UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DIPLOMOVÁ PRÁCE

2020 Tomáš Valko

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Cloudová NoSQL databáze jako uložiště a zachycovač událostí

Tomáš Valko

Diplomová práce

2020

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2009/2010**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	[redacted]
Osobní číslo:	[redacted]
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	[redacted]
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

[redacted]

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

***Kathy Sierra, Bert Bates: Head First Java, O'Reilly Media 2005, počet stran 720, ISBN-10 1600330002**

***Kline Kevin, Kline Daniel, Hunt Brand: SQL in a Nutshell, O'Reilly Media 2008, počet stran 592, ISBN-10 0596518846**

***Ajit Sagar, Sue Spielman a kol.: Professional Java Server Programming J2Ee 1.4 Edition, Wrox Press 2003, počet stran 850, ISBN-10 1861008139**

***Vivek Chopra, Sing Li, Jeff Genender: Professional Apache Tomcat 6, WROX Press 2006, ISBN-10: 0471753610**

***Sun Microsystems: The Java EE5 Tutorial [online], 2007 [cit. 2009-10-08],dostupný z: http://java.sun.com/javaee/5/docs/tutorial/doc/docinfo.html**

Vedoucí bakalářské práce: [redacted]
Katedra informačních technologií

Datum zadání bakalářské práce: **15. ledna 2010**

Termín odevzdání bakalářské práce: **14. května 2010**

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. [redacted], Ph.D.
vedoucí katedry

V Pardubicích dne 31. března 2010

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 24. 5. 2020

Tomáš Valko

Poděkování

Rád bych na tomto místě poděkoval své rodině a přátelům za podporu během studia. Dále bych rád poděkoval vědoucí mé práce, paní Ing. Monice Borkovcová, Ph.D, za trpělivost s mým ne vždy stoprocentním přístupem a její cenné rády, které mě obohatily nad rámec této práce.

Anotace

Práce popisuje problematiku tzv. Real-Time databází a možnosti jejich využití v praxi. V práci jsou postupně rozebrány pojmy související s Real-Time databázemi spolu s principy fungování těchto databází a jejich optimálním použitím. Součástí práce je také vlastní implementace demonstrační aplikace spolu s popisem jejího vývoje a popisem její architektury.

Klíčová slova

NoSQL, cloudová databáze, dokumentové databáze, Real-Time databáze, cloud, klient, server, mobilní aplikace, BaaS, Firebase, Android, Java, JavaScript, Node.js

Title

Cloud NoSQL database as storage and event catcher

Annotation

The thesis discusses the issue of so-called Real-Time databases and the possibilities of theirs using in practice. This thesis explores related concepts with Real-Time databases, principles of these databases running and theirs optimal using. Part of the thesis is own implementation of demonstration application and a description of its development, architecture and startup instructions.

Keywords

NoSQL, cloud database, document stores, Real-Time databases, cloud, client, server, mobile app, BaaS, Firebase, Android, Java, JavaScript, Node.js

OBSAH

Seznam ilustrací a tabulek

Seznam zkratek a značek

ART	Android Runtime
ACID	Atomicity, Consistency, Isolation and Durability
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
BaaS	Backend as Service
BSON	Binary JSON
CSV	Comma-Separated Values
DBMS	Database Management System
HW	Hardware
IaaS	Infrastructure as a Service
ICT	Information and Communication Technologies
IO	Input-Output
JSON	JavaScript Object Notation
MVVM	Model View ViewModel
ORM	Object Relational Mapping
PaaS	Platform as a Service
POJO	Plain Old Java Object
RDBMS	Relational Database Management System
SaaS	Software as a Service
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
UI	User Interface
WSDL	Web Services Description Language
XML	Extensible Markup Language

Úvod

Databáze prošly od začátku nového tisíciletí značným vývojem. Již není pravidlem, že základem každé aplikace je relační databáze a před začátkem vývoje informačního systému se řeší maximálně volba konkrétního RDBMS. Postupně se ukázalo, že pro některé typy aplikací je relační databázový model zbytečně omezující a začaly se prosazovat NoSQL databáze zaměřené na výkon, flexibilitu a škálovatelnost.

Ani tyto vlastnosti však nemusí vždy dostačovat. Některé systémy totiž již ze své podstaty vyžadují mít klíčová data neustále aktuální, což NoSQL databáze nezajišťují a bývá tak kladena zátěž na vývojáře, aby toto chování zajistili. S novým pojetím a řešením přišly tzv. Real-Time databáze, které zachycují události spojené se změnou dat v databázi a automaticky tyto změny synchronizují do klientských aplikací.

Tato diplomová práce se primárně zabývá využitím právě tohoto typu databází. Z hlediska teorie si práce klade za cíl prozkoumat vlastnosti a možnosti, které tento databázový typ nabízí. Vzhledem k tomu, že Real-Time databáze většinou bývají zároveň NoSQL databáze, je znalost těchto databází nezbytná pro pochopení jejich fungování. V práci jsou tak zároveň stručně popsány i NoSQL databáze. Další náplní práce je představení jedné z forem cloudových služeb, která je nazvaná „backend jako služba“ (BaaS), a to z důvodu, že Real-Time databáze bývají poskytovány právě touto formou. Konkrétně je pak popsána cloudová služba Google Firebase, která je využita v praktické části práce.

Cílem praktické části práce je demonstrovat probranou teorii ve formě mobilní aplikace pro platformu Android, která je napojená na Real-Time databázi poskytovanou cloudovou službou Firebase. Tato aplikace by měla být zpracována tak, aby ukázala výhody pramenící z použití Real-Time cloudové databáze. Součástí praktické části je také popis případů užití v aplikaci spolu se scénáři některých z těchto případů, popis architektury aplikace, popis jednotlivých obrazovek aplikace a popis integrace klientské a cloudové části.

Na začátku práce jsou stručně vysvětleny některé základní pojmy, které se v práci vyskytují a jejichž znalost je důležitá pro správné pochopení dalších částí práce.

První kapitola se soustředí na srovnání databázových systémů, postupně tak dojde ke shrnutí výhod a nevýhod relačních databází, NoSQL databází i jejich hybridů. Nedílnou součástí této první kapitoly je i ukázka popularity jednotlivých typů DBMS a trend jejího vývoje.

Ve druhé kapitole je plánováno představení NoSQL databází. Postupně jsou popsány jednotlivé datové formáty používané pro ukládání dat v NoSQL databázích, vyjmenovány základní principy těchto DBMS a rozebrány jejich jednotlivé typy.

Třetí kapitola se věnuje Real-Time databázím. Je zde popsána reaktivní aplikace, rozhodovací strom Real-Time dotazu, ve stručnosti jsou tak představeny populární řešení těchto databází a také bude nastíněno optimální využití Real-Time databází.

Čtvrtá kapitola se soustředí na vysvětlení principu cloudových služeb ve formě BaaS, shrnutí jednotlivých funkcí poskytovaných cloudovými službami této formy, výhody a rizik pramenící z používání BaaS, optimální postup při výběru konkrétního BaaS na začátku vývoje informačního systému a na závěr kapitoly dojde ke stručnému představení konkrétní cloudové služby ve formě BaaS.

Pátá kapitola se již zaměřuje na cloudovou službu Google Firebase, která je použita v praktické části této práce. Po řádné představení této služby nejsou opomenuty její jednotlivé části nezahrnující pouze NoSQL databázi, ale i další součásti backendu potřebné pro vývoj klientské aplikace.

V šesté kapitole dochází již k popisu praktického výstupu této práce. Nejprve jsou definována základní kritéria vyplývajících ze zadání diplomové práce a z teorie probrané v předešlých kapitolách. Následuje postup tvorby zadání pro praktickou část spolu s jeho konečnou podobou zadání. Poté jsou vyjmenovány jednotlivé případy užití demonstrační aplikace spolu s některými jejich scénáři. Nakonec je popsána architektura aplikace, zobrazeny jednotlivé obrazovky aplikace a popsána integrace klientské a cloudové části.

Základní pojmy

ACID vlastnosti

ACID (atomicita, konzistence, izolovanost, trvalost) v databázových systémech odkazuje na standardní sadu vlastností garantujících, že databázové transakce jsou spolehlivě zpracovány.

Atomicitou je myšlena garance, že celá transakce bude úspěšná nebo naopak celá selže. Je tedy zaručeno, že nikdy nebude úspěšná pouze část transakce.

Konzistence zajišťuje, že všechna data budou konzistentní. Tím je myšleno, že data budou platná podle všech definovaných pravidel včetně jakýchkoli omezení, kaskád a spouštěčů, které byly v databázi aplikovány.

Izolovanost zaručuje, že všechny transakce proběhnou nezávisle. Žádná transakce tedy nemůže být ovlivněna jinou transakcí (např. čtením dat z ní), která ještě nebyla dokončena.

Trvalostí je myšleno, že jakmile je transakce potvrzena (tzv. commitem), zůstane v systému i v případě jeho bezprostředního selhání. Jakékoli změny z transakce musí být trvale uloženy. [1]

AJAX

Asynchronní JavaScript a XML (AJAX) je termín, který vytvořil Jesse James Garrett v roce 2005. Popisuje přístup k používání řady existujících technologií společně.

Pokud jsou tyto technologie kombinovány v AJAX modelu, tak jsou webové aplikace schopné provádět rychlé přírůstkové aktualizace UI bez opětovného načtení celé stránky.[2]

Android

Základ mobilní platformy Android tvoří OS pro chytré telefony, který od roku 2005 vlastní společnost Google a vyvíjí konsorcium Open Handset Alliance tvořené mnoha technologickými a mobilními společnostmi. Systém je postavený na linuxovém jádře a je navržený tak, aby mohl běžet na různém HW. Další nedílné součásti této platformy jsou

- *distribuční kanál, který se v současnosti nazývá Google Play Store,*
- *výrobci mobilního HW s OS Android,*
- *vývojáři vytvářející mobilní aplikace pro OS Android*
- *a hlavně uživatelé mobilních zařízení této platformy.* [vlastní]

Aplikační doména

Aplikační doména je segment reality, pro který je vyvíjen softwarový systém. Je to pozadí nebo startovací bod pro analýzu aktuálního stavu a vytvoření modelu domény. [3]

Architektura klient/server

Architektura klient/server je přístup k softwaru, ve kterém jedna aplikace (klient) požaduje a přijímá služby z jiné aplikace (server), která je obsluhuje (např. když klient požaduje určitá data, tak server je přečte z databáze a odešle). Obě aplikace běží nezávisle na sobě. [4]

Autentizace

„Pod pojmem autentizace rozumíme ověření totožnosti uživatele dat, tzn. ověření toho, že je uživatel skutečně tím, za kterého se vydává. Metod autentizace je mnoho, můžeme je však rozdělit do tří základních skupin:

- *autentizace založená na určité znalosti (např. hesla),*
- *autentizace biometrická, jenž spočívá v tom, že každý jedinec má určité biologické znaky jedinečné (např. snímání sítnice oka)*
- *a autentizace prostřednictvím autentizačního předmětu, kdy se identita prokáže vlastnictvím jedinečného předmětu (např. platební karty).“*[5]

Autorizace

„Autorizace je proces, který navazuje na autentizaci a můžeme ho chápat jako přiřazení určitých práv (co uživatel může a co ne) pro práci a využívání příslušného systému.“ [6]

Hašovací tabulka

Hašovací tabulka je datová struktura používaná pro potřeby rychlého vyhledávání (podobně jako třeba binární vyhledávací strom). Lze se ji představit jako pole obsahující jednotlivé položky. [7]

Typ položek (např. těleso), který je ukládán do tabulky, musí obsahovat hašovací funkci (např. objem tělesa), pomocí které je vypočítán hašovací klíč. Pomocí něj lze přistupovat k jednotlivým položkám (případně přidávat další). Hašovací klíč si lze představit jako index položky v poli.

Linuxové jádro

Jádro operačního systému vytvořil finský student Linus Torvalds, který v roce 1991 uvolnil jeho zdrojové kódy pod licencí zakazující komerční distribuci. Následně se k vývoji začali přidávat programátoři z celého světa. Dnes linuxové jádro tvoří několik milionů řádků

zdrojového kódu a nové verze jsou uvolňovány pod licencí GNU GPL, což je licence pro svobodný software, která vyžaduje, aby byla odvozená díla dostupná pod toutéž licencí. [8]

Objektově relační mapování

Relační databáze (dále RDB) a objektově orientované programovací jazyky (dále OOPL) jsou založené na odlišných paradigmatech obsahujících technické a koncepční neslučitelnosti. Objektově relační mapování (dále ORM) zahrnuje řešení pro automatizované mapování objektů z aplikační vrstvy do relačních dat. [9] Díky použití ORM může být vývojář, který v aplikaci pracuje s daty reprezentovanými objekty, odstíněn od nutnosti pracovat s konkrétními SQL dotazy. ORM tedy v aplikaci zjednodušuje provádění běžných databázových operací jako je čtení, zápis, úprava a mazání dat.

Open source

Open Source je licence, pod kterou je vyvíjen software, který obdrží certifikaci od společnosti Open Source Initiative. Tento software musí být volně šiřitelný a poskytovat své zdrojové kódy, které je možné prohlížet, upravovat a distribuovat dále pod stejnou licencí. [10]

Režim offline

Režim offline označuje stav, kdy není zařízení, se kterým uživatel pracuje, připojené k internetu. [11] Když tedy uživatel pracuje na zařízení, které se zrovna nachází v tomto stavu, aplikace nemůže stáhnout ze serveru aktuální data, ty se tak stáhnou až ve chvíli, kdy se zařízení může znova připojit na server (dostane se do režimu online). V tu chvíli se také nahrají na server změny, které uživatel v aplikaci provedl během doby, kdy bylo zařízení v režimu offline.

