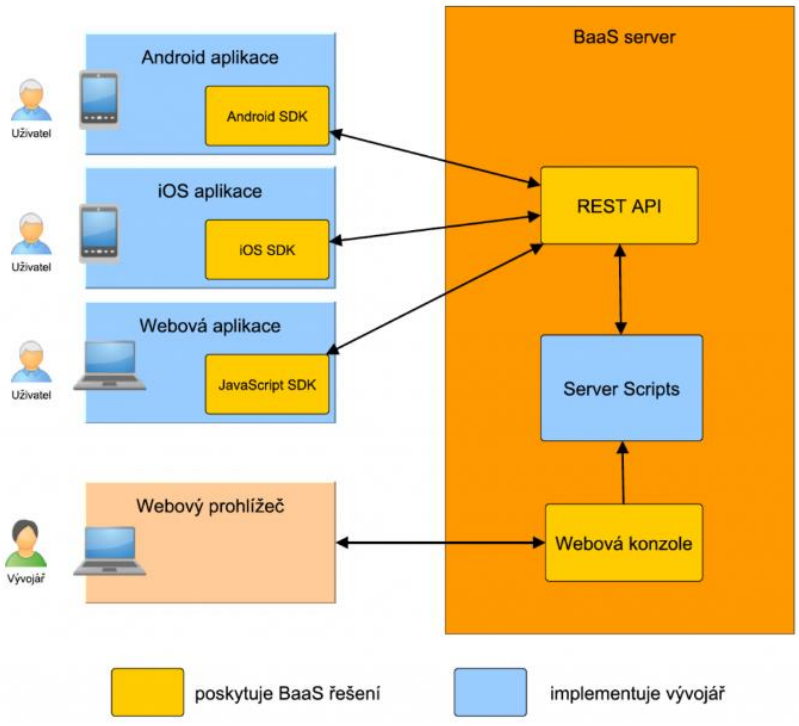
4.1 Služba BaaS

BaaS je cloudová služba poskytující vývojářům připojení aplikací k implementaci backendových funkcionalit na vzdálených serverech poskytovatelů těchto služeb. BaaS se obecně skládá ze tří komponent:

Serverová část je přístupná prostřednictvím webového rozhraní.

REST API zpřístupňuje funkcionalitu serverové části a umožňuje vývojáři připojit k němu svoji aplikaci pomocí SDK.

SDK pracuje s daným REST API, které usnadňuje jeho použití a zefektivňuje tak práci s BaaS [9].



Obrázek 1 – Architektura a komponenty BaaS řešení [9]

4.2 Základní funkcionality BaaS

Jednotlivá řešení mají různé množiny nabízených služeb, avšak některé funkce jsou součástí téměř každé BaaS platformy. V první řadě to jsou databáze, nejčastěji se jedná o určitý typ NoSQL databází. Tento typ databází je využíván právě ve službách Firebase. Existují však i BaaS využívající relační databáze. Příkladem je služba Backendless, která využívá MySQL [9]. Podrobněji jsou tyto typy databází a jejich využití v rámci Firebase popsány v kapitole 6. Databáze.

Další službou je správa uživatelů. Ta zahrnuje ukládání autentizačních a autorizačních údajů, implementaci systému pro registraci, přihlášení a dalších souvisejících funkcionalit [9]. Této problematice se věnuje kapitola 5. Správa uživatelů.

Mezi další často poskytované služby patří messaging, umožňující komunikaci s uživatelem mimo samotnou aplikaci, implementaci vlastních scriptů, geolokaci, analytiku a další [9]. Tyto doplňující a rozšiřující služby jsou v souvislosti s platformou Firebase popsány v kapitole 7. Služby pro optimalizaci, rozšíření.

4.3 Platforma Firebase na trhu BaaS

BaaS platformy se dají kategorizovat několika způsoby:

1. Oblast zaměření. Platformy mohou být jednak specializované, což znamená že se soustředí na podporu specifického typu aplikací nebo technologií, nebo univerzální. V tom případě je lze využít pro vývoj různých typů aplikací [1]. Z tohoto hlediska platforma Firebase spadá mezi univerzální.
2. Způsob hostování. V tomto případě se platformy rozlišují dle způsobu, jakým hostují své služby. První možností jsou „Managed“, kde se o hostování stará poskytovatel služby. Dále „On-premise“, v tomto případě poskytovatel nasazuje služby na servery zákazníka a spravuje je. A nakonec „Self-hosted“, kde odpovědnost a řešení nasazení a údržby je zcela na zákazníkovi [1]. V případě Firebase, se jedná o „Managed“ hostování.
3. Způsob účtování. Zde rozdělujeme platformy na základě způsobu zpoplatnění jejich služeb. Obvykle se jedná o tři různé „aspekty“ BaaS platforem, pro které jsou určeny limity, velikost úložiště, počet uživatelů či

zařízení a objem toku dat. Způsob účtování za tyto služby se pak dá rozdělit do tří kategorii [1]:

a. Platba za tarif – Poskytovatel definuje několik úrovní se specifikovanými limity pro dané aspekty.
b. Platba za navýšení limitu – Jednotlivé limity jsou účtovány zvlášť. Základní verze do určitého limitu může být zdarma.
c. „Pay as you go“ – V tomto modelu zákazník platí zvlášť za jednotku každého aspektu. Cena tedy přesně odpovídá využitým zdrojům.

4.4 Cenová politika platformy Firebase

Firebase poskytuje dva plány, „Spark Plan“, který definuje limity pro každou službu, do kterých je daná služba poskytována zdarma. A „Blaze Plan“, který spadá do kategorie „Pay as you go“ [2]. Následující tabulka představuje ceny služeb, popsaných v této práci.

Služba	**Spark Plan**	**Blaze Plan**
Autentizace	10 000 / měsíc	0.06$ / Ověření
Cloud Firestore		
Uložená data	10 GB	0,18$ / GB
Zápisy do dokumentu	20 000 / den	0,18$ / 100 000
Čtení dokumentu	50 000/ den	0,06$ / 100 000
Smazání dokumentu	20 000 / den	0,02$ / 100 000
Real-Time databáze		
Souběžné připojení	100	200 000 / databázi
Uložená data	1 GB	5$ / GB
Stažená data	10 GB / měsíc	1$ / GB
Crashlytics	Zdarma	
Cloud Messaging		
Performance monitoring		
Google Analytics		

Tabulka 1 – Ceny vybraných služeb platformy Firebase [2]

5 Správa uživatelů

Většina moderních aplikací využívá identifikaci uživatelů. Tato funkcionalita umožňuje individuální nastavení preferencí, ukládání dat uživatelů a možnost přizpůsobit aplikaci pro zlepšení uživatelské zkušenosti [3].

V této kapitole si nejdříve představíme problematiku ověřování identity uživatelů a poté se podíváme, jakým způsobem Firebase řeší ověřování a správu uživatelů v rámci projektů.

5.1 Autentizace uživatelů

Obecně jsou řešení pro ověřování uživatelů kompromisem mezi bezpečností a jednoduchostí jejich provedení. Existuje několik metod, které jsou založeny na různých způsobech ověření totožnosti. Nejpoužívanější metody jsou: ověření důkazem znalostí jako jsou hesla, ověření důkazem vlastnictví, například čipové karty a ověření vlastností jako otiskem prstu [4].

V současnosti nejpoužívanějším způsobem ověření je metoda založena na základě znalostí. Její hlavní výhodou je jednoduchost implementace. Nevyžaduje žádný fyzický objekt, pouze abstraktní znalost, kterou lze snadno přenášet a zadávat. Její nevýhodou je však limitovaná bezpečnost, která závisí na uživateli zvoleným heslem. Tento nedostatek lze částečně vyvážit vynucením komplexnosti hesla, například požadavkem, aby heslo obsahovalo čísla, malá i velká písmena a speciální znaky [5].

Metoda založená na důkazu o vlastnictví využívá fyzické objekty takzvané Tokeny. Tento způsob ověření má několik výhod jako jsou schopnost uchovávat a zpracovávat komplexní informace, obtížené kopírování a snadno zjistitelná ztráta. Tyto vlastnosti výrazně zvyšují úroveň zabezpečení při používání tokenů. Mezi nevýhody patří omezená vzájemná kompatibilita, složitost z fyzického hlediska, náklady na výrobu a potenciální poruchovost. Pro jejich využití je také potřeba čtecího zařízení, což zvyšuje náklady při implementaci systému [5].

Ověření pomocí vlastností využívá takzvaných biometrik. Biometrika jsou vyhodnotitelné biologické informace, jejichž proces vyhodnocení je možné automatizovat. Typicky se jedná o otisky prstů nebo rozeznávání obličeje. Na rozdíl

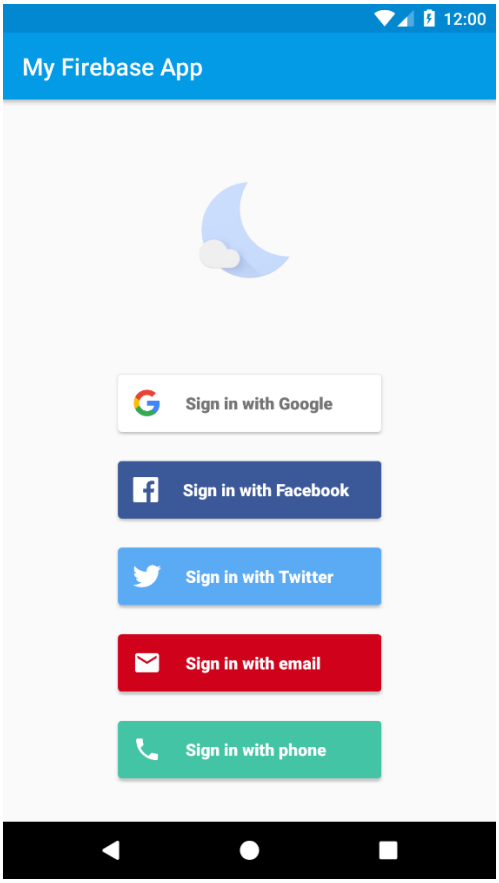
od předchozích metod ověření zde nehrozí ztráta či zapomenutí informace pro přihlášení. Nastává však problém s obtížností měření biometrické informace, což komplikuje implementaci tohoto systému [5].

Všechny uvedené metody mají své výhody a nevýhody, pro jejich vyvážení se tedy často využívá jejich kombinování. Použití dvou či tří ověřovacích metod se označuje jako dvou, respektive tří faktorová autentizace [5].

5.2 Firebase autentizace

Firebase autentizace poskytuje backendové služby, snadno použitelnou sadu vývojových nástrojů (SDK) umožňující vlastní implementaci různých způsobů ověření uživatelů a předpřipravenou knihovnu FirebaseUI Auth poskytující uživatelská rozhraní a kompletní implementaci systému přihlašování uživatelů.

Přihlášení tedy probíhá v několika krocích. Nejdříve je třeba získat přihlašovací údaje od uživatele. Ty mohou být ve formě uživatelova emailu a hesla nebo OAuth tokenu, který je standardem pro využití přihlašovacích údajů poskytnutých službami třetí strany jako jsou Google nebo Facebook. Po úspěšném přihlášení uživatele můžeme řídit jeho pravomoci a přístup k datům. Získaný přihlašovací token lze také využít pro autentizaci ve vlastních backendových službách [2].



Obrázek 2 – UI pro přihlášení podporovanými službami třetí strany [2]

5.2.1 Firebase SDK autentizace

Řešení pomocí autentizačního SDK umožňuje vlastní integraci jednoho nebo více způsobů přihlašování. Výběr z následujících metod je zcela na vývojáři a odvíjí se od požadavků aplikace.

První z podporovaných metod je ověření založené na emailové adrese a hesle. Firebase SDK poskytuje služby pro vytvoření a správu uživatelských účtů. Řeší také případ obnovení hesla zasláním emailu. Pro ověření pomocí služeb třetí strany lze využít účty Google, Facebook, Twitter a GitHub. Kombinace těchto dvou metod je velmi efektivní a u uživatelů oblíbená. Dále je možná implementace ověření pomocí telefonního čísla, kdy uživatel obdrží SMS zprávu s ověřovacím kódem, kterým následně prokáže identitu. Firebase také umožňuje připojení vlastního autentizačního systému pro získání přístupu k dalším službám platformy Firebase. Poslední metodou je Anonymní autentizace. Tento přístup umožňuje uživatelům získat dočasný účet. Tento účet lze později převést na účet normální.

Jak už bylo zmíněno Firebase umožňuje libovolnou kombinaci těchto metod a poskytuje nutné beckendové služby pro jejich správné fungování. Je však nutná vlastní implementace uživatelského prostředí, které uživateli umožní přihlášení. Dále musíme zajistit správnou konfiguraci vyžadovanou službami třetí strany a předání autentizačního OAuth tokenu [2].

5.2.2 FirebaseUI autentizace

Firebase prezentuje FirebaseUI autentizaci jako doporučované kompletní řešení přihlašovacího systému. Toto řešení poskytuje implementaci přihlašovacích metod popsaných v kapitole 4.2.1 a zároveň uživatelské prostředí, které je kompletně přizpůsobitelné vzhledu aplikace.

Komponent FirebaseUI implementuje osvědčené postupy pro autentizaci na mobilních zařízeních a webových stránkách. To může být faktorem při maximalizaci konverze registrace a přihlášení uživatelů k aplikaci. Řeší také krajní situace jako obnovení nebo propojování účtů, které bývají citlivé na bezpečnost a náchylné k chybám [2].

5.2.3 Autentizační tokeny

Firebase autentizace využívá tří základních typů autentizačních tokenů, které slouží k přenosu přihlašovacích údajů uživatele. Prvním typem je Firebase ID token. Tento token je vytvořen ve chvíli přihlášení uživatele do aplikace. Jedná se o JSON web token (JWT), který bezpečně uchovává data s využitím formátu JavaScript Object Notation (JSON) a identifikuje uživatele v projektu Firebase. JWT obsahuje základní informace o uživateli včetně jedinečného ID. Token je odeslán na server, kde slouží k identifikace přihlášeného uživatele.

Druhým typem jsou identifikační tokeny vytvořené třetí stranou. Tyto tokeny mohou mít různé formáty, avšak velmi často se jedná o formát autorizačního protokolu OAuth 2.0. Tyto tokeny jsou požity pro identifikaci u třetí strany a poté převedeny do údajů použitelných službami Firebase.

Posledním typem jsou vlastní Firebase tokeny. Firebase podporuje využití tokenů vytvořených vlastním autentizačním systémem. Tyto tokeny jsou ve formátu JWT a

Firebase je využívá stejným způsobem jako dříve zmíněné tokeny vytvořené třetí stranou [2].

5.3 Uživatelé v rámci projektu Firebase

Po přihlášení uživatele je vytvořen objekt, který reprezentuje jeho účet. Instance uživatelských účtů jsou nezávislé na instancích Firebase autentizace. Může tedy existovat několik referencí různých uživatelů v rámci jednoho kontextu aplikace bez omezení přístupu k jejich metodám [2].

5.3.1 Aktuální uživatel

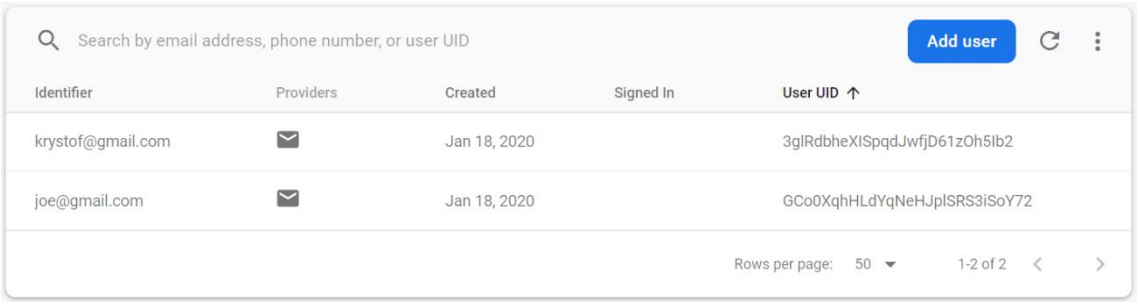
Po přihlášení se uživatel stává aktuálním uživatelem dané autentizační instance. Tato instance uchovává uživatelův stav což znamená, že po restartování aplikace nejsou jeho informace ztraceny. Uchovává také všechny poskytovatele spojené s daným uživatelem a umožňuje tak aktualizaci jeho informací [2].

5.3.2 Atributy uživatele

Uživatelé v rámci Firebase projektu mají čtyři základní předdefinované atributy. Konkrétně se jedná o jedinečné ID, emailovou adresu, jméno a URL odkazující na fotografii uloženou v projektové databázi uživatelů. K objektu uživatele nelze přímo přidávat další atributy. Ty musí být uloženy pomocí jiných služeb, například Google Cloud Firestore.