SQL

SQL je standardizovaný počítačový jazyk navržený pro získávání informací z databází, který se postupně stal nejpopulárnějším dotazovacím jazykem vůbec. [12]

Škálovatelnost DBMS

„Škálovatelnost je z hlediska problematiky zpracování dat vlastnost systému (popř. sítě, procesu apod.) flexibilně reagovat na měnící se požadavky.“ [1, str. 48] V oblasti databázových serverů je tím myšlen zejména zvyšující se provoz a objemy dat. DBMS lze škálovat vertikálně nebo horizontálně. To je ostatně popsáno v [13].

Při **vertikálním škálování** se zvyšuje výkon konkrétního fyzického serveru. To se provádí například zvyšováním paměti nebo výpočetního výkonu. Tento způsob je finančně velmi náročný a nelze takto zvyšovat výkon do nekonečna.

Horizontální škálování řeší problém odlišným způsobem. Namísto neustálého vylepšování jednoho serveru dochází k distribuci dat na více vzájemně spolupracujících serverů (zapojených do tzv clusteru). Finančně je tento způsob podstatně výhodnější, protože stačí pořízení levnějších strojů. Výkon lze navíc zvyšovat do nekonečna postupným přidáváním dalších serverů.

„Špagetový" kód

„Špagetový" kód je způsob programování, který vede k nesrozumitelnému a neudržovatelnému kódu. Špagetový kód může každý vývojář vnímat jinak podle toho, na jakém levelu jsou jeho programovací schopnosti. [14]

Testování softwaru

Testování softwaru je provádění programu s úmyslem nalézt chyby. V každém softwaru mohou být různé typy chyb jako jsou např. chybná specifikace, chybný návrh, chybný příkaz, chybný vstup nebo chybný test. Aby mohly být všechny možné chyby nalezeny, existuje několik fází testování:

- Jednotkové testování – testování jednotek u OOP znamená testování jednotlivých tříd a metod. „Testy jednotek se velmi špatně aplikují na již zaběhlých projektech. Proto je vhodné zabývat se těmito testy již v etapě návrhu aplikace a v té době se rozhodnout, zda tyto testy budeme využívat." [16] Testy by měly izolovány (aby se vzájemně neovlivňovaly) a fungovat za každých okolností (např. při spuštění v jakémkoli pořadí nebo bez přístupu k internetu).
- Integrační testování – ověření bezchybné komunikace mezi jednotlivými komponentami uvnitř aplikace. Tyto testy lze u menších projektů vynechat, pokud jsou provedeny následující fáze testování.
- Systémové testování – ověření aplikace jako funkčního celku. *„Podle připravených scénářů se simulují různé kroky, které v praxi mohou nastat. Obvykle probíhají v několika kolech. Nalezené chyby jsou opraveny a v dalších kolech jsou tyto opravy opět otestovány."*[16] Tato úroveň testování je poslední před předáním softwaru zákazníkovi.
- Akceptační testování – testování probíhající na straně zákazníka. *„Testy jsou prováděny podle připravených scénářů a nalezené nesrovnalosti mezi aplikací a specifikací jsou reportovány zpět vývojovému týmu. Opravené chyby jsou nasazeny na prostředí u zákazníka."*[16]

Zároveň platí, že v čím dřívější fázi jsou chyby nalezeny, tím méně prostředků stojí jejich oprava. Testování může být jak manuální, tak automatizované. Čerpáno z [15, 16].

Zdrojový kód

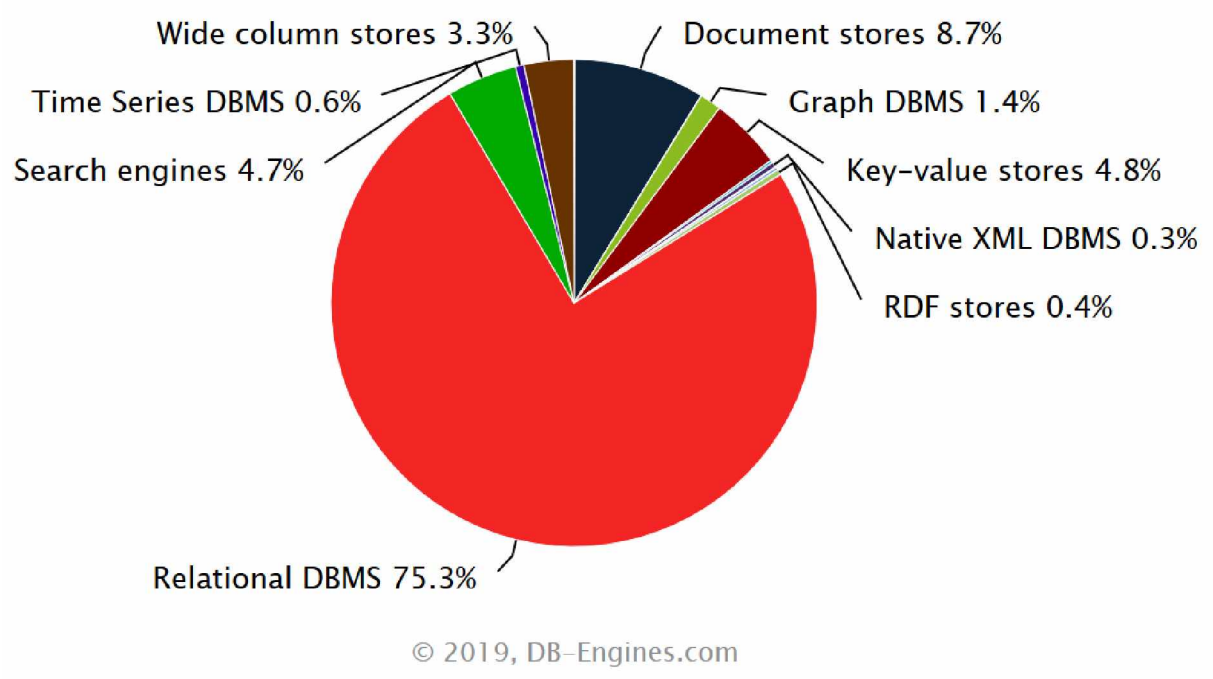
Zdrojovým kódem je popis programu pomocí programovacího jazyka, který lze přeložit do strojového kódu počítače a spustit. [17]

1 Srovnání databázových systémů

V minulých desetiletích postupně vznikla celá řada typů databázových systémů. Nejpoužívanějším z nich se bezesporu stal systém **relační**. Většinu ostatních lze s odstupem času vnímat spíše jako módní trendy, které se neprosadily. [18, str. 22-23] *„Např. objektové databáze začaly vznikat jako přirozenější uložiště pro struktury objektového programování. V dnešní době sice objektové programovací paradigma stále dominuje, ale využívanost objektových databází je oproti relačním podstatně nižší.“* [18, str. 23]

Na počátku tisíciletí začaly vznikat nové DBMS, které se souhrnně označují jako **NoSQL**. Cílem těchto DBMS nikdy nebylo nahrazení tradičních databázových systémů. Tyto systémy byly naopak nabízeny jako řešení pro nové typy aplikací, jimž relační DBMS nevyhovovaly.

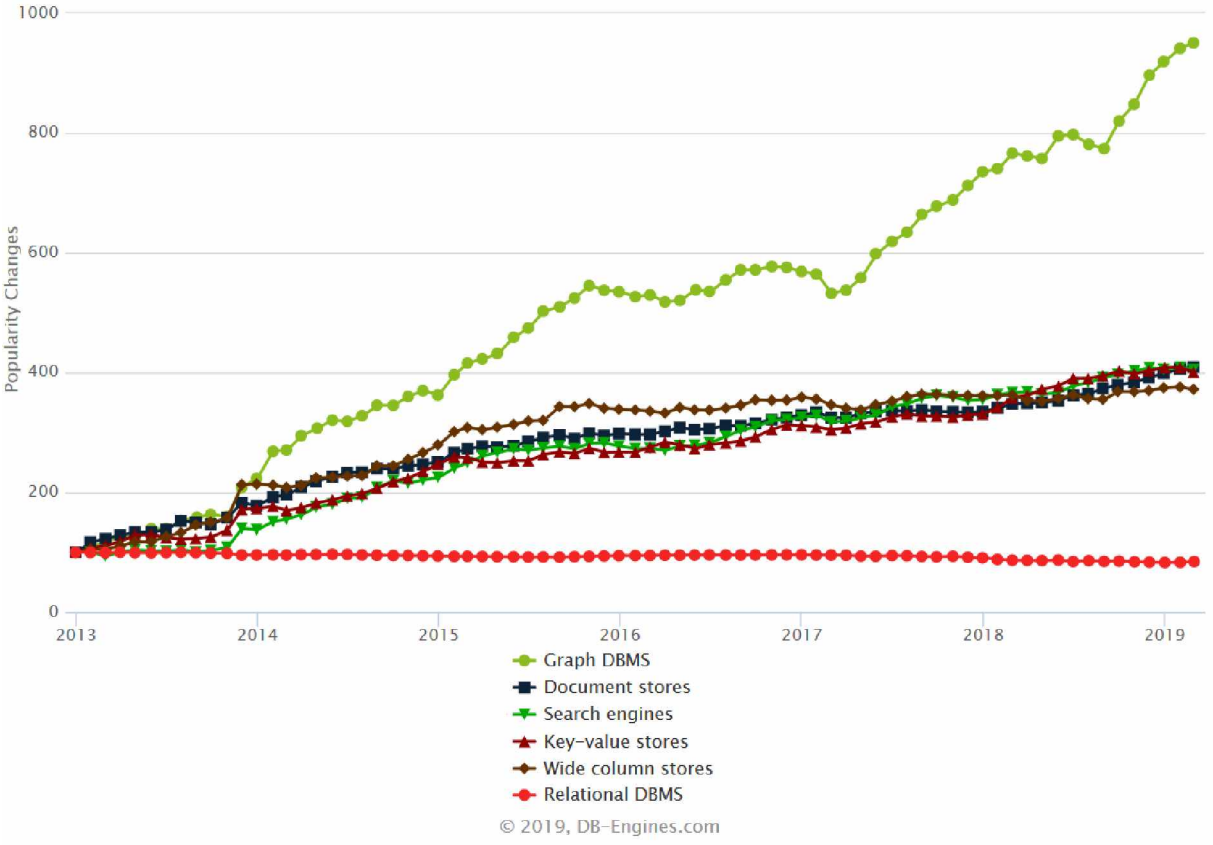
V dnešní době jsou již široce používány a postupně se stávají rovnocennou alternací k tradičním relačním databázím. [18, str. 23-25] Jak ukazuje graf popularity **DB-Engines** (obr. 1), jednotlivé typy NoSQL databází mají v součtu 22,9%, DBMS mimo relační a NoSQL již mají téměř nulovou popularitu.



Obrázek 1: Popularita kategorií DBMS, březen 2019 [1]

[1] zdroj: https://db-engines.com/en/ranking_categories

Další graf zobrazující trend popularity relačních a NoSQL DBMS (obr. 2) ukazuje, že popularita NoSQL databází od roku 2013 výrazně roste oproti relačním databázím, kde popularita spíš stagnuje.



Obrázek 2: Trend popularity kategorií DBMS [2]

V následujících sekcích této kapitoly je provedeno srovnání relačních a NoSQL databázových systémů.

Relační

„Relační model je založen na matematickém konceptu ***relace****, která je fyzicky reprezentovaná tabulkou.“* [19, str. 63] Ta je používána k uložení informací o objektech reálného světa, které mají být v databázi přítomny. Plnohodnotný výzkum probíhá od konce 70. let minulého století. Díky tomu dosahují databáze léty prověřené **robustnosti**. [19, str. 62]

„Relační databáze vychází z předpokladu, že dobře známe strukturu ukládaných dat, a tato data jsou často klienty a aplikacemi ***dotazována různými i předem neznámými způsoby****. Proto jsou datové struktury rozděleny na* ***co nejmenší kompaktní celky****, každý je uložen do samostatné tabulky a* ***odpověď na dotaz je pak sestavena z obsahu těchto tabulek****. Relační*

[2] zdroj: https://db-engines.com/en/ranking_categories

*databáze jsou také známé tím, že plně implementují transakční zpracování dotazů (vlastnosti ACID) s **nejvyšší úrovní izolace transakcí.**"* [18, str. 88]

Efektivní implementace sestavování odpovědí a transakční zpracování s vlastnostmi ACID jsou tedy důvodem, proč relační databáze **nebývají plně distribuované** a tedy horizontálně škálovatelné. [1, str. 88] Jediná možnost je tedy **vertikální škálování**, které je ale **finančně náročné a nelze provádět do nekonečna**. Kvůli nutnosti dodržení ACID vlastností je tak omezena **propustnost** databáze. [18, str. 23-24]

Zásadní pro relační databáze je **integrita dat**. To znamená, že data jsou ověřována ve všech tabulkách a do systému nejsou vloženy žádné duplicitní, nesouvisející nebo neoprávněné údaje. [20]

Předpokladem většinou je **dlouhodobé uložení dat** a jejich **časté aktualizace**. **Velikost dat** by měla narůstat **lineárně**. Probíhají **pravidelné zálohy dat** a **přístup k datům** zajišťuje **jediný server**. [18, str. 26]

Hned na začátku byl vyvinut strukturovaný dotazovací jazyk SQL, který se stal **formálním standardem** a postupně i standardním jazykem relačních DBMS. [2, str 62] Tento jazyk je **skvělý** obzvláště **při potřebě použití komplexních dotazů**, na druhou stanu vyžaduje **předdefinování schématu** pro určení struktury dat před prací s nimi. **Všechna data v tabulce** pak **musí tuto strukturu následovat**. [21]

NoSQL

Jde o novou generaci databázových systémů určenou pro správu velkého množství dat, které **exponenciálně roste**. NoSQL databáze vybízejí k tvorbě struktur, které budou nejlépe odpovídat často pokládaným dotazům. Ty pak bývají **velmi rychle zpracovávány** a systém jich je schopen **zvládat obrovské množství** za jednotku času. [18, str. 23-24]

*„NoSQL databáze tak poskytují specializovaná uložiště pro konkrétní typy dat, umožňují **seskupovaní různých záznamů**, na které se bude poté většinou přistupovat společně, a také **replikaci dat na více uzlech.**"* [18, str. 88]

Dalším znakem je **flexibilní datové model**. NoSQL databáze nemusí mít určené žádné databázové schéma nebo toto schéma může být proměnlivé. *„Dodržování tohoto schématu je často ponecháno na aplikaci a jeho změna neznamená pro databázi významnou zátěž."* [18, str. 23] Úroveň flexibility je pro různé typy NoSQL databází odlišná. [18, str. 88]

Hlavní výhodou NoSQL databází je možnost **horizontálního škálování** s čímž souvisí **finanční nenáročnost** a **efektivita** této operace. [3] Zpracování každé úlohy pak může být **distribuováno** v rámci množiny uzlů clusteru, kterou je možné podle potřeby rozšiřovat. K tomu se ale váže i nevýhoda v podobě **složitosti instalace** DBMS v rámci celého clusteru a následné **údržby** systému. [18, str. 23-24]

Vzhledem k tomu, že jsou RDBMS vyvíjeny násobně déle, systémy NoSQL stále ještě **nedosahují takové robustnosti**, ve většině případů neobsahují tolik funkcí a nejsou tak stabilní jako systémy relační. [22]

NoSQL DBMS dále nenabízejí prakticky **žádné standardy**. Prakticky každý DBMS si vyvinul svůj dotazovací jazyk a jeho složitější použití často vyžaduje netriviální programátorské schopnosti. [18, str. 24]

Velkou nevýhodou je také stále **nedostatek expertů** v oblasti NoSQL databází na trhu práce. Tato situace se bude postupně zlepšovat, ale v současné době je poměrně složité nalézt v této oblasti zkušené lidi. [22]

Hybridní

Kromě klasických NoSQL databází existují také tzv. **non-core** (nepravé) NoSQL databáze. Mezi ně jsou řazeny DBMS obsahující alespoň některé principy NoSQL databází. Jako hybridní databázi lze označit např. systémy, které dokáží pracovat s daty v relačním i dokumentovaném datovém modelu, nebo systémy používající různé dotazovací prostředky nad jedním datovým uložištěm. [18, str. 243]

První skupinou jsou tradiční databáze (relační nebo nativní XML), které postupně implementovaly **některé principy** NoSQL databází. Tyto databáze např. umožňují **ukládat data ve formátu JSON**, díky čemuž se zvyšuje efektivita (není potřeba konverze dat). Tyto DBMS jsou ale pouze **vertikálně škálovatelné**. [18, str. 244]

Další skupinou jsou NoSQL **databáze ve webovém prohlížeči**, které používají webové aplikace k uložení některých konfigurací a neuložených informací na straně klienta. Nejznámější takovou databází jsou samozřejmě **cookies**, které ale mají omezenou kapacitu. Aplikace, které potřebují ukládat více dat na klientské straně používají obvykle **Web Storage**. [18, str. 250]

Poslední skupinou, která bude v této práci zmíněna, jsou tzv. **NewSQL databáze**. Tyto databáze jsou **horizontálně škálovatelné** a přitom používají **osvědčený relační model dat** a vlastnosti ACID. Tyto databáze jsou používány jednak v případě, kdy aplikace již pracuje s relačním modelem a potřebuje řešit náhlý nárust dat, jednak v případě, kdy se aplikace nemůže ze své podstaty vzdát požadavku na silnou konzistenci dat. [18. str. 254]

Závěr

Každý z těchto typů databázových systémů má své výhody i nevýhody. Nelze tedy říct, že by byl některý z nich lepší nebo horší a výběr správného typu DBMS pro danou aplikaci by měl vždy záležet na potřebách samotné aplikace, charakteru ukládaných dat a přístupu k těmto datům.

2 NoSQL databáze

Během posledních let se v některých oblastech poměrně výrazně změnily požadavky na databázové systémy, což vedlo ke vzniku nových DBMS, které se rychle staly hojně využívané. Tyto systémy jsou unikátní svou nabídkou funkcí, vnitřním fungováním a efektivitou zpracování vybraných typů dotazů. [18, str. 87]

Cílem této kapitoly je poskytnutí znalostí nezbytných pro pochopení fungování NoSQL databází.

2.1 Datové formáty

„Na rozdíl od relačních databází, které staví nad abstraktním relačním modelem, u NoSQL databází často formální datový model neexistuje nebo je tak jednoduchý, že vnitřní strukturu ukládaných dat řídí sama aplikace. Typicky se k tomu používá některý existující široce rozšířený formát.“ [18, str. 29]

Také pro návrh aplikací, které vyžívají NoSQL databáze, je důležitá znalost těchto datových formátů. V dnešní době je většina aplikací postavena na architektuře klient/server, kdy komunikace mezi klientem a serverem probíhá v přesně definovaném formátu, který je ideálně shodný s formátem pro ukládání, aby se omezila režie potřebná pro konverzi dat. [18, str. 29]

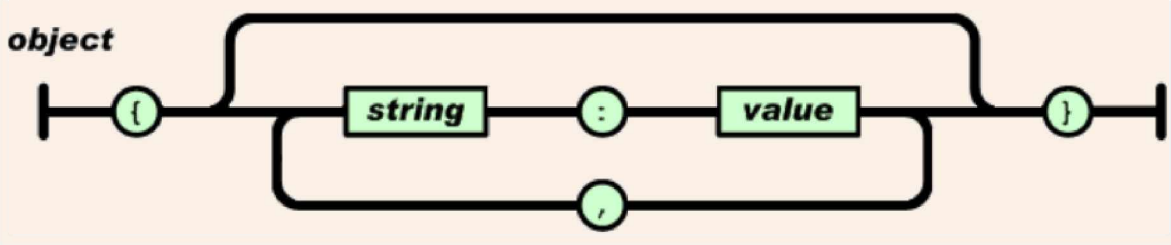
V následujících sekcích jsou rozebrány některé formáty používané v NoSQL databázích.

JSON

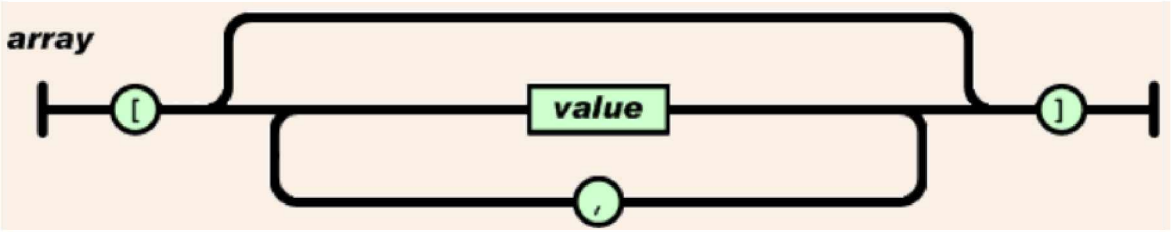
JSON je jednoduchý textový formát pro výměnu dat, který je založen na podmnožině skriptovacího jazyka JavaScript. [23] *„Postupem času se však JSON stal rozšířeným datovým formátem a knihovny umožňující jeho použití existují pro všechny běžné používané programovací jazyky.“* [18, str. 30] Tento formát je jednoduše čitelný a zapisovatelný pro člověka. Kromě toho se také snadno strojově analyzuje a generuje. [23]

„JSON je založen na dvou strukturách:

- *Kolekce párů název/hodnota. Ta bývá v rozličných jazycích realizována jako objekt, záznam (record), struktura (struct), slovník (dictionary), hash tabulka, klíčový seznam (keyed list) nebo asociativní pole* (pozn. autora: syntaxe zobrazena na obr. 3).
- *Seřazený seznam hodnot. Ten je ve většině jazyků realizován jako pole, vektor, seznam (list) nebo posloupnost (sequence)* (pozn. autora: syntaxe zobrazena na obr. 4).“ [23]



Obrázek 3: Syntaxe JSON objektu [3]



Obrázek 4: Syntaxe JSON pole [4]

V těchto strukturách umožňuje reprezentovat čtyři jednoduché datové typy (řetězec, číslo, logická hodnota, null). *„Pole a objekty mohou jako své prvky obsahovat libovolný další datový typ – jednoduchý i strukturovaný. JSON je tak velmi flexibilní a lze v něm reprezentovat v podstatě jakoukoliv datovou strukturu.“* [18, str. 30] Přílišná jednoduchost bývá tomuto formátu někdy vyčítána. Jeho syntaxe totiž nepodporuje ani komentáře a nelze tak okomentovat složitější JSON dokumenty. [18, str. 32]

JSON dokumenty jako základní formát pro reprezentaci dat zvyšují popularitu NoSQL dokumentovaných databází. Jedním z důvodů pro takovou popularitu je jejich schopnost držet velké objemy dat při absenci explicitního datového schématu. Při úlohách, kde je schéma dat nezbytné (např. vyhledávání), pak bývá použito JSON Schema. [24]

XML

„Značkovací jazyk XML vznikl jako zjednodušení komplexního značkovacího jazyka SGML. Původně se SGML a XML používalo zejména v oblasti elektronického publikování pro reprezentování rozsáhlých dokumentů s pevnou strukturou a potřebou lepšího propojení pomocí odkazů a prohledávání. Typickou oblastí nasazení XML tak byla například reprezentace právních textů (zákony, precedenty, smlouvy) nebo technická dokumentace.“ [18, str. 35]

[3] zdroj: https://www.json.org/json-cz.html
[4] zdroj: https://www.json.org/json-cz.html

V době vzniku XML byla velká poptávka také po formátu umožňujícím snadné propojení systémů a výměnu dat mezi nimi. XML se postupně stalo nejpoužívanějším formátem pro výměnu dat a vznikly a i složitější nadstavby, jako jsou webové služby (formát SOAP nebo WSDL), které samotné XML doplnily o další vrstvy. Před nástupem JSON formátu se právě formát XML používal v AJAXových aplikacích pro zasílání dat ze serveru na klienta.

Základem XML jsou elementy, do kterých mohou být uzavírány hodnoty nebo další elementy. Samotné XML nemá datové typy, veškeré hodnoty se chápou jako text. XML se však běžně používá v kombinaci s XML schématy, které kromě struktury dat opisují také datové typy jednotlivých elementů. Dokumenty mohou obsahovat i komentáře.

V této sekci bylo čerpáno z [18, str. 35-36].

BSON

BSON je binárně kódovaná serializace JSON dokumentů. Stejně jako JSON podporuje vkládání objektů a polí od dalších objektů a polí. Kromě toho také obsahuje rozšíření, které povoluje reprezentaci datových typů, které nejsou součástí JSON specifikace (BSON obsahuje např. datové typy Date a BinData).

BSON lze porovnat s dalšími binárními formáty pro výměnu dat jako Protocol Buffers. Na rozdíl od něj nemá explicitně definované schéma, což mu dává výhodu větší flexibility ale i mírnou nevýhodu v prostorové efektivitě (BSON má větší režijní náklady pro názvy uvnitř serializovaných dat).

BSON byl navrhnut pro tři následující charakteristiky:

- Odlehčenost znamenající držení prostorových režijních nákladů na minimu, což je důležité pro jakýkoliv formát datové reprezentace, speciálně pro formát používaný přes síť.
- Průchodnost, která značí, že lze tento data v tomto formátu snadno procházet. To je extrémně důležitá vlastnost v jeho roli primární datové reprezentace pro NoSQL dokumentovanou databází MongoDB.
- Efektivnost umožňující provést zakódování do BSONu a dekódování z tohoto formátu velmi rychle ve většině programovacích jazyků v důsledku použitých datových typů programovacího jazyka C.

V této sekci bylo čerpáno z [25].

Další binární formáty

Kromě formátu BSON se běžně používají i další binární formáty. Zejména v databázích klíč-hodnota, u kterých je důležitá rychlost a kompaktnost ukládaných dat, najdou své uplatnění např. **Protocol Buffers** od Google či **Apache Cordova**, který je původně od Facebooku.

2.2 Základní principy

Jak již bylo popsáno v minulé kapitole, existují určité principy, kterými se NoSQL DBMS odlišují od tradičních DBMS. Nejdůležitější z těchto principů jsou dále detailněji rozebrány.

Škálovatelnost

S rostoucím rozmachem internetu po celém světě je spojen rostoucí objem dat a uživatelů, s čímž je spojena potřeba neustálého zvyšování výkonu HW. To je na rozdíl od tradičních DBMS řešeno u NoSQL databází pomocí **horizontálního škálování**.

I když se tento způsob škálování zdá na první pohled výhodnější, existují i některé nevýhody:

1) Síť není stoprocentně spolehlivá a hrozí její výpadky.
2) Síť obsahuje určitou latenci, se kterou je potřeba počítat.
3) Síť má omezenou propustnost.
4) Komunikace po sítí není stoprocentně bezpečná.
5) Občas je potřeba změna topologie sítě.
6) Síť většinou spravuje více administrátorů.
7) Přenos dat po síti stojí určité náklady
8) Síť bývá heterogenní.