Na základě metody pro registraci jsou nastaveny dané atributy uživatele. V případě emailové adresy a hesla je vyplněna pouze emailová adresa. Pokud byly využity služby třetí strany, pak jsou využity dané poskytnuté informace. U vlastního autentizačního systému je třeba explicitně přidat poskytnuté informace k danému profilu.

Po vytvoření účtu je možné informace aktualizovat a přidávat na základě změn provedených uživateli na dalších zařízeních [2].



Obrázek 3 – Přehled uživatelů aplikace z webové konzole [2]

5.4 Admin SDK

Pro každý projekt Firebase poskytuje konzoli, která umožňuje provádět různé změny v nastavení, přidávat nebo odebírat služby a spravovat uživatele. Využití této konzole však nemusí být vždy nejvhodnější způsob ke správě uživatelů. Firebase Admin SDK poskytuje programový přístup k daným uživatelům a rozšiřuje možnosti správy poskytované konzolí [2].

5.4.1 Správa uživatelů

Admin SDK poskytuje API pro správu uživatelů v projektu Firebase se zvýšenými oprávněními API umožňuje programově řešit následující úkoly a zabezpečení prostředí serveru: vytvoření nových uživatelů bez jakýchkoliv limitů, vyhledání uživatelů na základě různých kritérii jako jsou ID, email nebo telefonní číslo, přístup k metadatům uživatelských účtů, smazání uživatelů bez použití jejich hesla, aktualizace atributů včetně hesla, ověření nebo změna emailové adresy, vytvoření nových uživatelů s využitím telefonního čísla a vytvoření vlastní uživatelské konzole, která je přizpůsobena danému systému správy uživatelů konkrétní aplikace [2].

5.4.2 Přístup k datům

Firebase poskytuje flexibilní a rozšiřitelná bezpečností pravidla, ta umožňují zabezpečení přístupu k datům uloženým ve službách jako Cloud Firestore, Firebase Real Time Database a Cloud Storage. Pravidla tedy specifikují přístup různých rolí uživatele k datům. Pravidla jsou velmi flexibilní a zcela přizpůsobitelná požadavkům dané aplikace.

Bezpečnostní pravidla využívají rozšiřitelné a flexibilní konfigurační jazyky pro specifikaci přístupu. Konkrétně pro Firebase Realtime Database se pro definování pravidel využívá formátu JSON. Pro definování pravidel ke službám Cloud Storage a Firestore se využívá speciální jazyk, který je navržen tak, aby umožňoval vytvoření komplexnějších struktur pro pravidla.

Pravidla fungují na základě porovnávání vzorů s cestami databáze, poté využijí vlastních podmínek pro umožnění přístupu k datům v těchto cestách. Všechna pravidla ve službách Firebase mají komponent odpovědný za porovnávání cest a vyhodnocení podmínek umožňujících přístup pro čtení a zápis dat. Pro všechny služby Firebase využité v aplikaci musí být pravidla specifikována [2].

5.4.3 Vlastní pravidla pro řízení přístupu k datům

Admin SDK umožňuje přidání vlastních atributů k uživatelským účtům. Při vytvoření vlastních uživatelských požadavků lze využít atributy představující jednotlivé role v rámci aplikace. Ty jsou poté vynuceny bezpečnostními pravidly dané aplikace.

Tento přístup uživatelských rolí lze využít pro obvyklé situace jako jsou přidání administrativních pravomocí přístupu konkrétnímu uživateli, definování různých skupin uživatel a poskytnutí víceúrovňového přístupu jako jsou například moderátor a běžný uživatel nebo platící a neplatící uživatel [2].

6 Databáze

Databáze jsou základní sužbou poskytovanou BaaS. V současnosti se databáze dělí na dva základní typy SQL a NoSQL. Služby BaaS se obvykle implementují typ NoSQL společně s Cloud technologií a Real-Time synchronizací dat. V této kapitole si zmíněné technologie databází představíme a poté se zaměříme na služby Firebase.
Firebase poskytuje dvě služby pro implementaci NoSQL cloud databází. Jedná se o "Firebase real-time database" a "Cloud Firestore" [3]. Tyto služby jsou popsány v podkapitolách 6.3 a 6.4.

6.1 NoSQL Cloud databáze

Cílem NoSQL databází je poskytnutí efektivního způsobu uložení a vyhledávání dat s důrazem na škálovatelnost a dostupnost dat [6].

6.1.1 Relační SQL a Nerelační NoSQL databáze

Relační databáze jsou strukturované. Skládají se z tabulek, které mezi sebou mají vztahy a závislosti. Pro práci s databází a uloženými daty se využívá strukturovaný dotazovací jazyk SQL. Jedná se o poměrně rigidní a standardní způsob ukládání dat s využitím tabulek sloupců a řádků. Sloupce představují atributy a jednotlivé záznamy jsou uloženy v řádcích. Jednotlivé tabulky by měly obsahovat pouze záznamy dané entity, je-li potřeba přidat detaily nebo data, vytvářejí se vztahy mezi tabulkami. Tyto vztahy jsou reprezentovány pomocí takzvaných klíčů, společných atributů v obou tabulkách [6].
Nerelační databáze umožňují větší flexibilitu a přizpůsobitelnost v průběhu navrhování aplikací. Jsou tedy vhodné pokud pracujeme s velkým množství nestrukturovaných dat, pro která by nebylo možné vytvořit relační databázi s jasně definovanými tabulkami a vztahy. Nerelační databáze jsou dokumentově orientované. Namísto tabulek se využívají dokumenty. Ty umožňují uložení nestrukturovaných dat do jednoho dokumentu. Například dokument obsahující detaily zákazníka může zároveň obsahovat jejich objednávky a další informace. Tento přístup může být více intuitivní, ale při nárůstu velikosti dokumentů se zvyšuje nutný výkon pro zpracování [6].

Relační SQL databáze	Nerelační NoSQL databáze
Microsoft SQL Server	MongoDB
Oracle	Oracle NoSQL
MySQL	Apache CouchDB
PostgreSQL	Redis

Tabulka 2 - Přehled populárních SQL a NoSQL databází [7]

6.1.2 Vlastnosti NoSQL databáze

Tři důležité základní vlastnosti NoSQL databází jsou škálovatelnost, flexibilita a replikace dat. Škálovatelnost odkazuje na dosažení vysokého výkonu v distribuovaném prostředí, využitím mnoha univerzálních zařízení. Umožňuje tedy rozložení dat na velké množství zařízení s distribuovaným zatížením při zpracování. NoSQL databáze často umožňují automatickou distribuci dat na zařízení nově přidaná do skupiny.

Flexibilita z hlediska datové struktury znamená, že není nutné definovat schéma pro vytvářenou databázi. To umožňuje uživateli ukládat data do různých struktur v rámci jedné databáze. To však znamená, že většina NoSQL databází nepodporuje využití dotazovacích jazyků.

Replikace dat je proces kopírování dat a jejich distribuce různým systémům za účelem vytvoření redundance a rozložení zátěže. To však vytváří riziko ztráty konzistentnosti dat mezi replikami. Konzistentnost a dostupnost jsou tedy hlavními faktory při hodnocení této vlastnosti [6].

6.1.3 NoSQL Datové modely

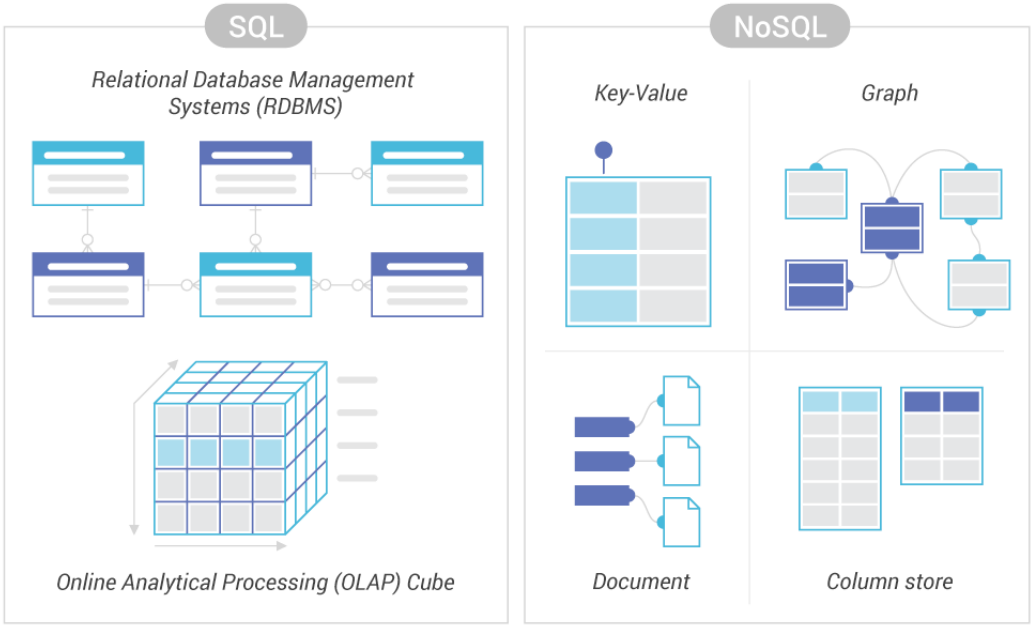
Datové modely popisují formát a strukturu uložených dat a jejich vztahy. Pro NoSQL databáze existuje několik kategorií datových modelů. Při výběru modelu je nutné zvážit požadavky vytvářené aplikace.

Za účelem poskytnutí souběžného přístupu k databázi byl navrhnut datový model „Klíč-hodnota“. Jedná se o jednoduchý ale zároveň velmi výkonný databázový model. Při využití tohoto modelu všechna data zahrnují pár jedinečného klíče a jemu přiřazené hodnoty. Při uložení hodnoty je aplikací vygenerován a přiřazen klíč.

Uložená data mohou mít dynamický soubor atributů. Klíče jsou tedy jediný způsob, jak přistoupit k uloženým hodnotám. Typ vazby hodnoty a klíče závisí na využitém programovacím jazyce. Obvykle se využívají hashovací funkce, kde aplikace hashuje klíč a nalezne umístění dat v databází. To umožní přístup k dané entitě a všem jejím datům.

Druhým modelem je dokumentově orientovaný datový model. Na abstraktní úrovni je podobný předcházejícímu „Klíč-hodnota“ modelu. K uloženým hodnotám můžeme přistoupit pomocí vygenerovaného klíče. Tento model využívá entit nazvaných dokumenty, které jsou vlastně množinou pojmenovaných polí. Každý dokument má tedy přiřazený klíč, který ho identifikuje. Co odlišuje tyto dva modely, je viditelnost dat. Na rozdíl od „Klíč-hodnota“ modelu, v dokumentově orientované databázi není dotazování omezeno pouze na hodnoty klíče. Tento model je tedy vhodné použít, pokud aplikace vyžaduje možnost vyhledávat v databázi na základě konkrétních atributů. Pro vytvoření dokumentů se obvykle využívají standartní formáty jako XML, BSON a JSON.

Pro práci s vysoce provázanými daty se využívá model pro tvorbu Grafových databází. Tyto databáze jsou specializované pro práci s daty obsahujícími mnoho vztahů. V tomto modelu pracujeme s třemi hlavními abstrakty, jedná se o uzly, hrany a atributy. Každý uzel obsahuje informace o entitě. Hrany reprezentují vztahy mezi entitami. Každý vztah má typ a směr s výchozím a konečným bodem (uzlem). Atributy „Klíč-hodnota“ jsou spojeny jak s jednotlivými uzly, tak i se vztahy. Atributy vztahů poskytují další informace o daném vztahu. Směr vztahu udává průchozí cestu z jednoho uzlu na další [6].



Obrázek 4 – Datové modely SQL a NoSQL databází [12]

6.1.4 Transakce NoSQL databází

Databázová transakce je skupina příkazů, převádějící databázi z jednoho konzistentního stavu do druhého. Transakce jsou dalším aspektem, ve kterém se SQL a NoSQL databáze liší. SQL transakce jsou založeny na přesně daných ACID (Atomicita, Konzistence, Izolovanost, Trvalost) vlastnostech. Pro NoSQL jsou však tyto vlastnosti příliš omezující. V roce 2000 Profesor Eric Brewer vymyslel teorém známý jako CAP (Konzistence, Dostupnost, Odolonst vůči přerušení). Teorém říká, že při návrhu databáze je možné dosáhnout jakékoliv dvojice z těchto vlastností v distribuovaném prostředí. Vždy je tedy možné dosáhnout právě dvou z těchto vlastností na úrok třetí. NoSQL transakce lze klasifikovat následovně.

Databáze s ohledem na konzistenci a dostupnost (CA). Tato databáze zajišťuje dostupnost dat a jejich konzistenci, využitím replikace dat. Nestará se však o odolnost vůči přerušení. V případě že mezi dvěma uzly dojde k přerušení, data přestávají být synchronizována.

Databáze s ohledem na konzistenci a odolnost vůči přerušení (CP). V tomto případě databáze zajišťuje konzistenci. Ale nezaručuje přístupnost k uloženým datům. Data jsou uložena v distribuovaných uzlech. Pokud je jeden z uzlů nepřístupný, dochází

ke ztrátě konzistence. Udržuje však odolnost vůči přerušení využitím preventivní synchronizace dat.

Databáze s ohledem na dostupnost a odolnost vůči přerušení (AP). Prioritou této databáze je udržení dostupnosti a odolnosti vůči přerušení. I v případě chyby v komunikaci mezi uzly zůstávají uzly online. Když dojde k vyřešení tohoto přerušení, provede se re-synchronizace dat, avšak nelze zaručit jejich konzistenci.

CAP teorém byl dále rozšířen na PACELC. Tento model bere v potaz odolnost sítě vůči přerušení, přidává tedy odezvu jako zásadní faktor. Později byl představen teorém BASE, jehož záměrem je docílit dostupnost namísto konzistence. Pokud data v systému BASE přestanou být dostupná, systém nepřestává pracovat. V případě selhání uzlu operace pokračuje s replikou dat v jiném uzlu. Pro udržení relevance dat v systému k nim musí někdo přistoupit nebo je aktualizovat. Jakákoliv aktualizace dat může vést ke ztrátě jejich konzistence, avšak systém zaručuje že eventuálně se konzistence dosáhne. Data jsou tedy v budoucnu konzistentní [6].

6.2 Real-time databáze

Real-time databáze jsou databáze hostované v cloudu. Data jsou ukládána ve formátu JSON a nepřetržitě synchronizována s každým přidruženým klientem. Při vytváření multiplatformní aplikace je většina požadavků závislá na jedné instanci, což umožňuje přístup k nejaktuálnějším datům [8].

6.2.1 Real-time model databáze

Každá nová transakce musí projít přes komponent kontroly vstupu. Tento komponent řídí všechny souběžně aktivní transakce v systému a zabraňuje jejich selhání. Jednotlivým transakcím jsou přiřazeny úrovně priority, které zajišťují jejich plánování s ohledem na následující souběžné transakce v systému.

Pro zajištění synchronizace je před uskutečněním transakce nutné provést kontrolu souběžnosti. V případě, že by požadavek transakce na určitých datech byl zamítnut z důvodu jejich nedostupnosti, je tato transakce převedena do fronty čekajících. Když jsou data opět dostupná, daná transakce se aktivuje a provede.

Po dokončení všech operací aktivní transakce jsou výsledky uloženy a všechna využívaná data jsou opět k dispozici ostatním transakcím. Před dokončením

transakce může několikrát dojít k jejímu přerušení a restartovaní. Důvodem k přerušení může být buď ukončující nebo neukončující přerušení. Ukončující přerušení je důsledkem chybějící ukončující lhůty nebo sebe-ukončení, ke kterému může dojít v případě výjimečného stavu. K neukončujícímu přerušení dochází v případě, že nastane deadlock nebo konflikt dat. V tomto případě může dojít k restartování dané transakce [8].

6.2.2 Plánování transakcí

Real-time databáze poskytuje funkce s ohledem na standardní fyzické zdroje, ve kterých jsou data uložena. Transakce přistupující k těmto datům musí být naplánovány s ohledem na real-time výkonnostní cíle. Plánování procesů transakcí v systémech real-time databází se skládá z kontroly souběžnosti a řešení konfliktů.
Protokoly kontroly souběžnosti jsou uzamykání, záznamy času, multiverze a validace. Jejich cílem je vynucení serializovatelnosti. Tyto protokoly je obvykle nutné optimalizovat a jejich kompromisy musí být přehodnoceny v systémech real-time databází.
Zámky v protokolu kontroly souběžnosti jsou využity k synchronizaci operací ve dvoufázovém zamykání. Všechny operace zamykání předchází první operaci odemykání v dané transakci. Dvoufázové zamykání se skládá z fáze rozšiřování, kdy dochází k získání zámků a fáze zmenšování kdy jsou zámky uvolněny. Řešení konfliktů je založeno na využívání priorit jednotlivých transakcí [8].