Některé z těchto nevýhod lze omezit poměrně jednoduše, u jiných je to nemožné. [18, str. 48-49]

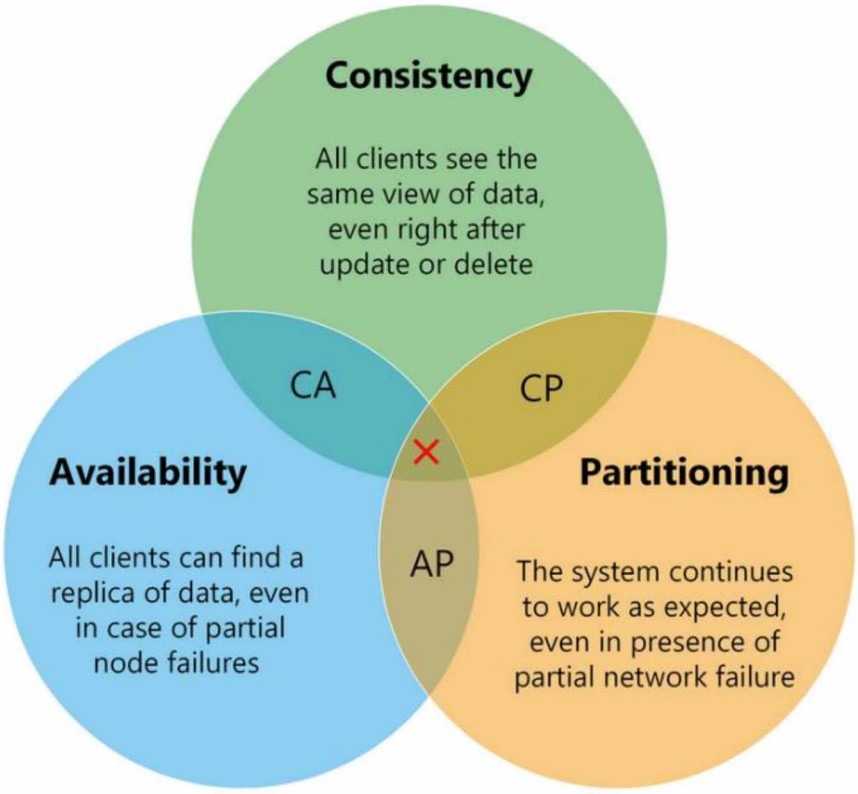
CAP teorém

V distribuovaných DBMS jsou používané tzv. **ideální vlastnosti distribuovaného systému**:

- **C: Konzistence** (**c**onsitency) označuje stav, kdy jsou všechny kopie dat konzistentní a aktuální.
- **A: Dostupnost** (**a**vailability) značí možnou obsluhu všech možných dotazů.
- **P: Odolnost vůči rozpadu sítě** (tolerance to network **p**artitions) znamená, že DBMS zůstává nadále funkční i v případě výpadku sítě mezi jednotlivými uzly. [26]

Všechny tři vlastnosti žádoucí, jak ale říká CAP teorém, databáze může splňovat maximálně dvě z nich (vizualizace tohoto tvrzení je zobrazena na obr. 5). [26]

V praxi bývá často u jednotlivých typů DBMS uváděno, které vlastnosti splňují. Např. RDBMS systémy splňují dostupnost a konzistenci. [26] „Pokud však uvažujeme distribuovaný systém, potřebujeme především odolnost vůči rozpadu sítě a tím pádem nám nezbývá, než do určité míry omezit konzistenci a/nebo dostupnost.“ [18, str. 53]



Obrázek 5: Vizualizace CAP teorému [5]

CAP teorému je vytýkáno především to, že některé vlastnosti nelze úplně porovnávat, protože např. s chybějící konzistencí se musí počítat po celou dobu fungování systému, zatímco chybějící dostupnost může nastat pouze v případě výpadku sítě, který nastává pouze v určitém okamžiku. [18, str. 53]

Konzistence

Ve světě NoSQL DBMS nelze z důvodů distribuce a výpadků sítě zajistit ACID vlastnosti (bez významného zpomalení systému), tak jak jsou známé z relačních DBMS. Alternativou ACID v distribuovaných systémech jsou tzv. BASE vlastnosti. [18. str. 52-54]

- **BA: Převážná dostupnost** (**b**asic **a**vailability) znamená, že v případě výpadku části distribuovaného systému zůstanou zbylé části systému nadále funkční.

[5] zdroj: https://www.researchgate.net/figure/Visualization-of-CAP-theorem_fig2_282679529

- **S: Volný stav** (**s**oft state) značí, že data mohou být přepsána jejich novější verzí, což souvisí s poslední vlastností.
- **E: Občasná konzistence** (**e**ventual consistency) označuje fakt, že databáze se v určité chvíli může nacházet v nekonzistentním stavu. [27].

Občasná konzistence je jakousi alternativou ke klasické silné konzistenci známé z RDBMS. Může nastat zejména v případě, kdy jsou data umístěna na několika serverech v rámci clusteru. *„Provede-li uživatel nebo aplikace aktualizaci dat na jednom ze serverů, jsou ostatní kopie dat po určitou (zpravidla krátkou) dobu nekonzistentní, a to tak dlouho, dokud replikační mechanismus NoSQL databáze neprovede aktualizaci všech kopií dat.* [28]

Distribuce

Protože v některých typech aplikacích využívajících NoSQL databáze bývají data takového rozsahu, že je není možné zpracovat v rámci jediného databázového serveru, využívají a kombinují se dvě ortogonální techniky distribuce dat. [18, str. 56]

První z nich spočívá v **rozdělení dat** (sharding) na vhodné podmnožiny (ne nutně disjunktní), které se uloží na různé uzly clusteru. Uživatel by měl ideálně přistupovat na co nejméně uzlů a fyzicky by data měla být uložena vzhledem ke geografické příslušnosti uživatelů, kteří k nim mají přistupovat. [18, str. 57] Tato technika je vhodná v případě, kdy různí uživatelé přistupují k různým částem dat. [27]

Rozdělení dat sice zvyšuje výkon DBMS, při výpadku některého uzlu ale hrozí, že uživatel nebude mít přístup k některým částem dat. Proto bývá rozdělení dat kombinováno s **replikací**. V takovém případě je určitá podmnožina dat replikovaná na jiné servery. Existují dva typy replikací:

- **Master-slave replikace** funguje na principu, že jeden uzel je určen jako primární (master) a zbylé uzly jsou sekundární (slaves). Pro čtení lze v takovém případě využít jakýkoliv uzel, požadavky na zápis jsou však vždy předány na primární uzel, který změny provede a následně je propaguje do sekundárních uzlů.
- **Peer-to-peer replikace** na rozdíl od předchozí nefunguje centralizovaně. Zde jsou si všechny uzly rovny a mohou obsluhovat čtení i zápis dat. V případě výpadku některého z uzlu tedy není ztracena možnost čtení ani zápisu. Na druhou stranu vzniká problém s udržením konzistence dat, kdy mohou vznikat konflikty při zápisu dat. [18, str. 58-59]

Master-slave replikace je výhodná v případě, kdy je většina požadavků na čtení dat a k modifikaci dat dochází minimálně. Je-li však požadavků na zápis více, databázi nelze výrazně škálovat a tato strategie není moc efektivní. [18, str. 58]

V takovém případě je potřeba použít peer-to-peer replikaci. Problém s udržením konzistence lze řešit buď vzájemnou koordinací uzlů, což ale může výrazně zpomalovat systém, nebo prostřednictvím volebního kvóra, kdy je stanoven minimální počet uzlů, na kterých musí zápis dat proběhnout, aby byl označen jako platný. [18, str. 59]

Jak již bylo uvedeno výše, v praxi se běžně kombinuje rozdělení dat s replikací. Kombinace master-slave replikace s rozdělením dat bývá provedena tak, že v clusteru je více primárních uzlů, ale pro konkrétní data existuje vždy použit pouze jeden. Při kombinaci peer-to peer replikace s rozdělením dat je pak každá část dat uložena na určitém počtu uzlů v clusteru, přičemž části dat na jednotlivých uzlech se liší. [18, str. 60]

2.3 Typy NoSQL databází

V předchozích podkapitolách byly vysvětleny základy pro pochopení fungování NoSQL databází. Tato podkapitola se již věnuje klasifikaci jednotlivých NoSQL databází tak, jak je běžně uváděna v literatuře, která se zabývá NoSQL DBMS. Kromě popisu jednotlivých typů databází jsou zde uvedeny jejich konkrétní příklady.

Klíč-hodnota

Jedná se o nejjednodušší formu NoSQL databáze. Jde v podstatě pouze o **datovou strukturu hašovací tabulka** a veškerá logika je ponechána na klientské aplikaci. Jediné podporované operace bývají pouze nastavení hodnoty pro daný klíč, čtení hodnoty pod daným klíčem a smazání hodnoty s klíčem. Tento typ DBMS vyniká nad ostatními především svou vysokou škálovatelností. Pro optimalizaci rychlosti bývají všechna data v těchto databázích ukládány jako binární objekty. [18, str. 96] [26] [27]

Databáze klíč-hodnota neobsahují žádné tabulky ani nějakou jejich obdobu. Ve většině těchto DBMS však lze vytvářet několik jmenných prostorů v rámci jedné databáze. V případě kombinace s vhodným pojmenováním lze vytvořit struktury, které lze vzdáleně připodobnit k tabulkám v RDBMS. [18, str. 97] [26]

Nejdůležitější vlastnost databází typu klíč-hodnota plyne už ze samotné použité datové struktury. V těchto DBMS lze k datům přistupovat pouze prostřednictvím klíče a není možné data vyhledávat podle hodnoty. [26]

Kromě použití jako persistentní distribuované uložiště (**Redis** a **Amazon Dynamo DB**) se tento typ uložiště totiž používá také např. jako cache (**Memcached**). [18, str. 99]

Dokumentové

Pro dokumentové databáze je klíčový koncept dokumentu jako datové struktury, která má samopopisný charakter (obsahuje data i metadata). Jako formáty se běžně používají JSON, jeho binární reprezentace BSON nebo XML. Všechny tyto formáty jsou vlastně hierarchické datové struktury obsahující asociativní pole a seznamy.

Dokumenty jsou zde kromě ukládání používány také ke komunikaci s klienty. To umožňuje vyhnout se častým konverzím (kromě ORM také zpět na JSON). Zde lze v ideálním případě použít uložená data přímo ke komunikaci s ostatními komponentami.

Velká výhoda spočívá v tom, že dokumentové formáty daleko více odpovídají struktuře tříd objektového programování, což usnadňuje konverzi. V případě dokumentových databází se tato konverze nazývá objektově dokumentové mapování (ODM).

Další výhodou je flexibilní datový model, kdy lze ukládat různorodé dokumenty společně bez ohledu na to, jak se jejich struktura liší či postupně v čase vyvíjí. Některé databáze neposkytují žádný způsob, jak vynutit konkrétní schéma pro dokumenty v kolekci. V praxi ale aplikace komunikují s databází prostřednictvím knihoven, jež jisté schéma často udržují.

Odkazování probíhá v dokumentové databázi dvěmi způsoby:

- **Vnořené dokumenty** mají výhodu rychlého přístupu, ale velikost dokumentu může při velkém množství vnořených dokumentů vzrůst na únosnou mez, což může negativně ovlivnit např. rychlost čtení.
- **Odkazy** se používají dvěmi způsoby. V databázích, které obsahují primární klíče, se odkazuje pomocí nich a operace funguje podobně jako v databázi klíč-hodnota. Dalším způsobem bývá v některých databázích zadání cesty k odkazovanému dokumentu, což má za následek neustále traverzování dokumentem a např. rychlost čtení poté výrazně klesá.

Na rozdíl od databáze klíč-hodnota umožňuje dokumentová databáze dotazování nad atributy v rámci hierarchické struktury, což je umožněno díky automatickému indexování obsahu každého dokumentu při vložení do databáze. Vyhledávání v databázi probíhá nezávisle na typu atributu. [26]

Mezi nejznámější představitele dokumentových databází patří **MongoDB**, **Amazon DynamoDB**, **Couchbase**, **Microsoft Azure Cosmos DB, CouchDB, MarkLogic** a **Firebase Real-Time Database**.

V této sekci bylo z větší části čerpáno z [18, str. 109-112].

Sloupcové

Všechny sloupcové databáze systémy se většinou odvolávají na datový model představený společností Google ve svém vlastním distribuovaném databázovém systému **Bigtable**.

Základní stavební prvkem tohoto datového modelu je **řádek** (row) identifikovaný unikátním klíčem (row key), kdy každý takovýto řádek může mít rozdílný počet sloupců a i jejich názvy se mohou lišit. **Sloupec** je definován svým jménem (column name), hodnotou (column value) a časovým razítkem (timestamp).

Sloupce se slučují do **skupin sloupců** (column families) vytvářených během návrhu schématu databáze, přičemž každý řádek může v rámci skupiny sloupců vytvářet libovolné množství nových sloupců, které v ostatních řádcích nejsou přítomny. Jeden klíč řádku pak lze použit napříč různými skupinami sloupců, čímž vznikne **logický datový celek**. Sloupce ve skupinách sloupců bývají stejného datového typu, protože je předpokládáno, že se budou volat společně, čímž je se skupinami sloupců dosahováno největší efektivity.

Na skupinu sloupců se lze dívat jako na relační tabulku s hodnotami null, což jsou chybějící záznamy v jednotlivých sloupcích (v relačních databázích je nutná existence všech sloupců v každém řádku).

Mimo jednoduché sloupce je možné v některých databázových systémech vytvářet i tzv. **supersloupce**, jejichž hodnota je složena z podsloupců. Klasickým příkladem supersloupce může být adresa složená z města, ulice, PSČ atd.

Motivace k použití sloupcových databází namísto klasických relačních se tedy skrývá v možnosti libovolného přidávání sloupců a ve schopnosti databáze efektivně pracovat nad takovýmto schématem dat. Použitím skupin sloupců probíhá sdělování, ke kterým hodnotám sloupců se bude přistupovat společně, načež databázový systém pak může sloupce z jedné rodiny vždy distribuovat na stejný uzel, díky čemuž lze zefektivnit distribuci dat.