6.3 Firebase Real-time Database

Firebase real-time database je cloudově hostovaná databáze, umožňující uložení dat ve formátu JSON a poskytující synchronizaci dat v reálném čase pro všechna připojené zařízení. Při využití této databáze všechna zařízení sdílí jednu instanci databáze, což umožňuje synchronizaci dat během milisekund. To platí i v případě využití při vývoji multiplatformní aplikace iOS, Android a JavaScript SDK [2].

6.3.1 Zásadní vlastnosti

Databáze funguje na základě synchronizace dat po každém provedení úprav. Vždy když dojde ke změně uložených dat, všechna připojená zařízení obdrží aktualizaci během několika milisekund.

SDK implementující tuto databázi ukládá data na lokální disk. Tato funkcionalita umožňuje aplikaci fungování při ztrátě připojení, avšak data nejsou aktuální. Po opětovném připojení k internetu se data aktualizují.

Pro přístup není nutné využití aplikačního serveru. Databáze je přístupná přímo z jakéhokoliv klientského zařízení. Zabezpečení je zajištěno pravidly, která jsou použita při čtení či zápisu dat [2].

6.3.2 Datová struktura

Data ve službě real-time databáze jsou uložena jako objekty formátu JSON. Celá databáze má stromovou strukturu. Při přidání dat se vytvoří nový uzel s náležitým identifikačním klíčem. Klíče je možné specifikovat vlastní, například ID uživatele a sémantická jména, nebo jsou automaticky generovány službou.

Firebase představuje následující doporučené postupy při vytváření databázové struktury. Minimalizace zanoření, rozdělení datových struktur do více cest a vytvoření dat s ohledem na jejich škálování.

Firebase Real-time databáze umožňuje zanořování dat až do 32 úrovní, což může vést k vytváření mnohaúrovňových struktur. Tento přístup však není optimální. V případě načtení dat z určitého uzlu se načtou i data všech jeho potomků. Další problém může nastat při přidělování přístupu. Jestliže uživateli umožníme čtení a zápis dat k danému uzlu, jsou tyto pravomoci delegovány i na všechny jeho potomky.

Tyto efekty zanořování dat mohou být nežádoucí, proto se doporučuje minimalizace zanořování.

Předchozí přístup tedy implikuje rozdělení datových struktur do více cest (denormalizace). Takto uložená data lze podle potřeby efektivně načíst na základě specifických žádostí.

Při načítání seznamů databáze je obvykle žádoucí možnost načtení pouze jejich části, obzvláště obsahuje-li seznam velké množství záznamů. V případě statického a jednosměrného vztahu lze tuto situaci řešit vnořením dat. V případě více dynamických záznamů může být nutné využití denormalizace. To lze často provést pomocí specifických dotazů. Avšak v případě, že pracujeme s dvousměrnými vztahy, je nutné využití redundantních informací. V tomto přístupu vytváříme indexy spojující jednotlivé záznamy. Příkladem mohou být záznamy uživatelů a skupin do kterých patří. Uživatelé mohou patřit do několika skupin a skupiny obsahují více uživatel. Takové indexování umožňuje efektivní vyhledání informací o skupinách a jednotlivých uživatelích [2].

6.3.3 Čtení a zápis dat

Pro zápis dat se využívá reference na instanci databáze spojené s projektem. Pro čtení dat se k dané referenci musí přidat asynchronní posluchač událostí. Tento posluchač je nezávislý na běhu aplikace a je aktivován při jakékoliv změně dat v databázi. Ve výchozím nastavení mají přístup k těmto operacím pouze ověření uživatelé.

Pro základní zapisovací operace je k dispozici metoda „setValue“, která je součástí reference databáze. Tato metoda umožňuje uložení dat a jejich přepsání pro specifikovanou cestu. Umožňuje použití datových typů, které jsou k dispozici ve formátu JSON. Konkrétně se jedná o String, Long, Double, Boolean, Map a List. Lze také uložit vlastní objekty.

V případě ztráty připojení aplikace nepřestává fungovat. Každý klient si udržuje lokální kopii dat. Veškeré zápisy dat provedené v této situaci jsou uloženy na lokální verzi. Firebase klient poté synchronizuje data s databázovým serverem.

Pro čtení dat a poslouchání událostí jsou k dispozici metody „addValueEventListener“ nebo „addListenerForSingleEvent“, které přidávají

posluchač událostí k referenci databáze. A metoda „onDataChange“ pro čtení dat na specifikované cestě. Tato metoda je spuštěna vždy když dojde ke změně dat a navrací všechna data na dané cestě včetně jejich potomků. V některých případech chceme čekat pouze na první událost, například pokud inicializujeme UI element, který se nebude měnit. Pro tuto situaci existuje metoda „addListenerForSingleValueEvent“, která je spuštěna pouze jednou [2].

6.4 Cloud Firestore

Cloud Firestore je služba poskytující dokumentově orientovanou NoSQL databázi. Databáze umožňuje ukládání a synchronizaci dat pro vývoj na straně serveru a klienta z platformy Firebase a Google Cloud [2].

6.4.1 Zásadní vlastnosti

Datový model databáze podporuje flexibilní, hierarchickou strukturu dat. Data jsou ukládána v dokumentech, které jsou organizovány v kolekcích. Dokumenty mohou obsahovat komplexní objekty a sub-kolekce.

Firestore umožňuje využití dotazů k načtení jednotlivých dokumentů nebo kolekcí. Dotazy mohou zahrnovat několik filtrů a řazení dat. Využívá také automatické indexování, což výrazně urychluje prohledávání databáze.

Podobně jako real-time databáze, Cloud Firestore využívá synchronizace dat mezi připojenými zařízeními. Databáze je však efektivně navržena i pro jednodušší, jednorázové dotazy čtení. Poskytuje také podporu při odpojení zařízení od internetu. Všechny změny jsou uloženy lokálně a později synchronizovány se serverem.

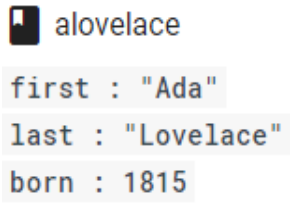
Cloud Firestore byl navržen s ohledem na škálovatelnost dat a se schopností pokrýt vysoké zatížení největších aplikací na trhu [2].

6.4.2 Datový model

Cloud Firestore je dokumentově orientovaná databáze. Data jsou tedy uložena do jednotlivých dokumentů ve formátu JSON. Dokumenty jsou pak uspořádány do kolekcí. Firestore je optimalizován pro ukládání velkého množství menších dokumentů. Všechny dokumenty musí být uloženy do kolekcí. Zároveň mohou

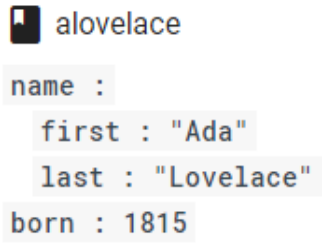
obsahovat sub-kolekce a vnořené objekty. Kolekce a dokumenty jsou vytvářeny implicitně při ukládání dat [2].

Každý dokument je označen jménem a obsahuje pole mapovaná k hodnotám.



Obrázek 5 – Struktura dokumentu [2]

Dokumenty mohou obsahovat komplexní, vnořené objekty nazývané mapy.



Obrázek 6 – Struktura dokumentu obsahujícího objekt [2]

Dokumenty jsou logicky seskupeny do kolekcí. Ty vždy obsahují pouze dokumenty. Nemohou obsahovat samotná pole nebo další kolekce. Názvy dokumentů jsou v rámci kolekce jedinečné. Vytváření a mazání kolekcí probíhá automaticky s přidáním prvního a odstraněním posledního dokumentu.

Obrázek 7 – Struktura kolekce [2]

Firestore je velmi flexibilní v ohledu na strukturu, obsah dokumentů a datové typy. V rámci jedné kolekce mohou dokumenty obsahovat různá pole a datové typy. Avšak doporučeným přístupem je pro podobné dokumenty zachovat konzistentní struktury, což usnadní vyhledávání [2].

6.4.3 Datové typy

Cloud Firestore podporuje využití jak základních datových typů využívaných ve formátu JSON, tak i komplexnějších typů. Následující tabulka představuje jednotlivé datové typy. A popisuje způsob řazení a porovnávání hodnot stejného typu.

Datový typ	Způsob řazení hodnot
Array	Hodnota elementu
Boolean	False < True
Byte	Pořadí byte
Datum a čas	Chronologicky
Float	Číselné
Geografická Poloha	Zeměpisná Šířka poté Výška
Integer	Číselné
Mapa	Klíče poté hodnoty
Null	-
Reference	Podle pole cesty (ID)
String	UTF-8 byte

Tabulka 3 – Datové typy služby Cloud Firestore databáze [2]

Při řazení hodnot pole, které může obsahovat různé datové typy, Firestore používá deterministické řazení. Řazení je založeno na vnitřní reprezentaci [2].

1.	Null
2.	Boolean
3.	Integer a Float, seřazeny podle velikosti
4.	Datum
5.	String
6.	Byte
7.	Reference Cloud Firestore
8.	Geografická poloha
9.	Array
10.	Mapy

Tabulka 4 – Řazení datových typů ve službě Cloud Firestore [2]

6.4.4 Indexy

Databázové indexy mapují jednotlivé položky databáze k jejich umístění v databázi. Při provedení dotazu může databáze pomocí indexu rychle vyhledat umístění požadovaných položek. Cloud Firestore využívá dvou typu indexů, index jednoho pole a složený index.

Pokud pro dotaz neexistuje žádný index, databáze obvykle postupně prohledávají jejich obsah. Cloud Firestore však garantuje indexy pro všechny dotazy. Výsledkem je rychlé vyřizování jednotlivých dotazů závislé pouze ne velikosti hledaných dat. Většina základních indexů je automaticky vytvořena, při využívání aplikace pomáhá Firestore identifikovat a vytvořit dodatečné indexy.

Index jednoho pole uchovává seřazené mapování všech dokumentů kolekce, které obsahují specifikované pole. Každá položka v záznamech Indexu jednoho pole, zahrnuje umístění dokumentu v databázi a jeho hodnotu pro dané pole.

Složený index uchovává seřazené mapování všech dokumentů kolekce, které obsahují danou množinu polí. Vzhledem k velkému počtu různých kombinací nejsou tyto indexy vytvářeny automaticky. Namísto toho Cloud Firestore pomáhá identifikovat vhodné složené indexy pro navrhovanou databázi [2].

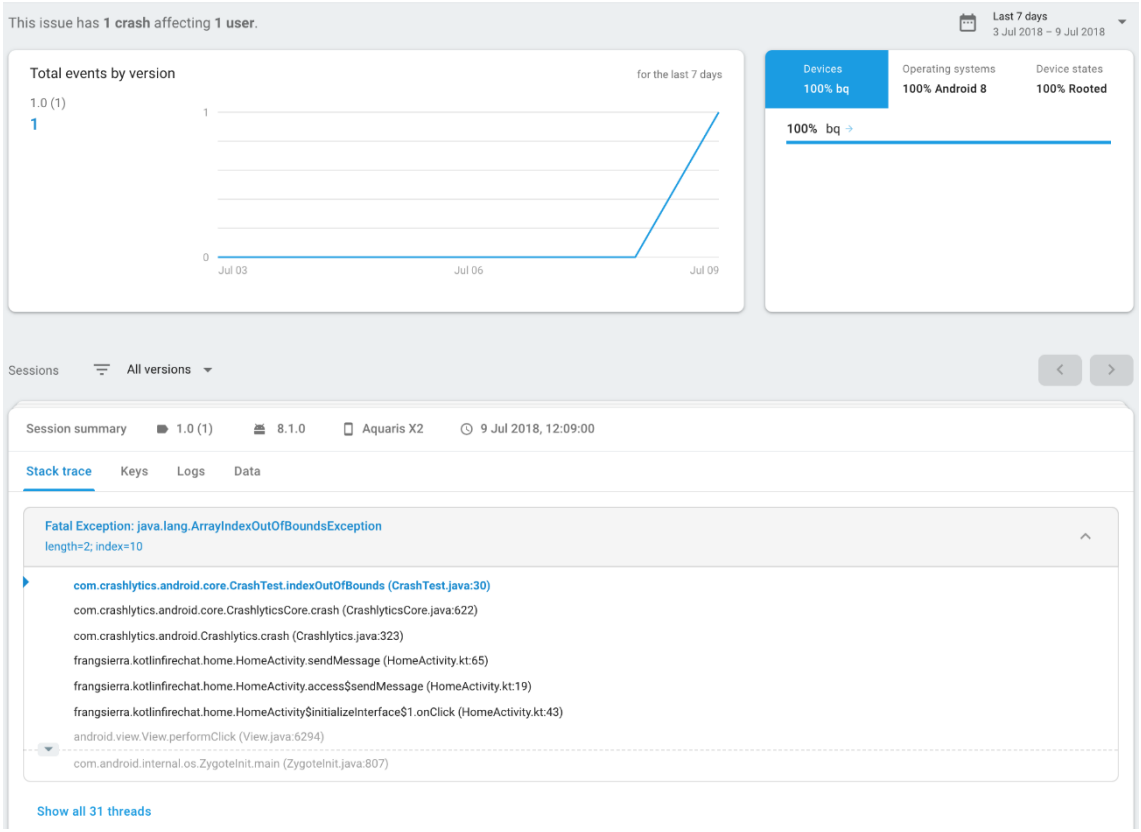
7 Služby pro optimalizaci, rozšíření

V této kapitole si představíme čtyři vybrané služby, které slouží pro optimalizaci a rozšíření aplikací. Konkrétně se jedná o „Crashlytics“ a „Performance Monitoring“, které jsou určeny pro nalezení chyb a zlepšení výkonu aplikace, „Firebase Cloud Messaging“ umožňující zaslání zpráv a upozornění uživatelům, a nakonec „Google Analytics“, která poskytuje zásadní statistiky a informace o aplikaci a jejím využití.

7.1 Crashlytics

Firebase Craslytics je podpůrná služba poskytující přehled o chybách nastávajících v reálném čase. Umožňuje sledovat, upřednostňovat a opravovat chyby v aplikaci. To je docíleno jejich chytrým seskupováním a zdůrazněním okolností, které k nim vedou [2].

V případě, že nastane chzba Crashlytics sbírá z uživatelských zařízení informace jako jsou charakteristiky zařízení, úroveň nabití baterie, rotace obrazovky, připojení k síti a mnoho dalších. Zjišťuje, zda daná chyba nastává u více zařízení, poskytuje upozornění, pokud se chyba začne objevovat častěji a může nalézt část kódu, který je její příčinou [10].



Obrázek 8 – Chyba zachycena v Crashlytics konzoli [10]

Služba Crashlytics se aktivuje ve webové konzoli, kde také najdeme přehled o nalezených chybách. Zde najdeme čtyři záložky:

1. „Stack trace“ zobrazuje trasování chyby v kódu aplikace.
2. „Keys“ což jsou vlastní atributy přidané k trasování vývojářem.
3. „Logs“ logování aplikace, jako jsou všechny akce provedené uživatelem před nastáním dané chyby.
4. „Data“ jsou obecné informace o zařízení a operačním systému.

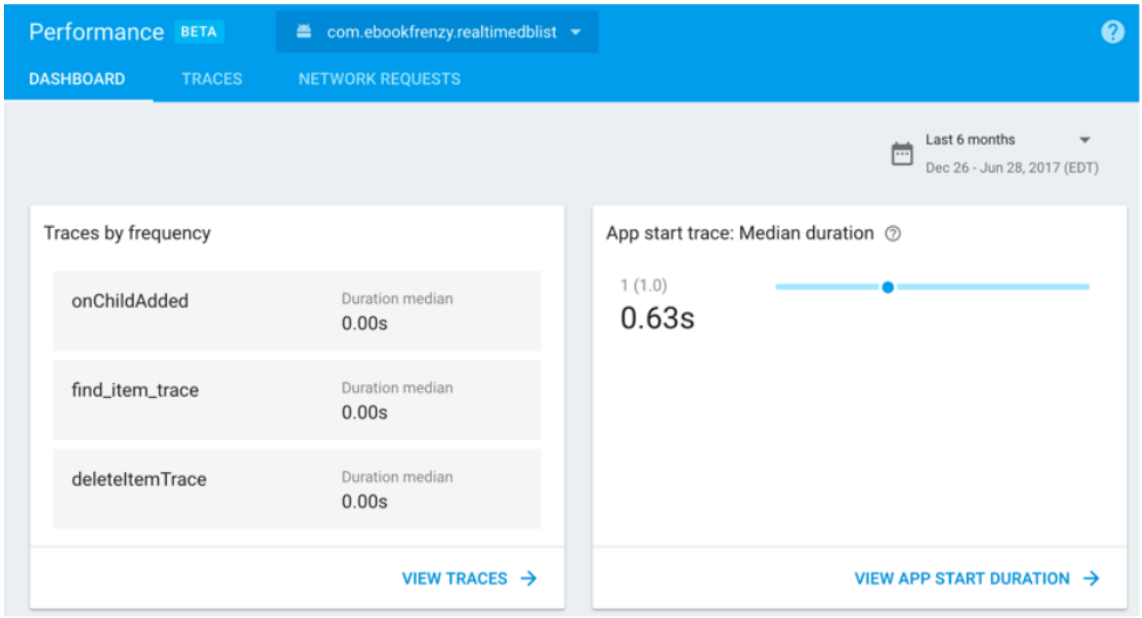
[10]

7.2 Performance monitoring

Firebase Performance Monitoring je služba poskytující monitorování a přehled charakteristik výkonu aplikace. Služba je založena na trasování. To je zachycení dat o výkonu, ke kterým dochází mezi dvěma body v rámci prováděcího cyklu aplikace. Služba také poskytuje vytvoření vlastního trasování pomocí implementace v kódu aplikace. To umožňuje lepší řízení monitorování a získání přehledu o specifických částech funkcionalit aplikací. Služba poté poskytuje data jako verze aplikace, země, operační systém, rádiové a přenosové informace. Pro HTTP požadavky, čas odpovědi, objem dat a úspěšnost [11].
Po připojení služby k aplikaci pomocí SDK se automaticky monitorují tyto trasy:

1. Spuštění aplikace poskytuje informace o výkonu, při spuštění aplikace až do zavolání metody onResume() v aktivitě spouštěče.
2. Aplikace na pozadí sleduje výkon aplikace při jejím běhu na pozadí. Začíná voláním metody onStop(), poslední aktivity po přechodu na pozadí, a končí metodou onResume(), první aktivitou po přechodu na popředí.
3. Aplikace na popředí sleduje výkon běhu aplikace na popředí. Tedy od volání onResume(), první aktivity na popředí, až do metody onStop()
4. HTTP Síťové požadavky sleduje výkon http požadavků od jejich vytvoření až do přijetí výsledků.

Výsledky těchto sledování jsou dostupné ve webové konzoli Firebase.



Obrázek 9 – Webová konzole Performance Monitoring [11]

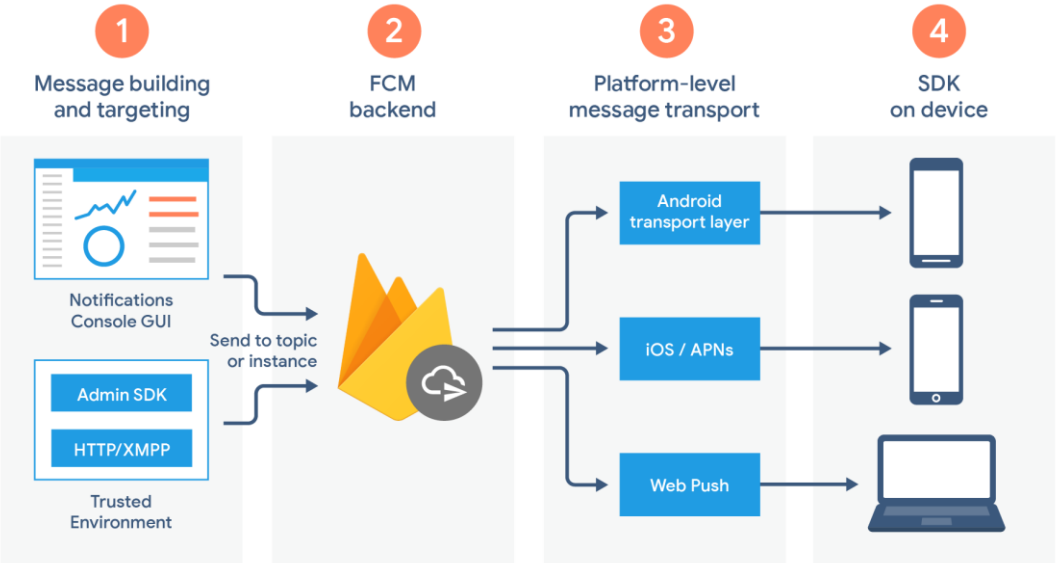
7.3 Firebase Cloud Messaging

Služba Firebase Cloud Messaging (FCM) umožňuje zasílání zpráv a upozornění všem zařízením připojeným k aplikaci. Lze tak upozornit klienta na možnost synchronizace nových dat nebo emailu. Slouží také jako prostředek k udržení uživatelů. Zprávy je možné zasílat jednotlivým zařízením nebo specifikované skupině zařízení [2].

FCM závisí na skupině komponent, které mají na starost vytvoření, transport a přijetí zpráv:

1. Pro vytvoření upozornění a zpráv je možné využít „Notification composer“, který poskytuje grafické rozhraní pro vytváření požadavků zpráv. Pro automatizaci a možnost využití všech typů zpráv je nutné zprávy vytvořit v serverovém prostředí, které podporuje Firebase Admin SDK nebo FCM serverové protokoly. Takové prostředí je například Cloud Funkce pro Firebase, Google App Engine nebo vlastní aplikační server.

2. Druhou částí je FCM backend, který přijímá požadavky, provádí rozřazení zpráv dle témat, a generuje metadata jako je například ID.
3. Další částí je transportní vrstva na úrovni platformy, která směruje správy k cílovým zařízením, zajišťuje jejich doručení a aplikuje potřebnou konfiguraci.
4. Poslední částí jsou uživatelská zařízení, kde je zpráva nebo upozornění zobrazeno a zpracováno dle stavu a aplikační logiky aplikace.



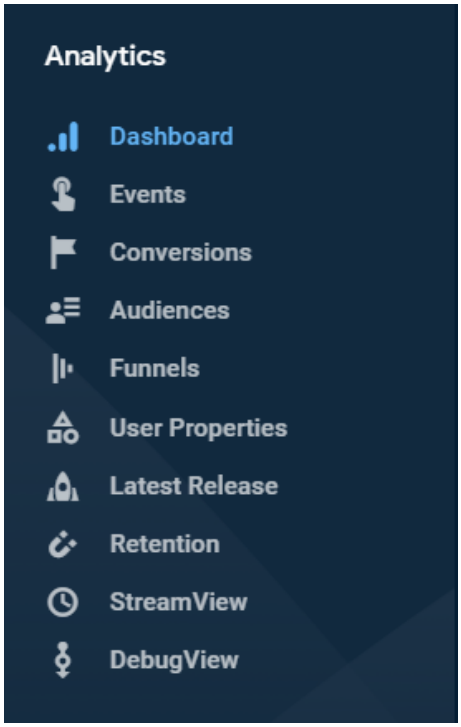
Obrázek 10 – Diagram komponent architektury FCM [2]

7.4 Google Analytics pro Firebase

Služba Google Analytics funguje na principu sledování událostí v aplikaci. Informace analyzuje, a tak poskytuje statistický přehled o interakci uživatele s aplikací. Ve službě je definována množina událostí, které jsou sledované automaticky po připojení služby k aplikaci. Poskytuje však i API, které umožňuje definování a sledování vlastních událostí. V rámci služby je možné vytvoření skupin uživatelů, které mohou být využity pro vytvoření specifické funkcionality. Například pokud definujeme skupiny na základě určité lokace, můžeme pak dané skupině poslat zprávu pomocí FCM. Google Analytics lze integrovat s dalšími službami Firebase jako jsou Firebase Cloud Messaging nebo Crashlytics [3].

Jak už bylo zmíněno, služba automaticky definuje množinu výchozích událostí, o kterých sbírá data. Mezi tyto události patří například, „first_open“ – první otevření po instalaci aplikace, „in_app_purchase“ – provedení platby v rámci aplikace, „user_engagement“ – tato událost je odesílána periodicky pokud je aplikace na popředí zařízení, „app_update“ – informuje, zda uživatel aktualizuje aplikaci, „app_exception“ – nastává, pokud dojde k chybě v aplikaci. Kromě událostí sbírá také informace o uživatelích a jejich zařízení, například věk, verze aplikace, země původu, značka zařízení a model, jazyk a další [3].

Statistiky jsou prezentovány ve webové konzoli Firebase Analytics. Zde je několik kategorií. „Dashboard“ poskytuje základní informace jako aktivní uživatelé, lokace uživatelů, stabilitu aplikace a další.



Obrázek 11 – Dostupné kategorie v Google Analytics konzoli [2]

8 Vývoj mobilní aplikace s využitím služeb Firebase

Tato část bakalářské práce popisuje postup vývoje mobilní aplikace s využitím vybraných služeb, které byly popsány v části teoretické. Měla by poskytnout přehled o možnostech využití služeb a demonstrovat jejich implementaci při vývoji aplikace.

8.1 Základní požadavky na aplikaci

Mobilní aplikací bude jednoduchá sociální síť. Její funkcionalita umožňuje využití klíčových služeb platformy Firebase.

Základní funkcí takové aplikace je správa uživatelských účtů. Musí poskytovat možnost vytvoření účtu, bezpečné zálohování uživatelských dat a autentizaci a autorizaci v rámci aplikace. Pro tyto účely bude implementována služba „Firebase Autentizace“, kterou se teoreticky zabývala kapitola 5.

V kapitole 6. byly rozebrány dvě služby poskytující cloud databáze. Pro účely této aplikace jsem se rozhodl využít službu Firestore. Ta nám umožní implementovat NoSQL databázi a poskytne real-time synchronizaci dat vygenerovaných uživateli. Konkrétně umožní ukládání aktuální geografické polohy klientských zařízení a funkcionalitu real-time chatu mezi uživateli. Dále zde také budeme ukládat informace o profilech, které budou využitz při vyhledávání uživatelů.

Poslední službou využitou při vývoji bude Cloud Messaging, která byla popsána v kapitole 7.3. Ta bude sloužit pro zasílání upozornění o nových zprávách uživatelům.

8.2 Inicializace a připojení k platformě Firebase

Pro správu služeb platformy je nutné využít webové konzole Firebase. Tato konzole umožňuje vytvoření projektu formou dialogu. Po vytvoření nového projektu získáme přístup ke službám platformy.

V rámci jednoho projektu je možné připojit více aplikací z různých platforem (Android, iOS, Web atd.), které potom sdílí databázi a další služby. Lze tak vytvořit multiplatformní projekt.

Počet vytvořených projektů je ve výchozím nastavení omezen na deset, ale o navýšení toho limit je možné zažádat. Využívané zdroje jsou sdíleny pouze v rámci jednoho projektu. Cenové plány popsané v kapitole 4.4 se vztahují k individuálním projektům. V každém nově založeném projektu je tedy vždy možnost limitovaného využití všech služeb zdarma.

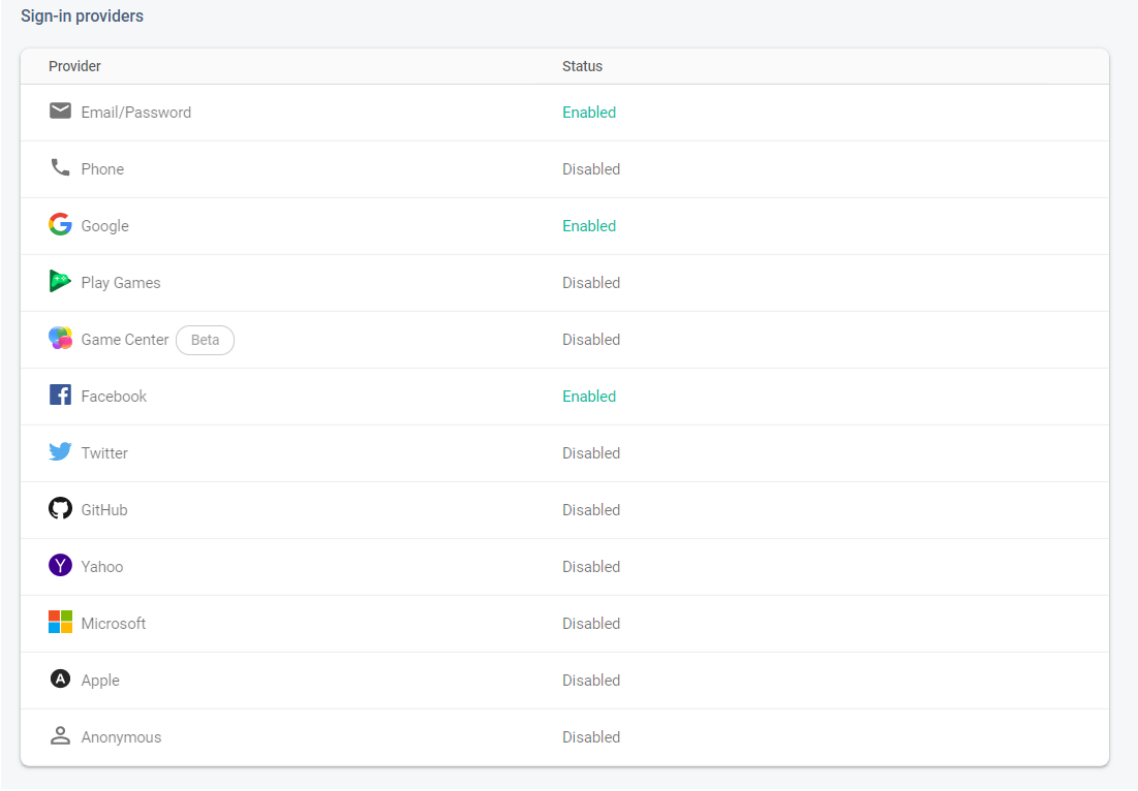
Po založení projektu a připojení aplikace umožňuje konzole aktivaci libovolných služeb platformy. Každá služba vyžaduje určité kroky pro aktivaci (specifikace závislostí a další nastavení). Těmito kroky provede uživatele dialog nebo jsou popsány v oficiální dokumentaci. Konzole pak umožňuje různá nastavení jednotlivých služeb a poskytuje užitečné přehledy, například využití nebo statistiky. Konkrétní postup aktivace, možnosti nastavení a přehledy představím v kapitolách implementace jednotlivých služeb.

8.3 Autentizace

Pro účely aplikace je nezbytné umožnění registrace a přihlášení uživatele do aplikace. Pro řešení je nutné využít službu Firebase Autentizace. Tato služba poskytuje registraci uživatelů, bezpečné uložení jejich informací, ověřování přístupu a správu účtů.

8.3.1 Nastavení služby ve webové konzoli

Službu je nejdříve nutné aktivovat ve webové konzoli. Prvním krokem po aktivaci je nastavení metod registrace a přihlášení, které bude aplikace využívat. V konzoli je toto nastavení dostupné v oddíle Authentication > Sign-in method. Pro účely aplikace byly vybrány metody Email/Password, Google Account a Facebook Account.

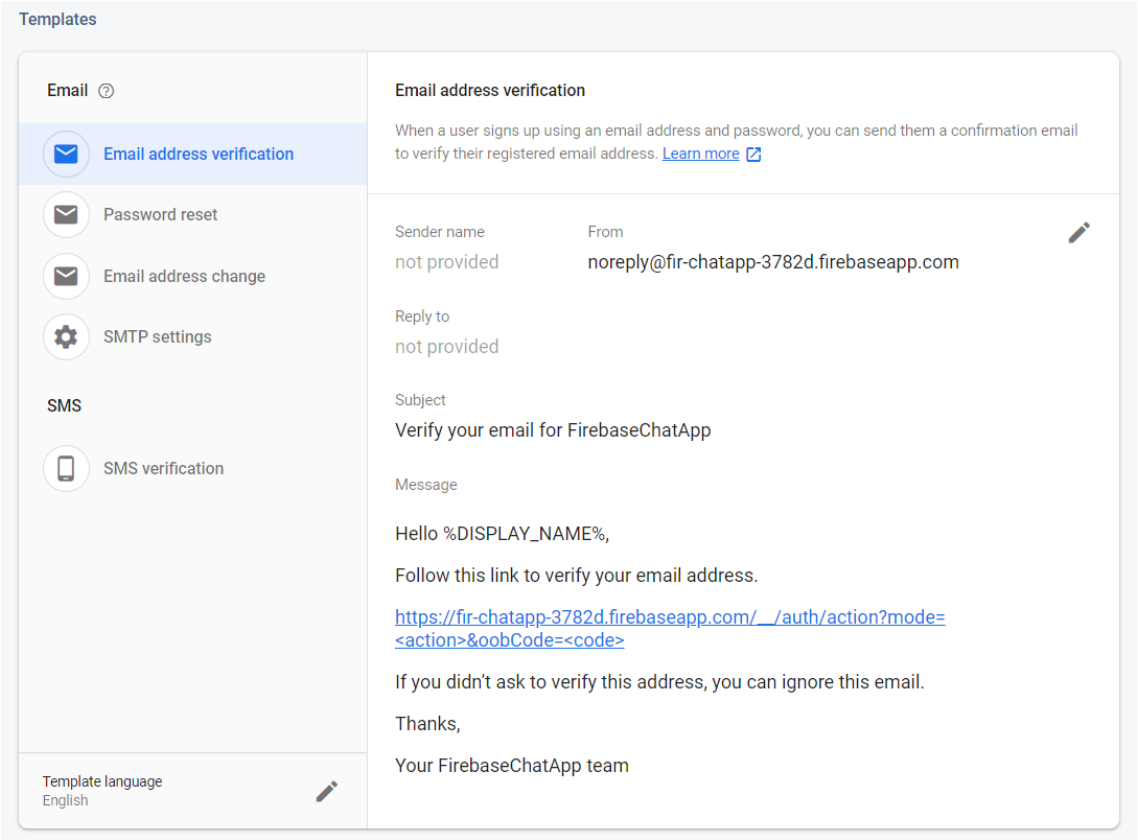


Obrázek 12 – Nastavení sign-up metod ve webové konzoli

Metoda Email/Password umožňuje vytvoření účtu pomocí libovolné e-mailové adresy. Je tedy vhodné zajistit způsob pro ověření přes e-mailový server. Server lze specifikovat vlastní nebo je možné využít servery Firebase.