Nejpopulárnějším databázovým systémem tohoto druhu je beze sporu **Apache Casandra** vytvořena původně pouze interně ve společnosti Facebook. Cassandra je v současnosti open source projektem, díky čemuž je volně k dispozici.

Grafové

Poslední typ NoSQL databáze pracuje s datovou strukturou **graf**, ve které je množina uzlů propojená hranami. **Uzel** představuje entitu, která může mít libovolný počet atributů. **Hrany** pak představují vztahy mezi jednotlivými entitami a mohou mít orientaci, typ, dobu platnosti vztahu, podmínky platnosti vztahu apod. Entit a hran může být v databázi libovolné množství (pokud postačuje paměť).

Grafové databáze se objevovaly už v 80. letech, moderní grafové databáze jsou tedy spíše znovuobjevením a rozšířením staršího nápadu. Proč tedy vlastně vytvářet speciální databázový typ pro grafové struktury? Relační databáze sice také umožňují uložení grafové struktury dat, nýbrž průchod grafem (základní operace nad grafem) je posléze značně neefektivní (jedná se o velké množství operací spojení tabulek). Tuto nevýhodu lze odstranit množinou indexů, ale to zase přináší nevýhody, které s indexy souvisí (primárně se jedná o pomalé uložení dat).

Hlavním cílem grafových databází je naopak co nejefektivnější uložení dat kvůli efektivnímu provádění obvyklých operací nad grafem. Vyšší rychlost uložení dat je zajištěna tím, že každý uzel je uložen přímo do sousedních uzlů a vazeb. Dále mají grafové databáze implementována **obdobná pravidla jako ACID**, kdy při vytváření vazby mezi dvěma entitami je zapotřebí spolu s tím také aktualizovat obě entity o informaci o této vazbě. Pokud se některá z těchto operací provede neúspěšně, k vytvoření vazby nedojde.

Existuje hned několik typů grafových databází:

- Orientované,
- neorientované,
- ohodnocené,
- neohodnocené,
- multigrafové (umožňují vytváření více vztahů mezi dvěmi entitami)
- a kombinace předešlých typů.

Jako příklad grafových databází lze uvést **Neo4j**. Jedná se databázi implementovanou v jazyce Java (což zajišťuje multiplatformnost) poskytovanou pod open source licencí. Má silnou vývojářskou komunitu, což přispívá k aktivnímu vývoji databáze. V kontextu NoSQL databází je důležité zmínit, že má tento databázový systém omezenou škálovatelnost, protože nedokáže graf distribuovat. [18, str. 143-146]

3 Real-Time databáze

Real-Time databáze mění způsob, jakým DBMS poskytuje data klientovi. Na rozdíl od tradičních DBMS, které poskytují konzistentní obraz aplikační domény, Real-Time databáze počítají s tím, že se data v průběhu času mění a že systém by měl informace, které se stanou nepravdivé, aktualizovat. [28, str. 21]

V této kapitole jsou nejprve představeny Real-Time databáze, následně je rozebráno, jak vypadá obecný Real-Time dotaz a nakonec jsou popsány populární řešení tohoto typu DBMS.

Informace níže obsažené platí pro aplikace, který mají měkčí časové limity a rozhodně nejsou adresovány bezpečnostně kritickým aplikacím nebo aplikacím se striktním časovým limitem jako jsou např. systémy řídící nukleární elektrárny.

3.1 Představení

V minulosti byl termín „Real-Time databáze“ používán jako odkaz na specializované pull-based DBMS, které produkují výstup uvnitř striktních časových intervalů. V této práci Real-Time databáze představují systémy, které nabízejí push-based přístup do databáze. [28, str. 21] Tento přístup vyhovuje tzv. reaktivním aplikacím, které potřebují držet kritické informace aktuální. Pro vysvětlení je potřeba nejprve vysvětlit rozdíl mezi reaktivními aplikacemi a pull-based aplikacemi. [29]

Reaktivní aplikace vs. pull-based aplikace

V posledních letech přicházejí lidé s očekáváním reaktivního chování aplikací, tj. předpokládají, že se změny provedené jinými uživateli okamžitě promítnou také v jejich UI. Nicméně postavení takové aplikace s tradičními databázovými technologiemi je složité, protože DBMS bývají po dekády vyvíjeny čistě kolem pull-based request-response přístupového vzoru.

V souvislosti s potřebou reaktivity na straně DBMS začaly vznikat nové databázové systémy, které jsou čistě push orientovaně a slibují jednoduchý vývoj reaktivních aplikací. Tyto systémy jsou označovány jako Real-Time databáze, protože drží data v klientovi synchronizovaná v „reálném“ čase (viz další sekce). Pokud jsou data, která jsou právě zobrazena v klientském UI, aktualizována v databázi, jsou tyto změny okamžitě promítnuty také do UI klienta. Takovéto proaktivní poskytování změn snižuje složitost a usnadňuje vývoj aplikace.

V této sekci bylo čerpáno z [29].

Synchronizace v „reálném“ čase

Jak již bylo vysvětleno na začátku této kapitoly, Real-Time databáze se v kontextu této práce nevztahují k striktním časovým omezením. Synchronizaci dat v reálném čase lze tedy spíše chápat jako vnímání chování aplikace jejím uživatelem (data se aktualizují sama bez jeho vyžádání v řádech milisekund).

Samozřejmě v případě, že uživatel aplikace provádí také aktualizaci dat, je složitější mu navodit pocit synchronizace v reálném čase. Tradičně se totiž při aktualizaci dat klientem nejprve data odešlou na server, kde se změny zapíší do databáze a až poté se promítnou v UI klienta, který data aktualizoval.

Některé Real-Time DBMS (např. Meteor) však toto obcházejí pomocí spekulativního chování, tzv. **optimistického UI**. V tomto případě jsou změny paralelně s odesláním na server zapsány do klientské cache a zároveň zobrazeny v UI klienta, který modifikaci provádí. Pokud se tyto změny z nějakého důvodu nepodaří synchronizovat se serverovou databází tak, jak bylo předpokládáno (toto může nastat např. pokud jsou data aktualizována v offline režimu a změny jsou synchronizovány na server po obnovení připojení), jsou v klientské cache a UI opraveny chyby.

V této sekci bylo čerpáno z [30].

Optimální využití

Není překvapením, že pouze určité typy aplikací dokáží naplno využít výhod Real-Time databází. Jsou to hlavně reaktivní aplikace postavené na aktuálnosti poskytovaných informací.

První skupinou jsou aplikace určené ke **sledování aktuálního stavu** daných událostí. Velmi populárními jsou aplikace **přinášející aktuální sportovní výsledky**. Kromě těch, které fungují dlouhodobě (např. Livesport), vznikají také aplikace zaměřené pouze na konkrétní události (olympijské hry, mistrovství světa v hokeji/fotbale apod.) či soutěže (např. fotbalová Liga mistrů či hokejová NHL). Dalšími aplikacemi z této skupiny jsou např. aplikace určené ke **sledování aktuálního stavu burzy**.

Druhou skupinou jsou aplikace, jejichž cílem je **okamžité doručení informace** z jednoho klientského zařízení na jiné. Příkladem jsou hlavně nejrůznější **chatovací aplikace** sloužící k interaktivní psané komunikaci mezi lidmi.

Dále existují i aplikace, u nichž není aktuálnost poskytovaných informací tak klíčová, ale dokáží také významně těžit z výhod, které poskytuje používání Real-Time databáze. Jako příklad lze uvést nejrůznější **sociální sítě** či aplikace zprostředkovávající **předpověď počasí**.

3.2 Real-Time dotaz

Informace dodána Real-Time databází skrze Real-Time dotaz má dvě komponenty: inicializační výsledek a události změn. Obě vychází z tradičních DBMS.

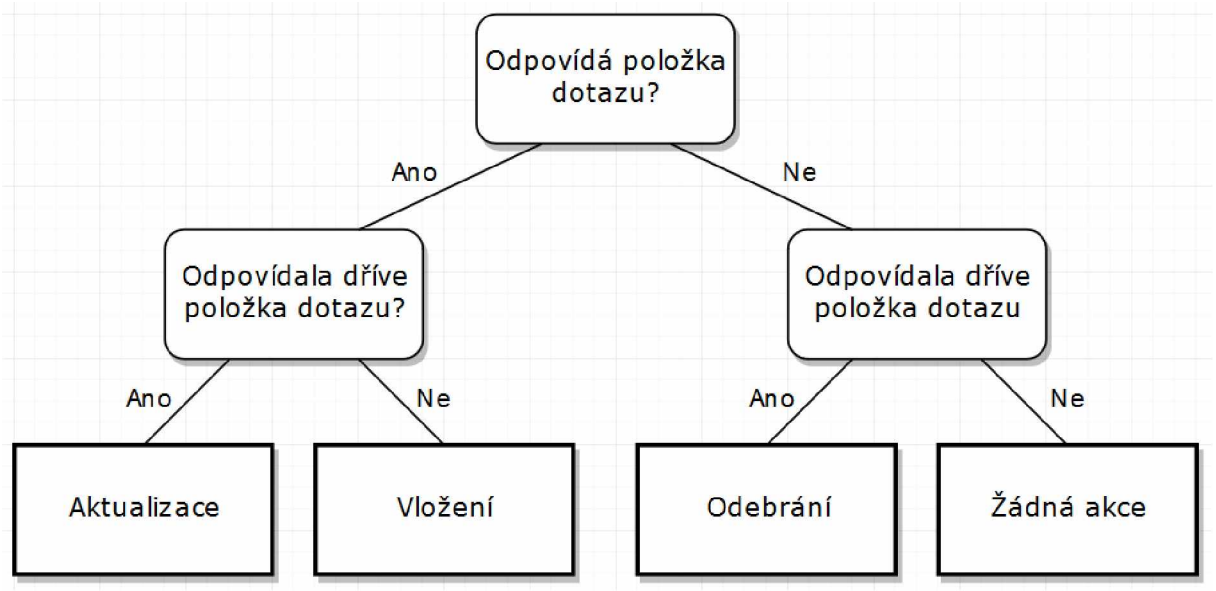
Inicializační výsledek koresponduje s daty vrácenými základním ad hoc databázovým dotazem a obsahuje obraz persistentních dat, který je požadován. **Události změn** (nazývané též notifikacemi změn) jsou odeslány zaregistrovaným klientům, pokud je výsledek aktualizován (vložení nových dat nebo změna stávajících), tj. zachycují vývoj obrazu dat v čase. Skrze tyto události klienti přijímají veškeré informace nutné k udržování inicializačního výsledku v aktuálním stavu.

Existují dva odlišné typy databázových Real-Time dotazů, které jsou různé v tom, jakým způsobem vystavují data aplikaci. **Event stream dotazy** jednoduše zveřejňují aplikaci surové události změn, takže může aplikaci udržovat aktuální kopii výsledných dat, popřípadě aplikovat nějakou svou business logiku. **Self-maintaining dotazy** jsou oproti tomu abstraktnější, protože provádějí údržbu výsledku transparentním způsobem a poskytují klientovi kompletní (aktualizovaný) výsledek dotazu na každou událost změny (namísto pouhého rozdílu oproti přechozímu výsledku).

Změny ve výsledku dotazu jsou shromažďovány skrze kontinuální proces, který se posouvá prostřednictvím datového toku databáze, podobně jako kontinuální proces v systému řízení. Za účelem identifikace relevantních aktualizací pro daný Real-Time dotaz musí databáze zkontrolovat každou operaci zápisu, která by mohla ovlivnit výsledek dotazu. Tento úkol je pro některé dotazy přímočarý a pro ostatní spíše komplikovaný.

Rozhodovací strom, který je zobrazený na obr. 6, ilustruje, jak je úkol (dotaz nebo aktualizace dat) překládán na jednoduché otázky pro každou zapisovanou datovou položku (pokud je aktualizovaných položek více, bývají transformovány na sadu aktualizací jednotlivých položek, aby byl tento přístup použitelný):

- Odpovídá nyní položka dotazu?
- Odpovídala dříve položka dotazu?



Obrázek 6: Rozhodovací strom pro Real-Time dotaz

Pokud datová položka **odpovídá dotazu po vložení** nebo **aktualizaci** (levá větev), je to buď položka, která již **byla změněna**, nebo dřívější neshoda, která odpovídá právě **vloženému výsledku**. Podobně, kdykoli je datová položka **odebrána** nebo **neodpovídá dotazu po vložení** či **aktualizaci** (pravá větev), jedná se buď o odpovídající položku, která právě opustila výsledek, nebo se nevztahuje k žádnému výsledku.

V případě **mnoha souběžných dotazů** v reálném čase nebo vysoké propustnosti aktualizace se samozřejmě tento **průběžný proces monitorování stává velmi nákladným**. Řazené dotazy navíc mohou vyžadovat dodatečné činnosti, aby bylo možné udržet pořadí výsledků a vynutit limit nebo offset. Stejně tak spojovací či agregační dotazy mohou ještě více zvyšovat režii, protože vyžadují zachování čítačů průběžných výsledků nebo jiných datových struktur, které jsou implicitně vyžadované pro udržení skutečného výsledku dotazu. Je možné aplikovat některé optimalizace (např. dávkování), které vymění průchodnost za zvýšení latence. Rovněž složitost nemusí být kvadratická pro dotazové výrazy, které umožňují efektivní indexaci (např. porovnání). Pokud je však povinná velmi nízká latence, neexistuje žádná alternativa k zvažování každé operace zápisu v kontextu každého aktivního Real-Time dotazu. Aby to bylo proveditelné, musí být aplikované odpovídající **škálování**.

V této podkapitole bylo čerpáno z [29].

3.3 Populární řešení

V této podkapitole jsou diskutována současná nejpopulárnější řešení Real-Time databází. Některé tyto systémy existují zcela samostatně (Firebase), jiné jsou postavené na vrcholu

klasické pull-based NoSQL databáze a tvoří tedy rozhraní skrze které se s těmito databázemi komunikuje. Protože v této oblasti stále nebyly zavedeny žádné standardy, jsou pro tento účel porovnány architektury jednotlivých systémů

Meteor

Meteor je JavaScript framework sloužící k vývoji reaktivních webových aplikací či stránek a jako datové uložiště využívá dokumentovou databázi MongoDB. Zajímavé je, že nabízí dvě různé implementace pro detekci relevantních změny výsledku dotazu.

První z nich je tzv. **původní přistup** kombinující dva mechanismy pro detekci změn přijatých serverem samotným a změn přijatých jiným serverem:

1) Meteor aplikační server nejprve vykoná **monitoring změn** a zjistí změnu stavu, která je relevantní s aktuálně aktivními Real-Time dotazy.
2) Nově přidaný objekt není pouze směřován do databáze, ale je také **přeložen do dotazu souvisejícího se změnou v databázi** pro každý Real-Time dotaz (pokud je např. detekováno přidání nového objektu, je tento objekt odeslán všem klientům, kteří poslouchají tento konkrétní dotaz).

Nicméně při nasazení aplikace na více serverů monitoring změn nestačí, protože zapisovací operace provedené na jednom serveru nebudou zachyceny na serveru jiném. Z tohoto důvodu je monitoring změn v původním přístupu kombinován se strategií **pool-and-diff**.

3) V té se provádí **periodické přehodnocování dotazu** (pool)
4) a **odeslání pouze relevantních aktualizací** (diff).

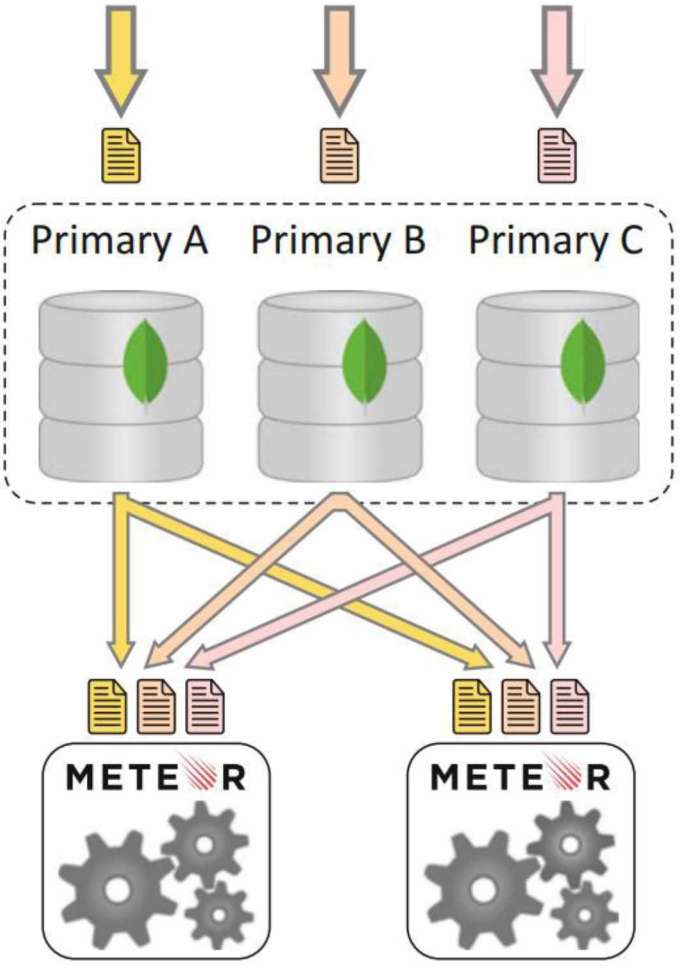
Při kombinaci těchto dvou strategií obdrží klient relevantní **aktualizace okamžitě po provedení lokálního monitoringu** nebo **s drobným zpožděním prostřednictvím strategie pool-and-diff** .

Zjevnou nevýhodou může být potenciální stálost oken ohraničených pooling intervalem (ve výchozím stavu 10 sekund) a i když je tolerována občasná několikasekundová latence, stategie **pool-and-diff se stane neproveditelnou při větším počtu současně aktivních Real-Time dotazů z důvodu,** že každý z nich periodickým poolingem zvyšuje režijní náklady na DBMS.

Z důvodu značných nevýhod strategie pool-and-diff byl původní přístup nahrazen implementací **oplog tailing**, která využívá **sledování MongoDB replikačního logu** k získání celého toku změn pro každý aplikační server. Využívá se zde distribuce dat v databázi MongoDB, kdy je **kombinováno rozdělení dat na jednotlivé servery s jejich současnou master-slave replikací**.

Všechny zapisovací operace jsou tedy nejprve aplikovány na primární server, zapsány do speciálního kruhového bufferu představující replikační log nazývaný **oplog** a následně doručeny sekundárním serverům. Základní myšlenka této implementace je tedy taková, že **Meteor aplikační servery jsou zapojeny do MongoDB replikace jako sekundární servery**.

Jak ilustruje obr. 7, každý **Meteor aplikační server využívá oplog každého primárního serveru** a **nikdy nemůže zmeškat jedinou zapisovací operaci**. Toto nastavení sice eliminuje nedostatky strategie pool-and-diff, ale **umožňuje pouze malou propustnost zápisu**. Zde se totiž Meteor odchyluje od cesty, jak MongoDB používá oplog.



Obrázek 7: Implementace Real-Time dotazu pomocí sledování oplogu v systému Meteor[6]

Zatímco každý MongoDB sekundární server využívá pouze jeden primární, Meteor server využívá kombinovanou propustnost celého MongoDB clusteru. Následkem toho je, že

[6]https://medium.baqend.com/real-time-databases-explained-why-meteor-rethinkdb-parse-and-firebase-dont-scale-822ff87d2f87

MongoDB sice může být škálována, Meteor však nikoliv, což je kritický problém pro produkční nasazení aplikace.

I přes to, že Meteor poskytuje dvě různé implementace Real-Time dotazů, **ani jedna nezvládne efektivně obcházet rozdělovací mechanismus databáze MongoDB**, kterou využívá na pozadí. V původním přístupu zvládne DBMS zátěž pouze při několika aktivních Real-Time dotazech. Implementace oplog tailing je zase efektivní pouze v případě, že je potřeba pouze nízká propustnost zápisu. [30]

RethinkDB

RethinkDB je **JSON dokumentová databáze** s expresivností dotazu srovnatelnou s MongoDB. Nicméně RethinkDB je kompletně nezávislý projekt, který se zdá v některých ohledech ambiciózněji než MongoDB, kdy nabízí pokročilé funkce jako pull-based spojovací dotazy. Mezi nejzajímavější speciality patří **changefeeds**: Real-Time dotazy s rozhraním pro **event stream dotazy**.

Technika propagace změn je velmi podobná Meteor implementaci oplog tailing. Klienti nekomunikují přímo s databázovými uzly ale s aplikačním serverem, přičemž **každý server běží na instanci RethinkDB proxy**, což je **proces přenášející komunikaci mezi klienty a RethinkDB clusterem**.

Klient nejprve **zaregistruje Real-Time dotaz** v RethinkDB proxy, která pošle dotaz databázi na odběr a následně **sleduje všechny operace zápisu** z RethinkDB clusteru k **udržování výsledku Real-Time dotazu**. Klient poté dostane zpět inicializační výsledek dotazu společně s tokem změn (changefeed).

RethinkDB oproti Meteoru nespoléhá na externí technologie, jako je oplog a interní propagaci změn řeší elegantněji. Oproti tomu, stejně jako oplog tailing, RethinkDB ruší veškeré benefity z databázového rozdělení dat tím, že zatěžuje jednotlivé aplikační servery monitoringem kompletního clusterového zapisovacího toku. Kvůli tomuto omezení propustnosti zápisu **mají RethinkDB changefeeds stejné výkonnostní limity jako sledování oplogu v Meteoru**. Dále **neobsahuje žádné API pro self-maintaining dotazy**, kvůli čemuž musí být aplikační logika implementována na kompletní výsledek poskytnutý event stream dotazem.

Baqend

Baqend je plně spravovatelná platforma pro vývoj webů a aplikací podobná Firebase, ale je postavena na pull-based databázi MongoDB, podobně jako systém Meteor.

Firebase

Firebase je proprietární služba o jejímž technologickém fungování za poskytovaným API existuje velmi málo informací (v porovnání s ostatními systémy diskutovanými v této podkapitole). Tato služba je vyvíjena již od roku 2011 a v roce 2014 byla odkoupena společností Google. [31]

Oproti ostatním JSON dokumentovým databázím, Firebase **databázový model není reprezentován kolekcí JSON dokumentů** ale pouze jedním cloudově hostovaným JSON dokumentem o **maximálně 32 úrovňové hloubce**. Aby bylo možné přistupovat k datům, musí klient procházet hierarchií a požadovat konkrétní podřízené uzly, pro které obdrží aktualizace okamžitě v případě úpravy dat někým jiným. Z toho plyne, **že Firebase je nativní push-based databází** (není pouze rozhraním kolem pull-based databáze). [30]

Databáze **umožňuje škálování rozdělením dat do několika instancí**, díky čemuž je možné přesáhnout výkonnostní limity platné pro jednotlivé instance (např. lze použít pro chatovací aplikaci sloužící samostatným nezávislým skupinám uživatelů, kdy se každá skupina připojuje k vlastní instanci). V tomto případě **každý dotaz musí běžet proti jediné databázové instanci**, **data by neměla být sdílena či duplikována** napříč jednotlivými instancemi (popř. pouze minimálně) a každá **klientská aplikace** by měla být **v daném okamžiku připojena pouze k jedné instanci**. [33]

Každá **klientská aplikace by také měla vědět, ke které databázové instanci se má připojit**. Existuje sice možnost vytvoření mapy obsahující informace o tom, jak jsou data uložena napříč jednotlivými instancemi, využívání takové mapy však pro DBMS náročnější než přímé připojování k dané instanci. Dále je ještě důležité zmínit, že pokud klient potřebuje připojení k více databázím během dané session, měl by být snížen počet současných připojení ke každé instanci připojením pouze po dobu, po kterou je to nezbytně nutné. [33]

Ve škálovatelnosti sice Firebase netrpí úzkými místy které se objevují v implementacích Real-Time dotazů systémů Meteor, RethinkDB nebo podobných, ale také **neobsahuje tak funkčně sofistikovaný dotazovací mechanismus**, jako obsahují tyto systémy. Komplexnější dotazy tedy musí být prováděny pomocí načtení nadmnožiny dat a následné operace nad nimi, což není efektivní. [32]

Firebase dále umožňuje **pouze načtení celých datových podstromů** (bez možnosti jakéhokoliv stránkování nebo určení limitu) cože vede jednak ke stažení zbytečně velkého množství dat (to lze řešit promazáváním databáze, příp. kopírováním těchto dat do archivní

databáze) a jednak k **denormalizovanému datovému modelu**, který není pro NoSQL databázi tak přirozený jako vnořování. [34]

V současné době je Firebase Real-Time Database postupně nahrazována novým databázovým řešením od Google nazývaným **Cloud Firestore**. Tento DBMS postavený nad dokumentovou databází Google Cloud Datastore **obsahuje několik vylepšení**, zmínit lze např. datovou strukturu skládající se z více JSON dokumentů, bohatší funkce, rychlejší dotazy spolu s lepší a automatizovanou škálovatelností.

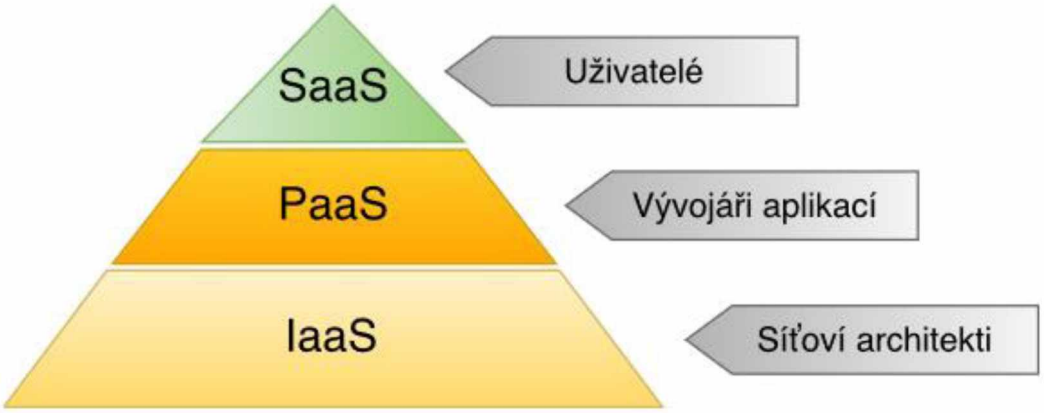
4 Backend jako služba (BaaS)

V předchozí kapitole byla rozebrána Real-Time DBMS, který je v praktické části použit pro ukládání dat a jejich synchronizaci na klientská zařízení. Tato databáze bývá většinou poskytována některou cloudovou službou ve formě nazývané „**backend jako služba**“ (BaaS).

V této kapitole je tento pojem nejprve vysvětlen, dále jsou uvedeny obvyklé funkce, které jsou pomocí BaaS zpřístupňovány, výhody a rizika, která jsou s touto cloudovou službu spojeny, jak vybrat správné BaaS řešení, a nakonec jsou představeny některé cloudové služby, které BaaS poskytují.