V nastavení Authentication > Templates se nacházejí šablony pro ověření adresy, změnu adresy a reset hesla. Zde specifikujeme informace a zprávu, která bude odeslána při ověřování.

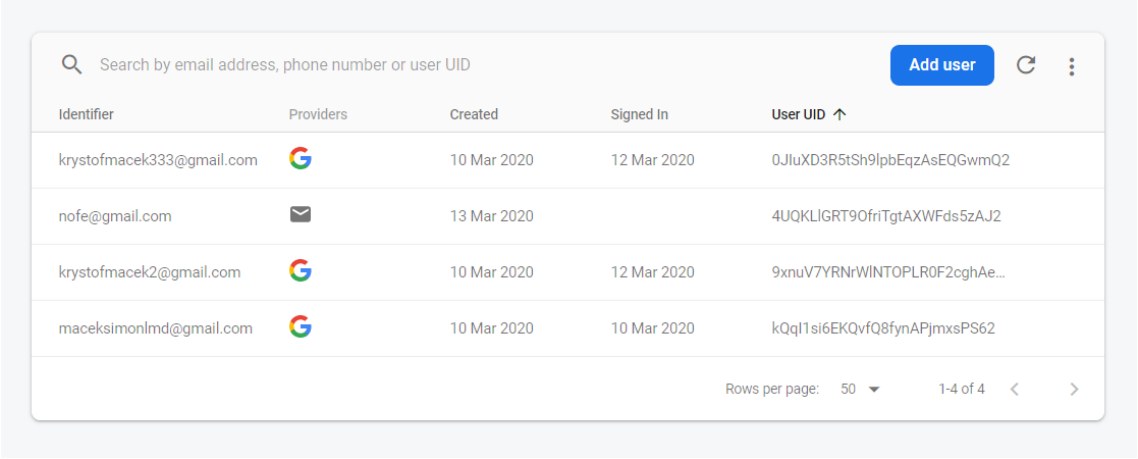
Metoda využívající Google a Facebook účty vyžaduje specifikaci klíče „SHA1 fingerprint“ v nastavení aplikace. Tento klíč je využíván jako identifikátor aplikace pro přihlašování. Při vývoji pro android ho lze vygenerovat přímo v Android Studiu, v jiném případě ho lze vygenerovat v příkazové řádce ze souboru keystore.



Obrázek 13 – Nastavení emailových šablon ve webové konzoli

Po úspěšné registraci je uživatel uložen do databáze. V kategorii Authentication > Users se zobrazuje přehled registrovaných uživatelů a vybraných informací. Konkrétně emailová adresa, způsob registrace, datum vytvoření, poslední přihlášení a jedinečná identifikace.

Účty je ze možné manuálně spravovat, můžeme přidávat nové, mazat, deaktivovat nebo resetovat jejich heslo. Přístupné jsou zde také parametry hashovací funkce, která je použita pro zabezpečení hesel uživatel.



Obrázek 14 – Přehled registrovaných uživatel ve webové konzoli

8.3.2 Připojení služby k aplikaci

Po dokončení nastavení v konzoli je možné službu implementovat ve zdrojovém kódu aplikace. Abychom mohli službu využít musíme do projektu přidat následující závislosti:

implementation 'com.firebaseui:firebase-ui-auth:6.2.0'

implementation 'com.google.firebase:firebase-auth:19.3.0'

8.3.3 Řízení spuštění

Pro mobilní aplikace obvykle platí, že po přihlášení uživatele zůstává daný uživatel přihlášen, dokud se manuálně neodhlásí. FirebaseAuth podporuje toto nastavení ve výchozím stavu.

Pro řízení této funkcionality je nutné vytvořit aktivitu, která se spustí vždy jako první a kontroluje, zda je uživatel přihlášen, kód této třídy je vidět na následujícím obrázku. S Firebase Autentizací se v kódu pracuje s balíčkem *com.google.firebase.auth.**.

```
1. @Override protected void onCreate(Bundle savedInstanceState) {
2.     super.onCreate(savedInstanceState);
3.     setContentView(R.layout.activity_splash_screen);
4.     FirebaseAuth auth = FirebaseAuth.getInstance();
5.     FirebaseUser user = auth.getCurrentUser();
6.     Intent intent;
7.
8.     if(user == null) {
9.         intent = new Intent(
10.            getApplicationContext(),
11.            SignupActivity.class
12.        );
13.
14.        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
15.        startActivity(intent);
16.        finish();
17.    } else {
18.        intent = new Intent(
19.            getApplicationContext(),
20.            MainActivity.class
21.        );
22.
23.        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
24.        startActivity(intent);
25.        finish();
26.    }
27. }
```

Zdrojový kód 1 - SignupActivity.java, kontrola přihlášení

Prvním krokem je získání instance autentizace pro toto zařízení. K této instanci se dostaneme přes třídu *FirebaseAuth*, která metodou *„getInstance()“* vrací objekt autentizace (ř. 4). Instanci je možné využít pro získání aktuálně přihlášeného uživatele (ř. 5). V případě, že je uživatel přihlášen, vrací metoda *„getCurrentUser()“* objekt, který ho reprezentuje. Tento objekt umožňuje přístup datům daného účtu jako ID, email, jméno, telefonní číslo a další. Pokud uživatel přihlášen nebyl, vrací metoda hodnotu *null.* Pokud tedy dostáváme objekt uživatele s hodnotou *null* (ř. 8), spustí se aktivita pro přihlášení (ř. 9-16), v jiném případě se spouští hlavní aktivita aplikace (ř. 18-25).

8.3.4 Registrace a přihlášení

Nyní se podíváme na SignUp aktivitu sloužící pro registraci a přihlášení do aplikace. Pro přihlašování uživatelů využijeme Firebase authUI. Tento balíček poskytuje třídy pro implementaci SignUp dialogu, který je velmi typický pro mobilní aplikace. Uživatelé jsou vždy identifikování e-mailovou adresou. Při jejím vyplnění Firebase zkontroluje, zda již existuje účet s touto adresou. Pokud neexistuje umožní jeho vytvoření. Pokud již existuje, bude uživatel vyzván k přihlášení.

Druhou možností bylo vytvoření vlastního uživatelského rozhraní pro přihlašování. Tato metoda podporuje registraci pomocí emailové adresy a hesla, a následné využití Firebase SDK pro vytvoření uživatel. Avšak v případě využívání poskytovatelů třetí strany (Facebook, Google atd.), je nutné autentizaci implementovat právě přes authUI. Nezbytné třídy pro práci s Firebase authUI získáme z balíčku *com.firebase.ui.auth.*.*

Aktivitu SignUp lze rozdělit na dvě částí, první část je implementována metodou *„onCreate()“*. V této metodě probíhá kompletní nastavení procesu SignUp. K tomu slouží metoda *„startActivityForResult()“*. Druhá část implementuje metodu *„onActivityResult()“*, která zajišťuje řízení výsledku metody *„startActivityForResult()“* a procesu SignUp.

```
1. AuthUI mAuthUI = AuthUI.getInstance();
2. startActivityForResult(
3.    mAuthUI.createSignInIntentBuilder()
4.        .setAvailableProviders(
5.              Arrays.asList(
6.
7.                new AuthUI.IdpConfig.EmailBuilder()
8.                        .setRequireName(false)
9.                        .build(),
10.
11.               new AuthUI.IdpConfig.GoogleBuilder()
12.                        .setSignInOptions(
13.                 new GoogleSignInOptions
14.                    .Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
15.                    .requestIdToken(
16.                       getString(R.string.default_web_client_id))
17.                          .requestEmail().build()
18.                    ).build(),
19.
20.               new AuthUI.IdpConfig.FacebookBuilder()
21.                          .build()
22.                 ))
23.               .setIsSmartLockEnabled(false)
24.               .setLogo(R.drawable.splash_image)
25.               .build(), RC_SIGN_IN);
```

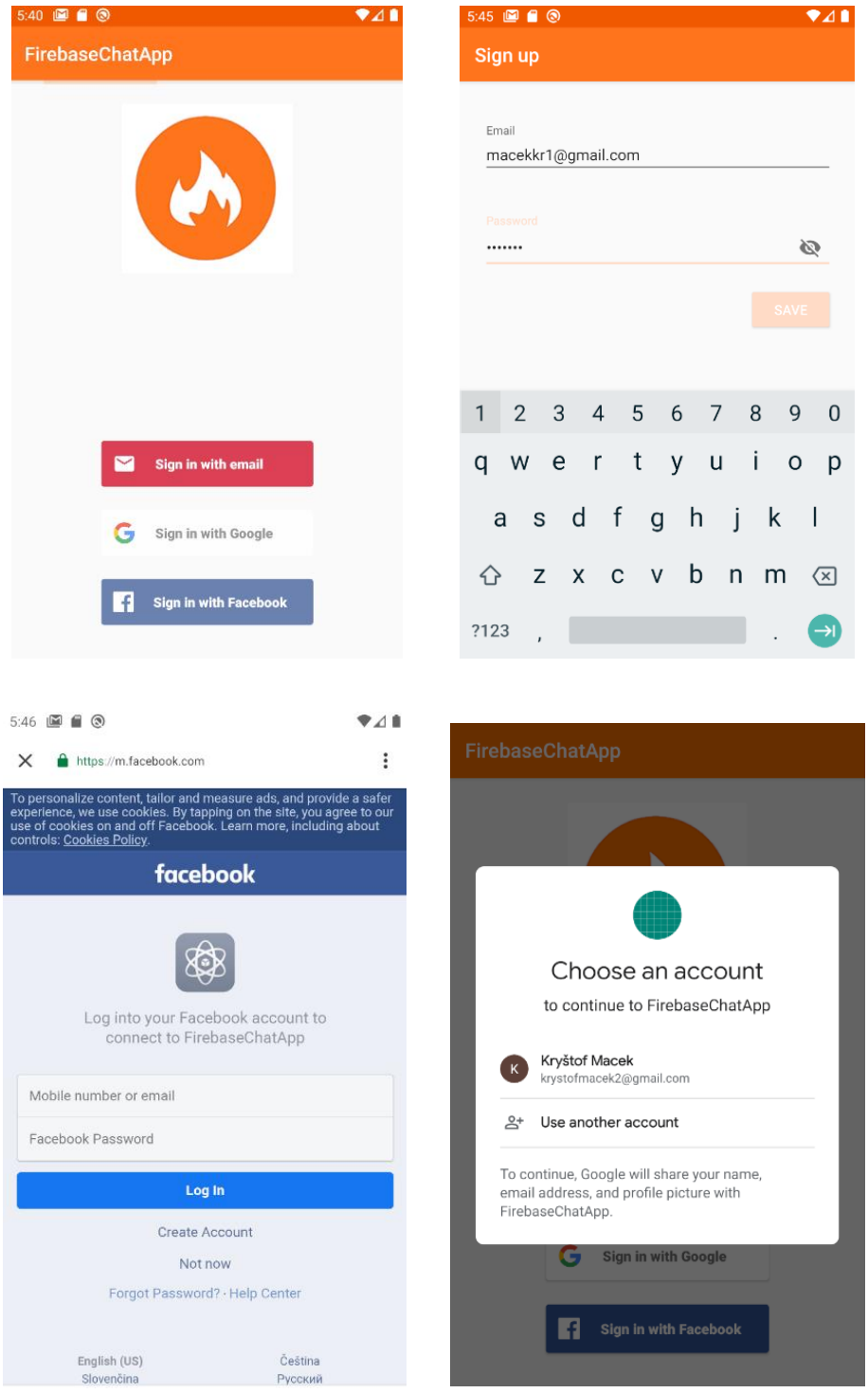
Zdrojový kód 2 - SignupActivity.java, poskytovatelé identity

Pro implementaci procesu SignUp potřebujeme získat instanci třídy *AuthUI* (ř.1). Samotné nastavení poté probíhá v metodě *„startActivityForResult()“* (ř. 2), která vyžaduje dva argumenty. Prvním argumentem je instance objektu Intent. Ten se využívá pro přechod mezi aktivitami.

Pomocí objektu *AuthUI* můžeme požadovaný *Intent* vytvořit (ř. 3-4). Po jeho vytvoření musíme specifikovat nastavení možností SignUp. Toto nastavení se provádí metodou *„setAvailableProviders()*“ (ř. 4). Metoda vyžaduje jeden parametr, kterým je seznam poskytovatelů identity (ř.5). Přes třídu AuthUI získáváme přístup k objektu IdpConfig (ř. 7, 11, 20), který reprezentuje konfiguraci poskytovatele identity. Tento objekt poskytuje různá nastavení pro přihlašování, například zda vyžadovat jméno (ř. 8) nebo emailovou adresu (ř. 17). Dále AuthUI umožňuje specifikovat dodatečná nastavení (ř.23-24), jako například logo nebo aktivace smart-lock.

Nakonec přidáme druhý argument metody *„startActivityForResult()“*, který specifikuje kód požadavku *RC_SIGN_IN* (ř.25), ten je využit v následující části pro řízení výsledku.

Provedením takového nastavení získáváme uživatelské rozhraní a dialog pro proces SignUp.



Obrázek 15 – UI SignUp Procesu

Druhou fází je řízení výsledku metody *„startActivityForResult()“* a procesu SignUp, které probíhá v metodě *„onActivityResult()*“.

```
1. @Override protected void onActivityResult(
2.     int requestCode,
3.     int resultCode,
4.     @Nullable Intent data) {
5.         super.onActivityResult(requestCode, resultCode, data);
6.         if (requestCode == RC_SIGN_IN) {
7.             IdpResponse response =
8.                 IdpResponse.fromResultIntent(data);
9.
10.            if (resultCode == RESULT_OK) {
11.                startActivity(
12.                    new Intent(
13.                    SignupActivity.this,
14.                    MainActivity.class)
15.                );
16.            finish();
17.        } else {
18.             if (response == null) {
19.                 Toast.makeText(
20.                     this,
21.                     "Sign in cancelled.",
22.                     Toast.LENGTH_SHORT).show();
23.                 return;
24.             }
25.             if (response
26.            .getError().getErrorCode( ) == ErrorCodes.NO_NETWORK){
27.                 Toast.makeText(
28.                     this,
29.                     "No internet connection.",
30.                     Toast.LENGTH_SHORT).show();
31.             }
32.         }
33.     }
34. }
```

Zdrojový kód 3 – SignupActivity.java, zpracování výsledku přihlášení

Do této metody přichází tři argumenty kód požadavku (ř. 2), výsledný kód (ř. 3) a data v objektu *Intent* (ř. 4). Cílem této metody je zkontrolovat výsledek požadavku a různé situace ošetřit.

Nejdříve je nutné provést kontrolu typu požadavku, který jsme vytvořili v předchozí metodě (ř. 6). Poté lze získat objekt obsahující data o odpovědi požadavku (ř.7-8). Ten bude využit při ošetření neúspěšného přihlášení. Nejdříve však zkontrolujeme,

zda je kód výsledku úspěšný (ř. 10). V tom případě je možné spustit hlavní aktivitu aplikace (ř. 11-15).

Při neúspěchu (ř. 17) řešíme dvě situace. Pokud v průběhu přihlašování uživatel proces přerušil, bude objekt odpovědi null (ř. 18). Druhou situací je nedostupnost připojení k internetu (ř.25-26), chybový kód získáme z objektu odpovědi (ř. 26). Třída *ErrorCodes* je součástí balíčku *firebase.ui.auth.** a obsahuje konstanty pro identifikaci různých chyb. V obou případech informujeme uživatele o neúspěchu (ř. 19-22, 27-30) .