4.1 Definice pojmu BaaS v rámci cloudových služeb

K vysvětlení termínu BaaS je potřeba nejprve znát pojmy **infrastruktura jako služba** (IaaS), **platforma jako služba** (PaaS) a **software jako služba** (SaaS), které se ve spojení s cloudovými službami vyskytly o něco dříve. Zjednodušeně lze říct, že IaaS představuje konkrétní hardware, který je využívaný některou cloudovou službou, která na něm provozuje svojí platformu (např. Amazon Web Services). Tato platforma je dále poskytována vývojářům pro nasazení jejich aplikace (PaaS), kterou následně používají koneční uživatelé (SaaS). [35] Celý tento tradiční model je zobrazen na obr. 8.



Obrázek 8: Tradiční distribuční modely cloudových služeb[7]

Protože nasazení aplikace na platformu bylo příliš složité, byl vymyšlen BaaS (obr. 9), který představuje určitou nadstavbu nad PaaS, kterou si lze představit jako SDK či API, prostřednictvím nichž jsou vývojářům zpřístupněny funkce dané platformy, které nahrazují cloudový backend pro mobilní či webovou aplikace (sestávající např. z databáze a určité aplikační logiky). [35, 36]

[7] [35]

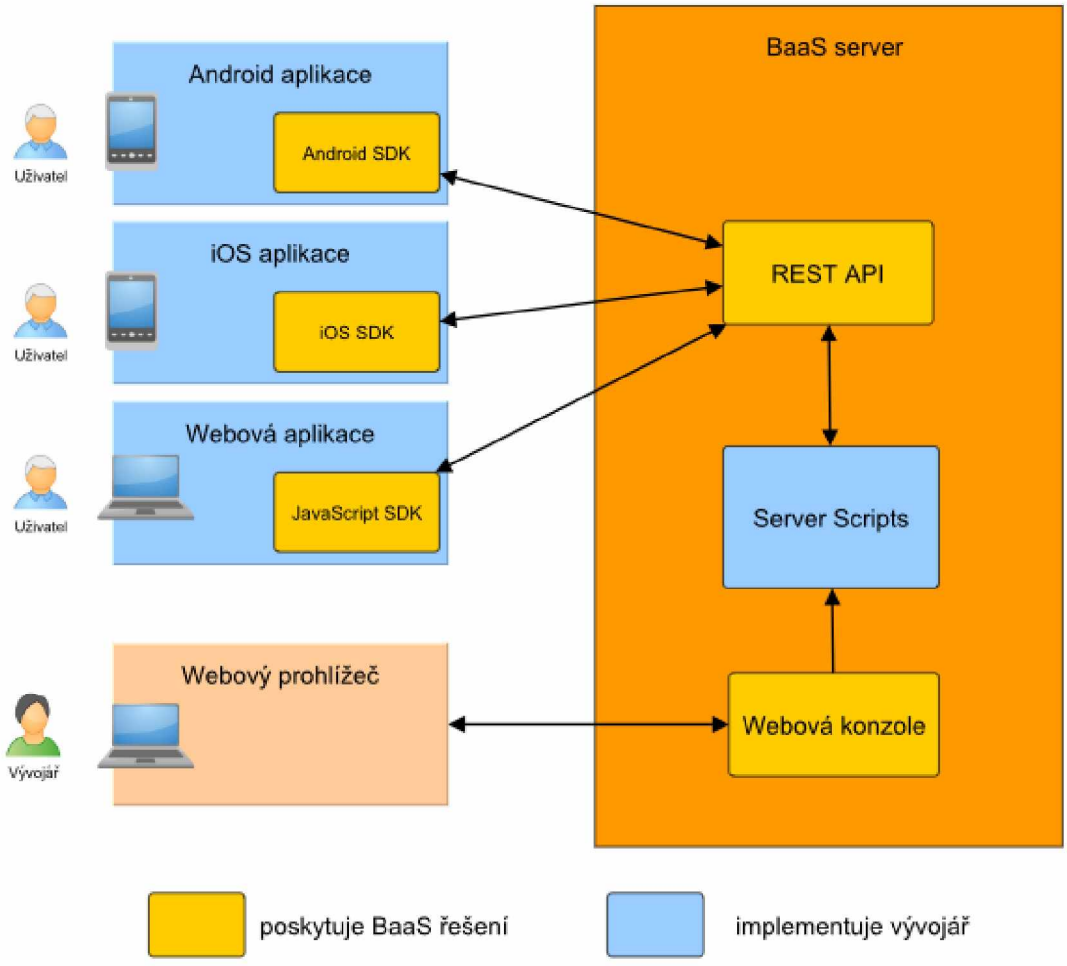
Existuje také BaaS určený především pro mobilní aplikace (tzv. mBaaS), ale většinou spíše záleží na marketingové strategii a mBaaS se od klasického BaaS příliš neliší. [35, 36]

4.2 Obvyklé funkce poskytované BaaS

I přes to, že funkce napříč jednotlivými BaaS řešeními bývají různé, lze vyjmenovat alespoň obvyklé funkce, které se většinou vyskytují. Cílem následujících sekcí je především obecná ukázka možností BaaS.

Datové uložiště

Většinou se jedná o některou NoSQL databázi (často bývá použita databáze MongoDB), i když se občas najdou i BaaS řešení, které používají tradiční RDBMS. Komunikace s databází probíhá prostřednictvím SDK. Dotazovací možnosti se liší a závisí především na typu databáze. [35]



Obrázek 9: Architektura BaaS řešení[8]

[8][21]

Správa uživatelů a autentizace

Klíčovou součástí BaaS je nějaká implicitní správa uživatelů, resp. jejich autentizačních a autorizačních informací, s čímž souvisí registrace, přihlášení, životní cyklus autentizace, role, oprávnění atd. Obvyklá je také podpora pro integraci autentizace skrze nějaký široce používaný autentizační systém (Google, Microsoft, Facebook atd.). [35, 37]

Zasílání zpráv

Baas řešení zpravidla obsahuje nějaký systém (či více systémů) pro zasílání zpráv uživatelům aplikace. Nejčastější implementovanou formou jsou push notifikace, dále bývá zasílání zpráv implementováno pomocí vestavěného chatu, emailu či SMS. [35, 37]

Cloudové funkce

Cloudové funkce umožňují implementaci vlastní aplikační logiky na straně serveru, většinou prostřednictvím skriptů. Tyto skripty bývají manuálně volané skrze API či SDK, navázané na vybranou událost či pravidelně volané plánovačem. [35]

Analýza a predikce chování uživatelů

Mnoho poskytovatelů BaaS jako klíčovou funkci svého řešení nabízí analýzu a predikci chování uživatelů v rámci aplikace. Pokročilejší informace jsou pak většinou nabízeny v rámci dražších tarifů služby. [37]

4.3 Výhody a rizika spojená s použitím BaaS

Jak již bylo uvedeno výše, hlavním cílem BaaS bylo poskytnutí funkcí obvyklých pro cloudový backend aplikace prostřednictvím API či SDK. Kromě toho ale logicky vyplynuly i některé další výhody a rizika s tím spojená. V této podkapitole je čerpáno hlavně z [35].

Výhody

Aplikaci již lze **vyvinout bez potřeby backend vývojářů**, když jsou pouze volány poskytované funkce dané platformy, což výrazně usnadňuje vývoj aplikace. BaaS řešení dnes poskytují i funkce, jejichž vlastní implementace by byla časově velmi náročná (tedy i drahá).

Protože se jedná o distribuovaný systém, plyne z toho i možnost **škálovatelnosti platformy / infrastruktury**, která někdy bývá i automatizovaná dle aktuálního provozu. Kvůli tomu většinou BaaS řešení využívá právě NoSQL databáze, které je možné také škálovat.

Častým problémem při vývoji webových či mobilních aplikací bývá komunikace se serverem. BaaS řeší tento problém za vývojáře a **obsahuje knihovnu umožňující bezpečnou a efektivní komunikaci mezi klientem a serverem**, která je pouze volána prostřednictvím API či SDK.

Rizika

Jak již bylo řečeno výše BaaS řešení často funguje prostřednictvím API či SDK, což kromě řady výhod přináší i řadu nevýhod.

Vyvíjená **aplikace se musí vždy určitým způsobem přizpůsobit** tomu, jak je daná funkcionalita navržena. V některých případech (hlavně u komplexnějších aplikací) se vývoj aplikace neobejde bez vlastního backendu pro řešení určitých specifických scénářů.

Největší limitem ovšem bývá **rozhraní pro komunikaci s databází**. To bývá většinou navrženo tak, aby poskytovalo pouze operace, které lze vykonat velmi rychle a škálovat, což může být značně omezující hlavně pro vývojáře zvyklé pracovat pouze s relační databází.

Dalším rizikem je **velká závislost na daném BaaS řešení**, které jako každá ze služeb třetích stran může jednoho dne skončit. S tím se musí vývojář smířit, pokud chce použít toto řešení. Proto je velmi důležitý řádný průzkum vybraného řešení před jeho použitím.

Dále je zapotřebí ve své aplikaci **obalit BaaS rozhraní vlastním rozhraním**, díky čemuž je aplikace méně závislá na externím rozhraní a vývojář lehce vyřeší případné změny externího rozhraní. Lze se tímto také připravit na eventuální výměnu používaného BaaS řešení, ale vzhledem k tomu že mezi BaaS řešeními neexistují nějaké jednotné konvence, tak je i tak tato výměna těžko proveditelná a nemělo by se s ní počítat.

4.4 Výběr vhodného BaaS řešení

Tato podkapitola nabízí možný způsob výběru BaaS řešení pro specifickou aplikaci, u které bylo rozhodnuto (nebo se zvažuje) o použití BaaS. V následujících sekcích jsou popsané jednotlivé kroky. Vychází se v nich z informací, které nabízí předchozí podkapitoly.

Definice kritérií

Nejprve je potřeba definovat kritéria, která musí splňovat vybrané BaaS řešení. Tato kritéria se musí týkat hlavně funkcí (popsaných v podkapitole 4.2), způsobu hostování řešení (jestli ho zajišťuje poskytovatel nebo se konečné řešení nasadí na servery zákazníka) a ceny řešení.

Základní analýza nabízených řešení

V tomto kroku je třeba provést analýzu trhu BaaS a zjistit informace o jednotlivých řešeních. Tyto informace se týkají primárně kritérií, která jsou definovaná v předchozím kroku. Důkladnější analýza je provedena později.

Selekce vyhovujících řešení

Zde se na základě definovaných kritérií a analýzy nabízených řešení vyberou ta řešení, která splňují definovaná kritéria. Pokud nebude vybráno žádné řešení, je potřeba upravit definovaná kritéria a provést selekci znova (případně ustoupit od úmyslu použití BaaS řešení).

Analýza vyhovujících řešení

Po selekci vyhovujících řešení nastává potřeba důkladnější analýzy vyhovujících řešení, jejíž cílem je zamezit, aby konečné řešení bylo chybové či jeho poskytovatel v brzké době ukončil svoji činnost.

Při té je potřeba se hlavně zaměřit na důvěryhodnost (jak známý je poskytovatel, jaká je spokojenost jeho zákazníků, jak dlouho je dané řešení na trhu apod.) a robustnost (na jakých technologiích dané řešení stojí, jak je stabilní apod.) vyhovujících řešení.

V případě zjištění, že některá řešení z tohoto pohledu nevyhovují, provede se další selekce.

Analýza proveditelnosti klíčových scénářů

Dále je třeba otestovat proveditelnost klíčových scénářů aplikace pro jednotlivá vyhovující BaaS řešení. Tento krok je jedním z nejdůležitějších a neměl by být podceňován.

Jeho cílem je zamezení tomu, že při implementační části dojde ke zjištění, že některé klíčové funkcionality aplikace jsou s daným BaaS řešení neproveditelné. Sleduje se zde např., jestli lze vykonat potřebné databázové operace v dostatečné rychlosti a jaká datová či výkonová náročnost je kladena na klientská zařízení.

V případě zjištění, že některá řešení z tohoto pohledu nevyhovují, provede se další selekce.

Výběr konečného řešení

Nakonec proběhne výběr konečného Baas řešení použitého pro danou aplikaci. Jednotlivá řešení, která prošla předchozími kroky, jsou zde porovnány hlavně z hlediska nákladů na provoz (předchozí selekce by měly zajistit, že všechna řešení jsou vyhovující).

4.5 Cloudové služby poskytující BaaS

V dnešní době, kdy se BaaS stal poměrně populární, existuje velké množství nejrůznějších řešení této formy cloudové služby. V této podkapitole jsou představeny některé z nejrozšířenějších a nejzajímavějších.

Back4App

BaaS je řešení nabízející správu databáze Real-Time dotazy, push notifikace, integraci pomocí SDK a správu uživatelů. Všechny tyto funkcionality jsou postavené nad open source technologiemi (zmínit lze např. Parse Server, MongoDB, Postgrees nebo NodeJS).

Tato platforma je jednoduchá k použití, vývojář může snadno spravovat své verze open source kódu a zdrojový kód může být snadno upraven pro potřeby maximálního výkonu.

Používán je dotazovací jazyk GraphQL (vyvinutý společností Facebook), jehož výhodou je načítání data předvídatelným způsobem v jednom requestu.

Parse

Parse je open source framework podporovaný širokou komunitou vývojářů. Jeho cílem je nabídnout vývojářům rychlejší cestu k vytvoření aplikace.

Vývojáři mohou využít API server modul pro NodeJS, SDK, knihovny a adaptéry. Zároveň nabízí funkci nazvanou Cloud Code, která umožňuje přizpůsobit kód pomocí Javascriptu. Tuto platformu v minulosti používal i Facebook, nejrozšířenější sociální síť na světě.

Parse nabízí možnost uložení základních datových typů, dotazů, lokací a obrázků prostřednictvím několika řádků kódu. Dále umožňuje zpracovávat, vyhledávat a filtrovat pomocí webového prohlížeče dat. Také zde je používán dotazovací jazyk GraphQL

Vzhledem k tomu, že je framework open source, lze zdrojový kód upravovat a přizpůsobovat konkrétní potřebám. Používání tohoto frameworku je navíc zcela zdarma a provozovatel aplikace tak musí platit pouze náklady na hosting.