8.4 Cloud Firestore

Platforma Firebase poskytuje dvě služby real-time databáze, které byly popsány v kapitole 6. V této kapitole budeme implementovat službu Firestore, která umožní realizaci dvou hlavních funkcí aplikace. Konkrétně uchování dat vygenerovaných uživatelem, která jsou využita pro jejich vyhledávání a chat, zajištěný real-time synchronizací dat v databázi.

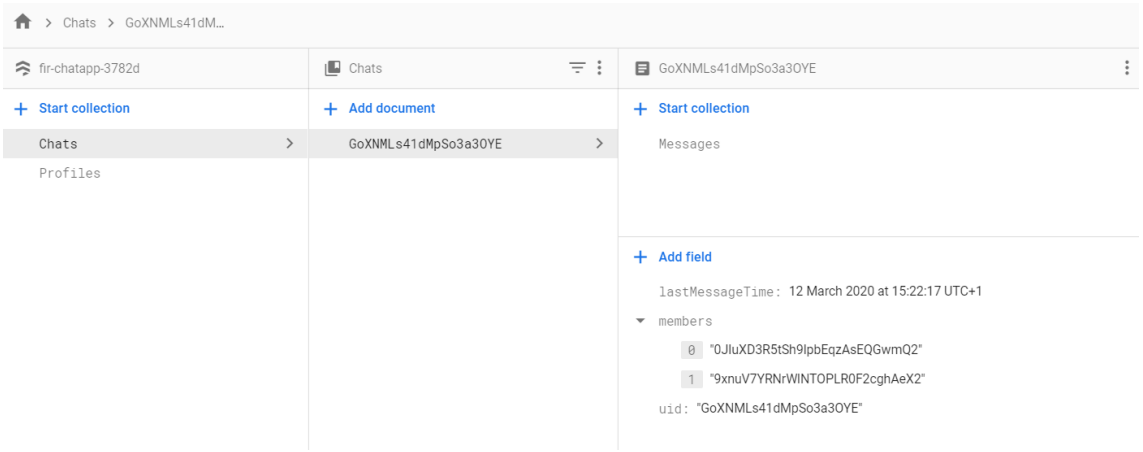
8.4.1 Nastavení služby ve webové konzoli

Prvním nastavením této služby, jsou bezpečnostní pravidla pro přístup k databázi. Jejich úvodní nastavení lze provést při aktivaci této služby, později jsou pravidla dostupná v kategorii Database > Rules.

Obrázek 16 – Specifikace pravidel pro přístup k databázi ve webové konzoli

Na obrázku je specifikováno pravidlo (ř. 5) umožňující přihlášeným uživatelům číst a zapisovat do databáze.

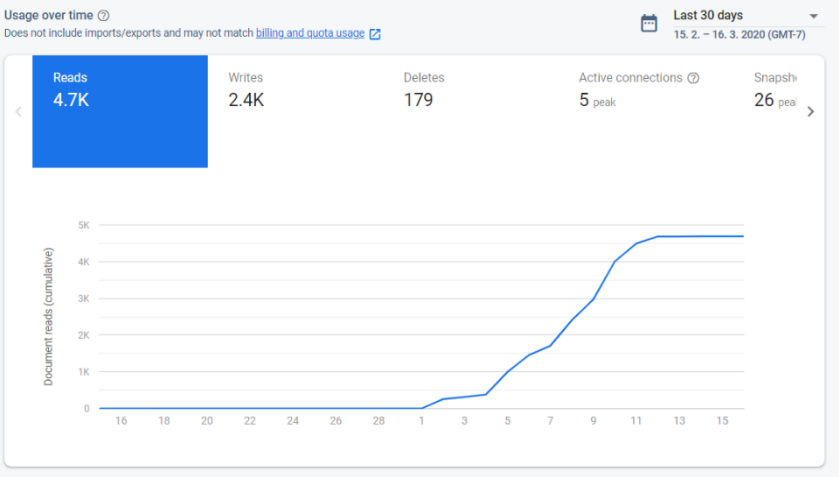
V konzoli této služby jsou dostupné další tři kategorie, jejich účel je především poskytnout informace a přehled o databázi. V kategorii Database > Data jsou dostupná data uložená v databázi. Zde je možné prohlížet všechny kolekce a dokumenty, a zároveň data přidávat nebo mazat. To může být užitečné pro vytváření testovacích dat. Skutečná data jsou pak přidávána přes aplikaci.



Obrázek 17 – Náhled do databáze z webové konzole

Při ukládání dat do Firestore databáze automaticky indexuje každé pole dokumentu. Indexování umožňuje velmi rychlé prohledávání databáze. Automaticky jsou indexy přidávány pouze pro samotná pole. Pokud je potřeba použít dotaz, který kombinuje více polí (např. vyhledání chatů a poté jejich seřazení dle času poslední zprávy) je nutné vytvořit vlastní Composite Index. Pro vytvoření slouží kategorie Database > Index zde je možné indexy přidávat nebo mazat. Tyto indexy je také obvykle možné vygenerovat přímo z vývojového prostředí, jako je například Android studio.

Poslední kategorii této služby je Database > Usage, která poskytuje přehled o využití služby databáze. Konkrétně měří počty čtení, zápisů, smazání, aktivních připojení a snapshot posluchače (které slouží k detekci změn v databázi a zajišťují real-time aktualizaci dat).

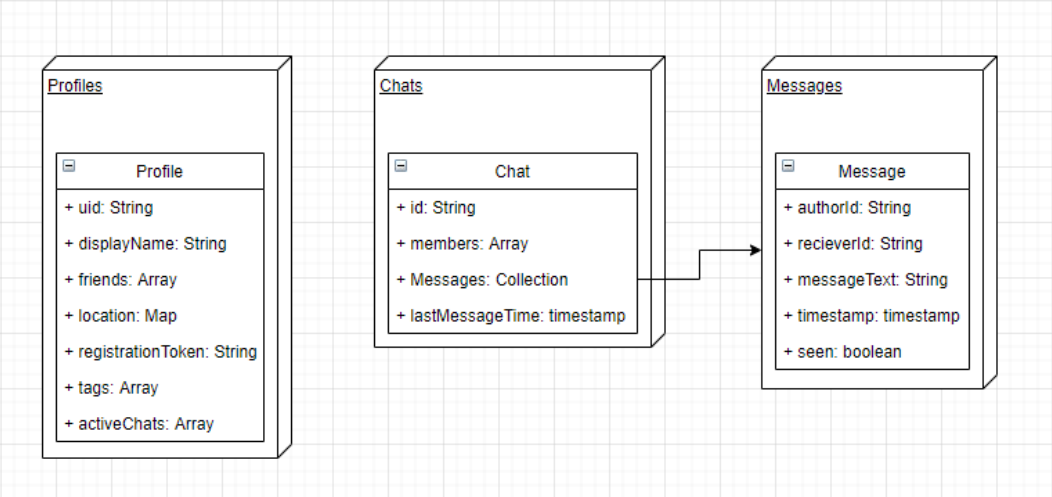


Obrázek 18 – Přehled využití databáze ve webové konzoli

8.4.2 Modely a struktura databáze

Pro práci s daty v aplikaci se využívají třídy modelu. Tyto třídy reprezentují dokumenty, které jsou ukládány a čteny z databáze. Firestore umožňuje automatické převádění objektů modelu na dokumenty a opačně. Pro správné fungování automatického mapování objektů na dokumenty musí třídy zahrnovat metody get a set, které odpovídají standardnímu pojmenování. Tedy metoda „getUsername()“ bude vracet hodnotu pole „username“. Musí také obsahovat prázdnou verzi metody konstruktoru.

V aplikaci bylo nutné vytvořit modelovou třídu Profile reprezentující profil uživatele, který může uživatel z aplikace upravovat. A dále dvě modelové třídy Chat a Message, které umožňují funkci chatování.



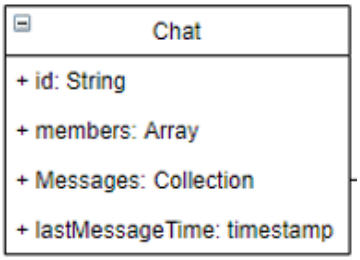
Obrázek 19 - Návrh databáze

Modelové třídy jsou převedeny na konkrétní dokumenty a uloženy v odpovídajících kolekcích. Databáze zahrnuje tři kolekce. Profiles obsahuje profily uživatelů, Chats obsahuje dokumenty reprezentující chat. Dokumenty typu Chat zahrnují vnořenou kolekci Messages, ve které jsou uloženy všechny zprávy daného chatu. Jednotlivé kolekce nemohou mít mezi sebou žádné vazby. Pokud modelujeme související kolekce, pak využijeme právě jejich zanoření. Při vytváření odkazu mezi dokumenty použijeme pole, které má hodnotu odpovídající ID souvisejícího dokumentu.

Prvním typem dokumentu je Profile. Tento dokument je inicializován při první úpravě dat uživatelem. Pro inicializaci profilu je využito jedinečné ID, které bylo automaticky vygenerováno službou Firebase Autentizace při registraci uživatele. Každý přihlášený uživatel má tedy profil, jehož data může v rámci aplikace upravovat.

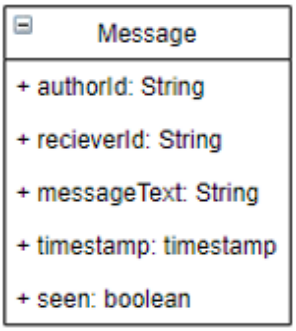
Obrázek 20 - Dokument typu Profile

Druhým typem dokumentu je Chat, tento dokument je vytvořen při inicializaci chatovací aktivity mezi dvěma uživateli.



Obrázek 21 - Dokument typu Chat

Posledním typem dokumentu je Message, který je vytvořen při zaslání nové zprávy.



Obrázek 22 - Dokument typu Message

8.4.3 Připojení služby k aplikaci

Stejně jako tomu bylo u služby Firebase Autentizace, služba Firebase vyžaduje připojení pomocí závislosti v souboru gradle:
implementation 'com.google.firebase:firebase-firestore:21.4.1'

Tato závislost umožní používání balíčku *com.google.firebase.firestore.**, který obsahuje všechny potřebné třídy pro přístup k databázi Firestore.

8.4.4 Základní využití a přístup k databázi

Pro práci s databází je prvním krokem získání její instance. Tato instance bude základem pro všechny přístupy k databázi. Získáme jí pomocí statické metody třídy *FirebaseFirestore*.

```
1. FirebaseFirestore firestore = FirebaseFirestore.getInstance();
```

Při práci s instancí databáze může v kódu docházet k častému opakování volání stejných nebo velmi podobných metod, a to může vést i k nepřehlednosti kódu. Pro řešení toho problému můžeme vytvořit vlastní třídu *FirestoreService*. Účelem této třídy je seskupení často opakovaných volání metod objektu databáze (např. přístup k profilu přihlášeného uživatele nebo opakované dotazování) a metod vytvořených pro specifické účely (např. založení nového chatu).

```
1. private final FirebaseUser signedUser;
2. private final FirebaseFirestore firestore;
3. private final CollectionReference chatsReference;
4. private final CollectionReference profilesReference;
```

Zdrojový kód 4 – FirestoreService.java, objekty

Atributy třídy *FirestoreService* jsou objekt uživatele *signedUser*, který je později využit k získání jedinečného id přiřazeného službou Firebase Autentizace, dále objekt *firestore*, který je odkazem na instanci databáze a nakonec dvě reference na kořenové kolekce Profiles a Chats.

K získání odkazu na konkrétní část databáze slouží dvě metody, jejichž řetězením specifikujeme cestu k požadovanému dokumentu či kolekci.

```
1. firestore.collection("NazevKolekce").document("id");
```

Pokud se jedná o kolekci je parametrem její název, v případě dokumentu pak jeho konkrétní id. Pokud specifikovaná kolekce nebo dokument neexistuje, jsou automaticky vytvořeny. V případě vytváření nového dokumentu je také nutné poskytnout objekt, který v něm bude uložen. Jestliže vytváříme nový dokument s automaticky generovaným id, voláme bez parametru.
Atributy třídy *FirestoreService* jsou iniciovány v konstruktoru třídy.

```
1. public FirestoreService() {
2.     signedUser = FirebaseAuth.getInstance().getCurrentUser();
3.     firestore = FirebaseFirestore.getInstance();
4.     chatsReference = firestore.collection("Chats");
5.     profilesReference = firestore.collection("Profiles");
6. }
```

Zdrojový kód 5 – FirestoreService.java, konstruktor třídy

Pro objekt přihlášeného uživatele a referenci databáze využijeme statické metody tříd *FirebaseAuth* a *FirebaseFirestore* (ř. 2-3). Poté iniciujeme odkazy na kořenové kolekce databáze (ř. 4-5).

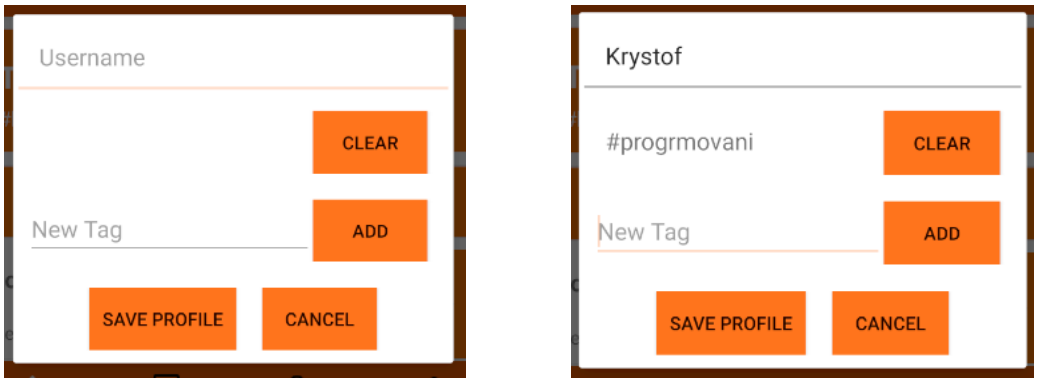
8.4.5 Práce s profilem uživatele

První metoda třídy FirestoreService *„getSignedUserDocumentRef()“* slouží pro přístup k dokumentu přihlášeného uživatele.

```
1. public DocumentReference getSignedUserDocumentRef() {
2.     return profilesReference
3.         .document(signedUser.getUid());
4. }
```

Zdrojový kód 6 – FirestoreService.java, metoda getSignedUserDocumentRef()

Metoda vrací objekt typu DocumentReference, který reprezentuje odkaz na jeden dokument databáze. Pro specifikaci dokumentu bylo použito id přihlášeného uživatele.



Obrázek 23 – Dialog pro úpravu uzivatele v aplikaci

Aplikace uživateli umožňuje, nastavení uživatelského jména „displayName“, které bude použito v rámci aplikace a specifikaci zájmů „tags“, které slouží pro vyhledávání. Nastavení probíhá pomocí dialogu, který je přístupný na domovské stránce aplikace, která je vždy zobrazena jako první po přihlášení. Zde uživatel zvolí své uživatelské jméno a vytvoří seznam zájmů v podobě tagů, dle kterých poté probíhá vyhledávání. Tyto informace může uživatel kdykoliv měnit.

```
1. firestoreService.getSignedUserDocumentRef()
2.     .set(sigendUserObject);
```

Zdrojový kód 7 – HomeFragment.java, uložení údajů uživatele

K uložení objektu nejdříve zvolíme dokument a poté zavoláme metodu „*set*()“ třídy *DocumentReference*. V případě prvního ukládání dochází k inicializaci dokumentu, jinak dochází pouze k aktualizaci změněných dat. Firestore automaticky převádí objekt typu User na dokument typu Profile.

Po nastavení profilu, můžeme přejít k vyhledávání uživatelů. K pohybu mezi jednotlivými stránkami slouží navigace ve spodní části obrazovky. Vyhledávání je dostupné na stránce Search.

Obrázek 24 – Navigační menu aplikace

8.4.6 Funkčnost geolokace

Databáze Firestore umožňuje uložení zeměpisných souřadnic v objektu typu GeoPoint. Firestore ani jiné služby platformy neposkytují žádnou funkčnost pro využití této lokace jako základ pro vyhledávání. Problém bylo nutné vyřešit na straně klienta. Pro účely aplikace tedy zjišťujeme aktuální souřadnice. Namísto jejich ukládání je však využijeme k získání informací o aktuální adrese. Poté jsou tyto informace uloženy v databázi Firestore a využity pro vyhledávání.

```
1. private Map<String, String> address = new HashMap<>();
2. …
3. address.put(„Country“, currentAddress.getCountryName());
4. address.put(„Region“, currentAddress.getAdminArea());
5. address.put(„City“, currentAddress.getSubAdminArea());
```

Zdrojový kód 8 – SearchFragment.java, vytvoření objektu adresy

Adresu ukládáme jako objekt typu *Map*. Pokud úspěšně získáme aktuální souřadnice, jsou využity pro vygenerování tří informací o adrese. Pro účely aplikace je potřeba získat Město, Region (kraj) a zemi (stát).

Vytvořenou adresu uložíme do profilu přihlášeného uživatele. Pro upravení specifického pole dokumentu byla vytvořena metoda *"updateField()"* ve třídě FirestoreService.

```
1. public void updateField (
2.     final String collection,
3.     final String document,
4.     final String field,
5.     final Object value) {
6.         firestore.collection(collection).document(document)
7.             .update(field, value);
8. }
```

Zdrojový kód 9 – FirestoreService.java, metoda updateField()

Metoda vyžaduje název kolekce obsahující dokument (ř. 2), id dokumentu (ř. 3), název pole, které chceme aktualizovat (ř. 4) a hodnotu, kterou do daného pole ukládáme (ř. 5).

```
1. firestoreService.updateField(
2.     "Profiles", signedUser.getUid(), "location", address
3. );
```

Zdrojový kód 10 – SearchFragment.java aktualizace adresy

Při aktualizaci adresy tedy specifikujeme dokument přihlášeného uživatele, pole „location" a objekt adresy.

Pokud se nepodařilo získat aktuální souřadnice, aplikace se pokusí načíst poslední uloženou adresu. Pro načtení adresy je nutné získat objekt uložený v dokumentu Profile přihlášeného uživatele.

```
1. firestoreService.getSignedUserDocumentRef().get()…
```

Na odkazu dokumentu tedy zavoláme asynchronní metodu *„get()"*, jejíž návratová hodnota je typu *„Task<DocumentSnapshot>"*. Pro obsluhu asynchronního volání umožňuje Firestore připojení posluchače, který čeká na jeho vyřízení (ř. 2).

```
2. .addOnSuccessListener(
3.     onSuccess (DocumentSnapshot documentSnapshot){
4.
5.         Map<String, String> location =
6.             documentSnapshot.toObject(User.class).getLocation();
7.
8.         if(location != null && location.size > 0) {
9.
10.            address.put(„Country", location.getCountryName());
11.            address.put(„Region", location.getAdminArea());
12.            address.put(„City", location.getSubAdminArea());
13.         }
14.  });
```

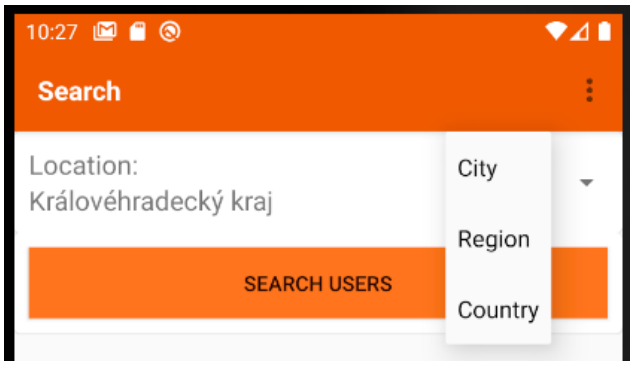
Zdrojový kód 11 – SearchFragment.java, naplněni objektu adresy

Dokument je poté převeden na objekt typu User, ze kterého můžeme načíst adresu (ř. 5-6). Pro převod objektů DocumentSnapshot slouží metoda *„toObject()"*, která převede dokument na objekt specifikované třídy. V případě že načítáme množinu dokumentů stejného typu můžeme využít metodu *„toObjects()"*.

Jestliže se nepodaří získat ani starší lokaci (nelze se připojit k databázi nebo zde nejsou uložena data), pak nelze použít vyhledávání a o situaci informujeme uživatele.

8.4.7 Vyhledávání uživatelů

Pokud je načtení lokace úspěšné, umožníme vyhledávání uživatelů. Lokace je zobrazena v UI a poskytneme možnost vybrat rozsah vyhledávání, z jednotlivých položek adresy. Uživatel tedy může vyhledávat v rámci města, regionu nebo země.



Obrázek 25 - UI pro vyhledávání uživatel

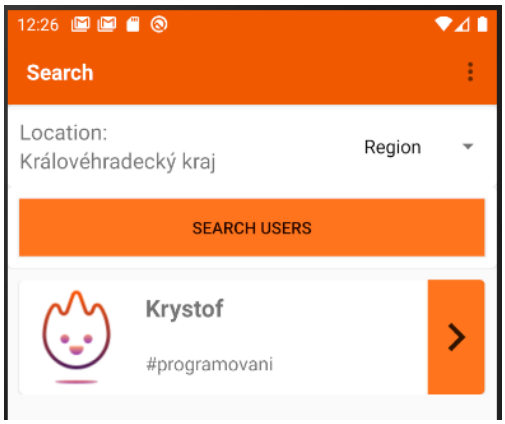
Vyhledávání probíhá ve dvou krocích. Nejdříve jsou nalezeny dokumenty obsahující vybranou adresu a poté kontrolujeme, zda se shodují data o zájmech (tagy). Pro tento účel byla vytvořena metoda *„searchUsersQuery()“* ve třídě *FirestoreService*.

```
1. public Task<QuerySnapshot> searchUsersQuery(
2.     final String locationField,
3.     final String userLocation,
4.     final List<String> tags) {
5.         return profilesReference
6.             .whereEqualTo(locationField, userLocation)
7.             .whereArrayContainsAny("tags", tags)
8.             .limit(10).get();
9. }
```

Zdrojový kód 12 – FirestoreService.java, metoda searchUsersQuery()

V případě této metody je návratovou hodnotou *„Task< QuerySnapshot >“.* Volání je opět asynchronní, získáváme však kolekci dokumentů odpovídajících podmínkám dotazu (ř. 5). První podmínkou dotazu specifikujeme dokumenty, které v adrese mají vybranou hodnotu (např. v poli adresy „location.Region“ je hodnota „Královéhradecký kraj“) (ř. 6).

Poté využíváme seznam zájmů („tags“). Dotazy podporované službou Firestore, které pracují s poli obsahujícími seznam mají několik omezení. Máme možnost hledat absolutní shodnost seznamů (seznamy jsou stejně dlouhé, všechny prvky jsou stejné a mají stejné pořadí). Druhou možností je využití metody *„whereArrayContainsAny()“*. Tato metoda hledá alespoň jeden prvek, který je v obou seznamech. Firestore nám neumožní specifikovat ani předem zjistit kolik prvků se bude shodovat. Pro optimální řešení problému, kdy bychom chtěli dokumenty řadit dle shody, neexistuje ve službě Firestore řešení. Pro účely aplikace byla tedy využita metoda pro nalezení alespoň jedné shody a počet načítaných dokumentů omezen na 10.



Obrázek 26 - Zobrazení nalezených uživatel

Aplikace zobrazí seznam profilů, s jejich atributem „displayName“ a jejich zájmy. Jednotlivé prvky tohoto seznamu umožňují iniciaci chatovací aktivity.

8.4.8 Inicializace chatu

O inicializaci nového chatu se stará metoda *„initializeNewChat()“* třídy FirestoreService.

```
1. public String initializeNewChat(
2.     final String otherUserId) {
3.
4.   DocumentReference newChatReference =
5.         chatsReference.document();
6.
7.   String newId = newChatReference.getId();
8.   List<String> members = new ArrayList<>();
9.   members.add(signedUser.getUid());
10.    members.add(otherUserId);
11.    Chat chat = new Chat(newId, members);
12.    chatsReference.document(newId).set(chat);
13.
14.    getSignedUserDocumentRef()
15.        .update("activeChats", FieldValue.arrayUnion(newId));
16.
17.    return newId;
18.  }
```

Zdrojový kód 13 – FirestoreService.java, metoda inicializeNewChat()

V metodě zakládáme nový dokument typu Chat. Pro jeho založení potřebujeme seznam členů, které získáme v argumentu a z objektu přihlášeného uživatele (ř. 7-9). Poté můžeme vytvořit nový objekt typu Chat, který do odkazu na dokument uložíme. Nakonec ještě přidáme nový chat mezi aktivní chaty přihlášeného uživatele (ř. 14-15).

Pro přidání aktivního chatu uživatelům slouží metoda *„addChatToActive()“* třídy FirestoreService.

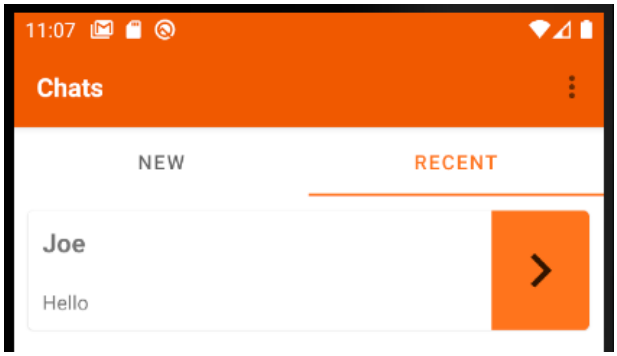
```
1. public void addChatToActive(final String chatId) {
2.   profilesReference.document(signedUser.getUid())
3.           .update(„activeChats“, FieldValue.arrayUnion(chatId);
4. }
```

Zdrojový kód 14– FirestoreService.java, metoda addChatToActive()

Účelem metody je aktualizovat pole „activeChats“, které obsahuje seznam ID aktivních chatů. Firestore poskytuje dvě metody pro úpravy seznamů. Jedná se o *„arrayUnion()“*, která přidává prvek do seznamu a *„arrayRemove()“* prvek odstraní.

Načteme tedy dokument uživatele (ř. 2) a pomocí metody *„update()“* aktualizujeme pole „activeChats“ přidáním nového id (ř. 3).

Atribut aktivních chatů společně s touto metodou nám umožní rozdělení chatů uživatele do dvou kategorii. Uživatel najde své chaty na stránce „Chats“, zde jsou záložky „NEW“ a „RECENT“. Záložka „NEW“ zobrazuje nové chaty uživatele, které zatím nemá v seznamu aktivních. „RECENT“ pak zobrazuje již aktivní chaty. Oba tyto seznamy jsou seřazeny dle času poslední zprávy.



Obrázek 27 – Zobrazení nedávných chatů

```
1. public Query queryByWhereIn(
2.   final String Colection,
3.   final String field,
4.   final List<String> values) {
5.
6.   return firestore.collection(collection)
7.           .whereIn(field, values);
8. }
```

Zdrojový kód 15 – FirestoreService.java, metoda queryByWhereIn()

V záložce „Recent“ jsou zobrazeny nejnovější zprávy v aktivních chatech. K tomu byla vytvořena metoda *„queryByWhereIn()“* využívající metodu databáze firestore *„whereIn()“*. Účelem této metody je načtení dokumentů ze seznamu aktivních chatů uživatele.

```
1. firestoreService.queryByWhereIn(„Chats", „uid", chatIds)
2.   .orderBy(„lastMessageTime", Query.Direction.DESCENDING)
```

Zdrojový kód 16 – SearchFragment.java, použití metody queryByWherIn()

Metodu *„queryByWhereIn()"* použijeme pro hledání dokumentů v kolekci Chats, jejichž uid je v seznamu aktivních chatů přihlášeného uživatele (ř. 1).

```
1. public Query queryByArrayContains(
2.     final String collection,
3.     final String array,
4.     final Object value) {
5.
6.     return firestore.collection(collection)
7.         .whereArrayContains(array, value);
8. }
```

Zdrojový kód 17 – FirestoreService.java, metoda queryByArrayContains()

Pro načtení kolekce dokumentů v záložce NEW využijeme metodu *„queryByArrayContains()"* třídy FirestoreService, která kontroluje seznam v dokumentu obsahuje jednu specifickou hodnotu.

```
1. firestoreService
2.     .queryByArrayContains("Chats","members",signedUser.getUid())
3.     .orderBy("lastMessageTime",Query.Direction.DESCENDING)
```

Zdrojový kód 18 – NewConversationFragment.java, využití metody queryByArrayContains()

Metodu *„queryByArrayContains()"* využijeme pro načtení dokumentů z kolekce „Chats", které mají v poli „members" id přihlášeného uživatele (ř. 2). Získáme tak kompletní seznam chatů přihlášeného uživatele, který na straně klienta můžeme přefiltrovat a zbavit se již aktivních. Výsledkem je seznam nových chatů. Po otevření chatu tohoto seznamu, je jeho id přidáno mezi aktivní.

8.4.9 Real-time chat

V aplikaci najdeme několik seznamů chatů a uživatelů, které umožňují iniciaci chatovací aktivity. Prvním krokem spuštění této aktivity je načtení zpráv chatu. Využijeme metodu *„queryMessagesCollection()“* třídy FirestoreService.

```
1. public Query queryMessageCollection(final String chatId) {
2.     return chatsReference.document(chatId)
3.         .collection("Messages")
4.         .orderBy("timestamp", Query.Direction.ASCENDING);
5. }
```

Zdrojový kód 19 – FirestoreService.java, metoda queryMessageCollection()

V metodě specifikujeme id vybraného chatu (ř. 2) a poté vnořenou kolekci Messages (ř. 3). Nakonec dokumenty kolekce seřadíme podle času poslední zprávy. Metoda vrací objekt „Query“ reprezentující dotaz. Ten je poté obsloužen chatovací aktivitou.

Pro realizaci real-time aktualizace zpráv poskytuje Firestore objekt typu „SnapshotListener“, který lze připojit na dotaz nebo odkaz na část databáze. Objekt čeká na provedení změn v dané části databáze (např. změna dat dokumentu nebo přidání dokumentu do kolekce). Při zaznamenání takové události jsou změny automaticky načteny a je proveden kód metody onEvent. V metodě máme přístup k objektu typu QuerySnapshot, ze kterého můžeme získat a použít data.

```
1. firestoreService.queryMessageCollection(chat)
2.     .addSnapshotListener(new EventListener<QuerySnapshot>() {
3.         @Override public void onEvent(
4.             @Nullable QuerySnapshot queryDocumentSnapshots,
5.             @Nullable FirebaseFirestoreException e) {
6.         if(queryDocumentSnapshots != null) {
7.
8.             List<Message> messagesList =
9.                 queryDocumentSnapshots.toObjects(Message.class);
10.            for(
11.                DocumentSnapshot doc :
12.                queryDocumentSnapshots.getDocuments()){
13.
14.                if(doc.getString("recieverId")
15.                    .equals(signedUser.getUid(){
16.
17.                    doc.getReference().update("seen", true);
18.                }
19.            }
20.         }
21.     }
22.  });
```

Zdrojový kód 20 – MessagingActivity.java, použití metody queryMessageCollection()

Na dotaz připojíme SnapshotListener (ř. 2), který bude čekat na událost přidání nové zprávy. Z načtených dokumentů vytvoříme seznam zpráv (ř. 8-9) a pokud jsou zobrazeny uživatelem, kterému byly určeny (ř. 14-15), jsou označeny jako přečtené (ř. 17). Dokud je uživatel v této aktivitě, je SnapshotListener aktivní a dochází k aktualizaci seznamu zpráv zobrazených uživateli na základě vytvořeného dotazu. V aplikaci byl SnapshotListener využit na více místech, například seznamy chatů, které zobrazují poslední zprávu každého chatu. Pokud uživatel obdrží zprávu při prohlížení nedávných chatů, seznam se aktualizuje a chat je přemístěn na první pozici s aktuální zprávou.

Pro zaslání nové zprávy, byla vytvořena metoda *„getEmptyMessageDocument()“* ve třídě FirestoreService.

```
1. public DocumentReference getEmptyMessageDocument(
2.     final String chatId){
3.         return chatsReference
4.             .document(chatId)
5.             .collection("Messages")
6.             .document();
```

Zdrojový kód 21 – FirestoreService.java, metoda queryMessageCollection()

Metoda vrací odkaz na prázdný dokument zprávy. Do dokumentu je poté uložen objekt typu Message, který je vytvořen na základě informací získaných z uživatelského prostředí.

```
1. firestoreService.updateField(
2.     "Chats",
3.     chatId,
4.     "lastMessageTime",
5.     newMessage.getTimestamp()
6. );
```

Zdrojový kód 22 – MessagingActivity.java, využití metody updateField()

Pro účely řazení chatů je při zaslání zprávy aktualizováno pole „lastMessageTime“. Pro aktualizaci je využita metoda *„updateField()“* třídy FirestoreService. Tato metoda je také využita pro správu seznamu přátel uživatele. V aktivitě chatu umožňujeme přidání uživatele mezi přátele.

```
1. firestoreService.updateField(
2.     "Profiles",
3.     signedUser.getUid(),
4.     "friends",
5.     FieldValue.arrayUnion(profile.getUid())
6. );
```

Zdrojový kód 23 – MessagingActivity.java, využití metody updateField()

Pro přidání přátel specifikujeme zapisovanou hodnotu pomocí statické metody *„arrayUnion()“* třídy „FieldValue“, která je součástí služby Firestore. Tato metoda přidá vložený argument do seznamu v poli. Seznam přátel je možné zobrazit v sekci „Friends“.

```
1. firestoreService.updateField(
2.     "Profiles",
3.     signedUser.getUid(),
4.     "friends",
5.     FieldValue.arrayRemove(profile.getUid())
6. );
```

Zdrojový kód 24 – MessagingActivity.java, využití metody updateField()

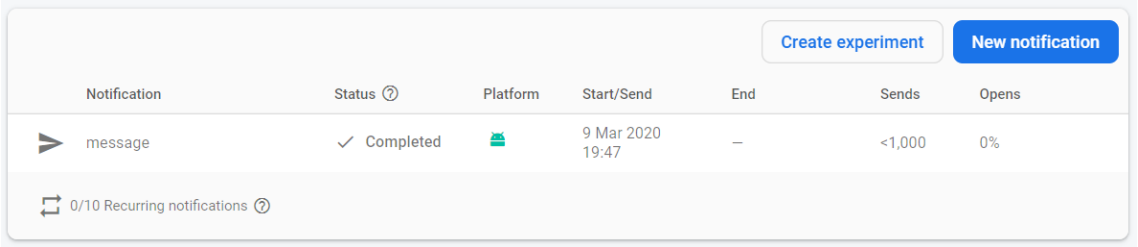
Ze seznamu je možné přátele odstranit. Odstranění probíhá analogicky, namísto *„arrayUnion()“* však použijeme možnost *„arrayRemove()“* (ř. 5), která najde hodnotu rovnou argumentu a odstraní ji.

8.5 Cloud Messaging

Poslední požadovanou funkčností aplikace bylo zasílání upozornění, když uživatel obdrží novou zprávu. Zasílání upozornění nebo zpráv umožňuje služba Firebase Cloud Messaging, která byla popsána v kapitole 7.3.

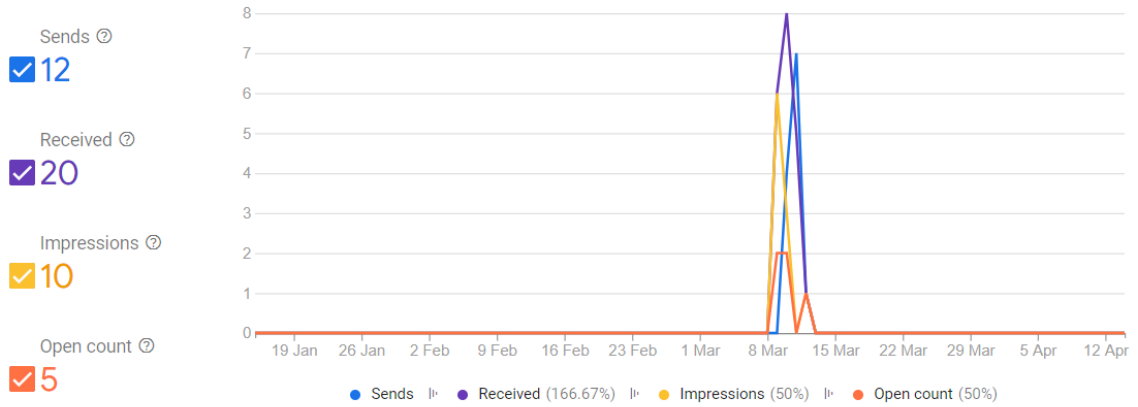
8.5.1 Používání služby

Samotná služba Cloud Messaging vyžaduje pouze připojení závislosti k aplikaci. Po připojení je možné zasílat zprávy vytvořené ve webové konzoli služby. Ve webové konzoli najdeme dvě kategorie. Notifications zobrazuje přehled upozornění a možnost vytvoření nového upozornění pomocí dialogu.



Obrázek 28 – Přehled Cloud Message úpozornění ve webové konzoli

Druhou kategorií je pak Reports, která zobrazuje využití služby.



Obrázek 29 - Přehled využití služby Cloud Messaging we webové konzoli

8.5.2 Připojení služby

Nejprve připojíme závislost služby Cloud Messaging.

implementation 'com.google.firebase:firebase-messaging:20.1.5'

Poté službu přidáme do souboru AndroidManifest.

```
1. <service
   android:name=".services.MessageNotificationsService"
   android:exported="false">
2.  <intent-filter>
3.    <action android:name="com.google.firebase.MESSAGING_EVENT" />
4.  </intent-filter>
5. </service>
```

Zdrojový kód 25 – AdnroidManifest, závislost služby cloud messaging

Tyto dvě části umožní zařízení využít jako klient pro přijetí zasílané zprávy.

8.5.3 Zasílání upozornění

Jak už bylo zmíněno, požadovanou funkčností aplikace je zasílání a přijímání upozornění při obdržení nové zprávy v chatu. Při spuštění události jedním klientem (zaslaní zprávy v chaty) potřebujeme vygenerovat upozornění s informacemi o zprávě, které je poté zasláno cílovému zařízení. Služba Cloud Messaging však přímo nepodporuje zasílání upozornění mezi dvěma konkrétními klienty, což znemožňuje implementaci požadované funkčnosti. Pro řešení této situace bylo nutné využití kombinace Cloud Messaging spolu se službou Cloud Functions.

8.5.4 Cloud Functions

Služba Cloud Functions nebyla součástí teoretické části práce, bude tedy představena pouze pro řešení této situace. Jedná se o framework umožňující vytvoření funkcí, které jsou automaticky spuštěny a provedeny v reakci na události, vygenerované dalšími službami (např. změna v databázi). Můžeme tedy vytvořit funkci, která při uložení nové zprávy do databáze vygeneruje a odešle upozornění službou Cloud Messaging. Implementace funkcí je v tuto chvíli možná pouze s využitím systému NodeJS.

8.5.5 Implementace cloud funkce

Pro práci se službou importujeme její modul Firebase-functions. Dále pak budeme potřebovat modul Firebase-admin umožňující přístup do dalších služeb propojených s aplikací.

```
1. const functions = require('firebase-functions');
2. const admin = require('firebase-admin')
```

Zdrojový kód 26 – závislosti pro vytvoření cloud funkce

Při odeslání zprávy v chatu cílové klientské zařízení obdrží upozornění, přes které je možné konkrétní chat v aplikaci otevřít.

```
1. admin.initializeApp();
2.
3. exports.notifyNewMessage = functions.firestore
4.     .document('Chats/{chat}/Messages/{message}')
5.     .onCreate((docSnapshot, context) =>{…}
```

Zdrojový kód 27 – Nastavení posluchače na cestu v databázi

Inicializujeme aplikaci a poté exportujeme funkci připojenou na firestore databázi. K databázi specifikujeme konkrétní cestu, kde bude zpráva uložena. Nakonec zavoláme metodu onCreate, která čeká na vytvoření nového dokumentu.

```
1. const message = docSnapshot.data();
2. const recieverId = message.recieverId;
3. const authorId = message.authorId;
4.
5. return admin.firestore()
6.     .doc('Profiles/' + authorId)
7.     .get()
8.     .then(authorDoc => { }
```

Zdrojový kód 28 – Callback funkce onCreate(), načtení informací o zprávě

Callback funkce onCreate přijímá parametr docSnapshot. Ten využijeme pro získání informací o zprávě, konkrétně jejího autora, příjemce a text, a poté načteme dokument autora zprávy.

```
const authorName = authorDoc.get('displayName');

    return admin.firestore()
        .doc('Profiles/' + recieverId);
        .get().then(userDoc => {

        const registrationToken =
                    userDoc.get('registrationToken');
        const notificationBody = (message.messageText);

         const payload = {
                notification: {
                    title: 'New message from ' + authorName,
                    body: notificationBody,
                    clickAction: 'MessagingActivity'
                },
                data: {
                    userid: authorId
                }
         }
```

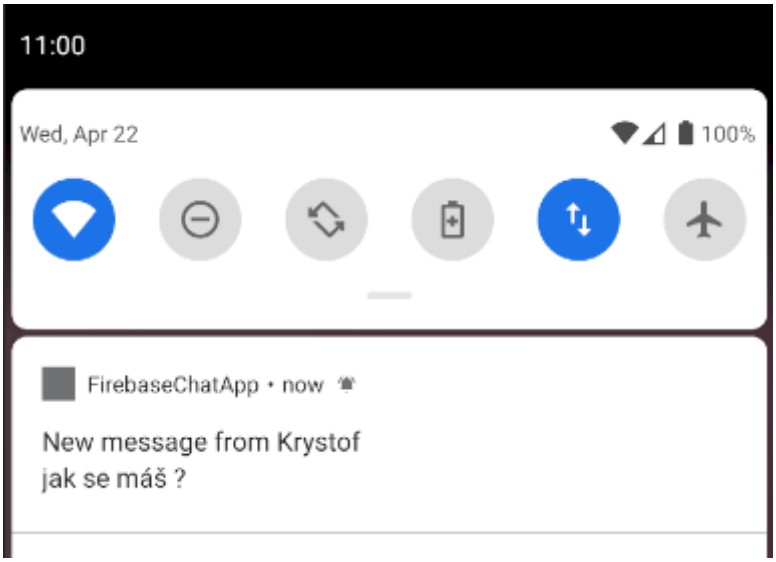
Zdrojový kód 29 – Vytvoření dat pro naplnění objektu upozornění

Dále načteme dokument uživatele, kterému je upozornění zasíláno. Z tohoto dokumentu získáme registrační token klienta. Vytvoříme objekt „payload“, který obsahuje všechny informace pro nastavení notifikace. Specifikujeme „title“, ve kterém využijeme jméno odesilatele. Tělem upozornění pak bude samotná zpráva. Nakonec specifikujeme spuštění aktivity chatu po kliknutí na upozornění.

```
return admin
    .messaging()
    .sendToDevice(
    registrationToken, payload)
```

Zdrojový kód 30 – Odeslání upozornění

Posledním krokem je zaslání upozornění. Využijeme zde přístup ke službě Cloud Messaging, na které zavoláme metodu *„sendToDevice()“.* Jejími atributy bude registrační token cílového zařízení a dříve vytvořený objekt payload.



Obrázek 30 – Zobrazení upozornění na mobilním zařízení

Po odeslání zprávy uživateli je tedy vygenerováno upozornění, které je na cílové klientské zařízení zasláno.

9 Shrnutí výsledků

V rámci praktické části byla vytvořena aplikace s využitím služeb poskytovaných platformou Firebase. Z požadavků na aplikaci vyplynulo několik případů užití služeb platofrmy. Prvním případem bylo poskytnutí možnosti registace a přihlášení uživatelů k aplikaci. S využitím služby Firebase Autentizace jsme byli schopni vytvořit kompletní registraci a přihlašování uživatelů k aplikaci. Současně jsou uložené informace automaticky zabezpečeny platformou. Nebylo tedy nutné vytvářet vlastní řešení zabezpečení, jako je např. hashování hesel. Jedním nedostatkem u této služby je dle mého názoru fakt, že registrované uživatele nelze přímo ukládat do databáze Firestore. To vede k duplikaci dat reprezentujících uživatele. Při práci s modelem uživatele v kódu aplikace je pak nutný přístup k oběma službám namísto jen jedné.

K navržení modelu a struktury databáze jsme využili NoSQL databázi Firestore. Databáze umožňuje vytvořit libovolnou strukturu (model) databáze, avšak je potřeba ji využívat efektivně aby nevznikaly zbytečné požadavky. Cena za službu by se pak mohla významně zvýšit. Například v současné implementaci Chat-Zpráva dochází k velkému počtu čtení dokumentů (zpráv). Alternativním řešením by mohlo být ukládání zpráv jako seznamu jednoho dokumentu. Maximální velikost dokumentu v takovém případě ale omezuje počet zpráv.

Pro řešení chatu mezi uživateli jsme využili real-time synchronizaci databáze Firestore. Toto řešení se při testování jeví jako dostatečně rychlé a efektivní pro použití v produkci. Při využití posluchačů událostí poskytnutých platformou, dochází při změně dat v databázi k téměř okamžité aktualizaci na straně klienta.

Jako poslední jsme využili službu Cloud Messaging, která umožňuje zasílat upozornění. Pro správné fungování ji však bylo potřeba propojit s databází pomocí služby Cloud Funkce. To nám umožnilo zasílat upozornění při vytvoření nového dokumentu – zprávy.

I přes určitá omezení služeb se nakonec podařilo splnit všechny požadavky na funkčnost aplikace.

10 Závěr

V teoretické práci byla představena „Backend as Service“ platforma Firebase. Podrobně byly rozebrány její služby Firebase Autentizace, Databáze Firestore a Real-time Databáze, a nakonec služby pro optimalizaci a rozšíření aplikace. Záměrem této části bylo poskytnout teoretické základy o daných službách a usnadnit tak jejich implementaci při vývoji aplikací.

Praktická část byla zaměřena na implementaci služeb pro řešení konkrétních případů užití. Jednotlivé kapitoly představily možnosti služeb a způsob jejich implementace. Poté bylo demonstrováno jejich využití v kódu aplikace.

Práce tedy poskytuje základy pro porozumění a použití platformy při vývoji aplikací. Shrnuje možnosti služeb, jejich využití a představuje možné řešení různých případů užití. Zároveň i upozorňuje na některá omezení a specifické vlastnosti služeb na příkladu projektu praktické části.

Dle mého názoru může být práce přínosná zejména pro začínající vývojáře, kteří hledají backendové řešení pro jejich aplikace, nebo se o platformu Firebase přímo zajímají. Dále také pro představení možných řešení obvyklých požadavků aplikace, jako jsou autentizace nebo databáze.

11 Seznam použité literatury

[1] MICHAL, Májský, 2016, Analýza současných řešení Backend-as-a-Service pro vývoj mobilních a webových aplikací. *ČVUT DSpace* [online]. 14.6.2016. [cit. 26.1.2020]. Dostupné z: https://dspace.cvut.cz/handle/10467/65072

[2] Dokumentace | Firebase, [2014]. *Google* [online], [cit. 26.1.2020]. Dostupné z: https://firebase.google.com/docs

[3] MORONEY, Laurence, 2017, *The Definitive Guide to Firebase Build Android Apps on Googles Mobile Platform*. Berkeley, CA : Apress.

[4] ROUSE, Margaret, 2014, What is user authentication? - Definition from WhatIs.com. *SearchSecurity* [online]. 18.12.2014. [cit. 26.1.2020]. Dostupné z: https://searchsecurity.techtarget.com/definition/user-authentication

[5] KRHOVÁK, Jan and MATYÁŠ, Václav, 2011, Autentizace a identifikace uživatelů. *Autentizace a identifikace uživatelů* [online]. 14.11.2011. [cit. 26.1.2020]. Dostupné z: http://webserver.ics.muni.cz/bulletin/articles/560.html

[6] SETHI, Biswajeet, MISHRA, Samaresh and PATNAIK, Prasant Kumar, 2014, A Study of NoSQL Database. *International Journal of Engineering Research & Technology* [online]. 23.4.2014. [cit. 26.1.2020]. Dostupné z: https://www.ijert.org/a-study-of-nosql-database

[7] WILLIAMS, Trevoir, 2019, Relational SQL vs. Non-Relational NoSQL Databases. *The DEV Community* [online]. 11.7.2019. [cit. 26.1.2020]. Dostupné z: https://dev.to/trevoirwilliams/relational-sql-vs-non-relational-nosql-databases-hi5

[8] KHEDKAR, Sonam and THUBE, Swapnil, 2017, Real Time Databases for Applications. . *International Journal of Engineering Research & Technology* [online]. 2017. [cit. 26.1.2020]. Dostupné z: https://www.irjet.net/archives/V4/i6/IRJET-V4I6401.pdf

[9] MÁJSKÝ, Michal, 2016, Začínáme s Backend as a Service. *Zdroják* [online]. 8.8.2016. [cit. 26.1.2020]. Dostupné z: https://www.zdrojak.cz/clanky/zaciname-s-backend-service/

[10] SIERRA, Francisco García, 2018, Firebase Android Series: Crashlytics. *Medium* [online]. 12.7.2018. [cit. 26.1.2020]. Dostupné z: https://proandroiddev.com/firebase-android-series-crashlytics-29de3f507d6

[11] Techotopia, 2017, Firebase Performance Monitoring, 2017. *Techotopia* [online], 30.8.2017. [cit. 26.1.2020]. Dostupné z: https://www.techotopia.com/index.php/Firebase_Performance_Monitoring

[12] TURNER, Ashley, NoSQL vs SQL. *ScyllaDB* [online]. [cit. 26.1.2020]. Dostupné z: https://www.scylladb.com/resources/nosql-vs-sql/

12 Seznam obrázků

13 Seznam tabulek

14 Seznam zdrojových kódů