Google Firebase

Firebase je populární BaaS řešení zakoupené společností Google. Toto řešení je použité také v rámci této diplomové práce a popsáno v další kapitole.

Cloudkit

Cloudkit je uzavřené řešení provozované společností Apple, které je používáno pro vývoj iOS aplikací a má pro to nativní SDK. Tato platforma běží od roku 2015 a je snadné ji integrovat s iOS aplikacemi.

Poskytuje databázové, autentizační a úložné služby, které umožňují vývojářům soustředit se na vývoj na straně klienta (klientskou aplikaci). Soustředí se primárně na uložiště dat s podporou autentizace pomocí iCloud identifikace a API před ním.

Hlavní výhodou Cloudkitu je nativní SDK pro iOS. Nativní aplikace běžící na iOS mají kompletní přístup do zařízení a mohou využít všech jeho funkcí. To vede k nejlepšímu možnému výkonu a uživatelskému dojmu.

Kinvey

Kinvey představuje BaaS řešení zaměřené na podniky, které umožňuje provozovat podnikové aplikace bez potřeby vlastního backendu a nabízí funkce podobné ostatním poskytovatelům BaaS. Ty zahrnují např. databázi, autentizaci, push notifikace nebo služby určování polohy. Řešení (včetně hostingu a provozu) je k dispozici i zdarma pro aplikaci s maximálním počtem 100 uživatelů.

Mezi hlavní výhody používání Kinvey patří podnikové služby zahrnující např. integraci se Salesforce, SAP, Workday, Microsoft Active Directory nebo Oracle. Dále nabízí také zpravodajské nástroje zahrnující nejrůznější analytické reporty (dostupné pro zdravotní aplikace kompatibilní s HIPAA). Součástí Kinvey je také podpora dostupná 24/7.

AWS Amplify

Jedná se o open source knihovnu, která spolupracuje s Amazon Web Services. Vývojáři ji mohou využít pro tvorbu škálovatelných aplikací bez vlastního backendu. Kromě toho se také snadno integruje s iOS, Android a webovými aplikacemi. Funkce zahrnují analytiku, notifikace, AR/VR, uložiště a API. Služba nabízí také omezenou bezplatnou verzi pro produkční prostředí.

Mezi hlavní výhody patří extrémně jednoduchá integrace s CloudFront a nasazení obsahu globálně. Dále nabízí funkci nazvanou GraphQL Transform, která usnadňuje proces vývoje.

Azure Mobile Apps

Také Microsoft nabízí své BaaS řešení, kterým jsou Azure Mobile Apps (Azure App Service nebo Azure Mobile Services). Tato platforma se mimo Azure integruje také s multiplatformní frontend službou Xamarin.

Mezi hlavní výhody patří bezpečnostní protokoly, které poskytují výkonné podnikové zabezpečení při vytváření aplikací. Dalšími výhodami jsou např. funkce Off-Line Sync (automatická synchronizace uživatelských dat) a AD Integrations (umožňuje firemní přihlášení).

5 Google Firebase

Google Firebase je BaaS řešení, které vyrostlo v platformu pro vývoj vysoce výkonných a škálovatelných aplikací pro web a mobilní zařízení. Firebase byl původně vyvinut pro tvorbu chatovacích aplikací komunikujících v reálném čase, ale postupně se začal používat i k dalším účelům.

V této kapitole jsou postupně popsány jednotlivé části Google Firebase, které byly použity v praktické části této diplomové práce. V jednotlivých podkapitolách je čerpáno z oficiální dokumentace Google Firebase [38].

5.1 Authentication

Většina aplikací potřebuje znát identitu uživatele. Znalost identity umožňuje aplikaci bezpečně ukládat uživatelská data v cloudu a nabízet stejnou uživatelskou přívětivost na všech zařízeních uživatele.

Firebase Authentication nabízí backend služby, snadné použití nejrůznějších SDK a dokonce možnost použití hotových UI knihoven k autentizaci uživatele. Podporuje autentizaci pomocí hesel, telefonních čísel, populárních autentizačních systémů třetích stran (jako např. Google, Facebook nebo Twitter).

Dále je úzce integrovaná s ostatními Firebase službami a využívá průmyslové standardy jako OAuth 2.0 a OpenID Connect, což umožňuje snadnou integraci s vlastním autentizačním systémem.

Předpřipravené autentizační řešení FirebaseUI Auth

Jedná se o doporučenou cestu k přidání kompletního autentizačního systému do aplikace. FirebaseUI nabízí předpřipravené autentizační řešení zahrnující UI komponenty pro přihlášení uživatele s použitím emailu, hesla, telefonního čísla nebo populárních autentizačních systémů třetích stran. Další výhody tohoto řešení jsou:

- Implementuje osvědčené postupy pro autentizaci na mobilních zařízeních a webových stránkách.
- Zpracovává také okrajové případy typu obnovení účtu a propojení účtů, které mohou být citlivé na bezpečnost a složitější ke správné implementaci.
- Lze snadno přizpůsobit tak, aby zapadalo do vizuálního stylu aplikace a je open source, takže nijak vývojáře neomezuje při integraci do aplikace.
- Dostupné při vývoji aplikacích na platformách iOS, Android a Web.

Firebase Authentication SDK

Pokud vývojáři nevyhovuje použití předpřipravených UI komponent, může k autentizaci využít Firebase Authentication SDK. To nabízí autentizaci pomocí emailu a hesla, autentizačních systémů třetích stran (Google, Facebook, Twitter a GitHub) nebo telefonního čísla. Zároveň umožňuje integraci s vlastním autentizačním systémem nebo vytváření dočasných anonymních účtů, které můžou být později povýšeny na běžné uživatelské účty.

Jak to funguje

Po přihlášení uživatele do aplikace nejprve Firebase backend služby obdrží ověřovací údaje uživatele (těmito údaji mohou být např. email a heslo nebo OAuth token z autentizačních služeb třetích stran), tyto údaje jsou ověřeny a je vrácena odpověď do aplikace.

Po úspěšném přihlášení má aplikace přístup do základních informací z uživatelova profilu a může řídit přístup uživatelů k datům uložených v dalších Firebase produktech. Aplikace může také použít autentizační token k ověření identity uživatele ve vlastních backend službách.

Defaultně můžou autentizovaní uživatelé číst a zapisovat data do Firebase Realtime Database a Cloud Storage. Aplikace může ovládat přístup uživatelů modifikací Firebase Realtime Database[9] nebo Cloud Storage Security Rules[10].

5.2 Cloud Messaging

Firebase Cloud Messaging (FCM) je multiplatformní messaging řešení umožňující spolehlivě zasílat zprávy bez nákladů.

Pomocí FCM je možné zaslat notifikaci klientské aplikaci, že byl synchronizován nový email nebo nějaká jiná data. Pomocí zasílání notifikačních zpráv je možné podpořit opětovné zapojení a udržení uživatele. Pro případy typu instant messaging může být do klientské aplikace spolu se zprávou zaslána příloha o velikosti až 4KB.

Odeslané notifikace se mohou okamžitě zobrazit uživateli nebo může aplikace rozhodnout, co se na jejich základě stane (platí pro datové zprávy). Distribuce zpráv může probíhat do specifického zařízení, do skupiny zařízení nebo do zařízení přihlášených k odběru.

Lze také posílat odpovědi, rozhovory nebo jiné zprávy z klientského zařízení zpět na server přes spolehlivý a bateriově nenáročný (platí zejména v případě mobilních aplikací) FCM připojovací kanál.

[9] https://firebase.google.com/docs/database/security/index
[10] https://firebase.google.com/docs/storage/security/index

Jak to funguje

FCM implementace zahrnuje dvě hlavní komponenty pro odesílání a přijímání zpráv:

- Důvěryhodná prostředí jako Cloud Functions for Firebase nebo aplikační server, na kterém se sestavují, cílí a odesílají zprávy.
- iOS, Android nebo webová klientská aplikace, která přijímá zprávy prostřednictvím odpovídající transportní služby specifické pro danou platformu.

Zprávy lze zasílat přes Firebase Admin SDK[11] nebo přes FCM server protokoly[12]. Pro testování nebo zasílání obchodních či interaktivních zpráv s výkonným vestavěným cílením a analytikou lze také použít Notification composer[13].

Integrace do aplikace

K integraci FCM do aplikace je zapotřebí:

1. Nastavit Firebase a FCM v dané aplikaci podle pokynů pro danou platformu
2. Implementovat logiku zachycování zpráv, odběru zpráv nebo jinou funkčnost do dané aplikace.
3. Použít Firebase Admin SDK nebo jednoho z FCM server protokolů k vytvoření zasílací logiky (autentizace, sestování žádosti o odeslání a zachycování odpovědí) a vytvořit logiku ve svém důvěryhodném prostředí.

5.3 Real-Time Database

Firebase Real-Time Database (FRTD) ukládá a synchronizuje data s NoSQL cloudovou databází, ta jsou ukládána do jednoho JSON dokumentu.

Při vytváření aplikací napříč platformami (iOS, Android a JavaScript) všichni klienti sdílejí jednu FRTD instanci a automaticky přijímají aktualizace nejnovějších dat.

Oproti klasickým HTTP requestům FRTD používá synchronizaci dat. Díky tomu je docíleno toho, že při každé změně dat obdrží aktualizace všechna připojená zařízení, kterých se aktualizovaná data týkají, v řádu milisekund. To vývojáře oprošťuje od nutnosti implementace síťové komunikace mezi klientským zařízením a cloudovou databází a zároveň přináší výhodu v podobě vždy aktuálních dat v aktuálně připojených klientských zařízeních.

[11] https://firebase.google.com/docs/cloud-messaging/server#firebase-admin-sdk-for-fcm
[12] https://firebase.google.com/docs/cloud-messaging/server#choose
[13] https://console.firebase.google.com/project/_/notification

Synchronizována data zároveň zůstávají dostupná i po přechodu aplikace do offline režimu. Pokud se aktualizovaná data týkají klientského zařízení, které je v době synchronizace offline, obdrží zařízení aktuální data v okamžiku obnovy spojení.

Jak to funguje

FRTD umožňuje vytvářet bohaté spolupracující aplikace tím, že umožňuje bezpečný přístup do databáze přímo z kódu klientské aplikace. Data jsou ukládána lokálně a dokud není klientské zařízení offline, tak neustále přijímá realtime události, což poskytuje interaktivní zážitek. Když je obnoveno spojení, FRTD synchronizuje lokální změny se vzdálenými, ke kterým došlo, když bylo klientské zařízení offline (automaticky vyřeší všechny konflikty).

FRTD nabízí flexibilní jazyk pravidel postavený na výrazech, nazývaný FRTD Security Rules, definující, jak by data měla být strukturovaná a kdy mohou být čtena nebo zapisovaná. Během integrace s Firebase Authentication může vývojář definovat, kdo má přístup k jakým datům a jak k nim může přistupovat.

FRTD je NoSQL databáze a jako taková má rozdílné optimalizace a funkčnost oproti relační databázi. Realtime Database API bylo navrženo tak, aby umožnilo pouze operace, které lze provést rychle. Toto umožňuje vývojáři vybudovat skvělou realtime zkušenost, která může obsluhovat miliony uživatelů bez vlivu na odezvu. Z toho důvodu je důležité přemýšlet o tom, jaký přístup ke svým datům uživatelé potřebují a strukturovat je podle toho.

Součástí placené verze FRTD je také možnost škálování prostřednictvím více databázových instancí v jednom Firebase projektu. Tato možnost zjednodušuje autentizaci uživatelů napříč jednotlivými databázovými instancemi a zároveň je možné řídit přístup k datům v každé databázi s vlastními FRTD Security Rules pro každou databázovou instanci.

Integrace do aplikace

K integraci FRTD do aplikace je zapotřebí:

1. Integrovat potřebná FRTD SDK pomocí Gradlu, CocoaPods nebo skriptu.
2. Vytvořit reference na FRTD. Tímto způsobem lze referencovat JSON data ve stylu „users/user:1234/phone_number“ (telefonní číslo uživatele s ID 1234 v poli uživatelů).
3. Použit vytvořené reference k zápisu dat nebo k odebírání změn v datech.
4. Povolit zápis dat na lokální uložiště tak, aby byla dostupná i v případě, pokud je klientské zařízení offline.
5. Zabezpečit data pomocí FRTD Security Rules.

5.4 Remote Config

Firebase Remote Config je cloudová služba umožňující změnu chování a vzhledu klientské aplikace bez potřeby aktualizace (v případě webové aplikace je tím myšleno nasazení nové verze, v případě mobilní aplikace pak stahování a instalace aktualizace).

Když je Remote Config požíván, tak je potřeba nejprve nastavit výchozí hodnoty, které ovládají chování a vzhled aplikace. Tyto hodnoty pak lze později změnit přes Firebase konzoli či Remote Config REST API a přepsat tak výchozí hodnoty pro všechny uživatele aplikace či pro určité segmenty uživatelů.

Aplikace může pravidelně kontrolovat, zda jsou tyto změny dostupné, se zanedbatelným dopadem na výkon. Vývojář má pak možnost nastavit, kdy jsou změny aplikovány.

Tímto způsobem lze například měnit layout aplikace nebo barevné schéma (lze například využit pro sezónní propagaci). Dále lze takto poskytovat například různé variace uživatelského prostředí v závislosti na jazyku či jiných hodnotách.

To vše umožňuje klientská knihovna Remote Config. Pomocí této knihovny lze načítat hodnoty parametrů a cachovat je. Dále dává také možnost zabezpečit prostředí aplikace ovládáním načasování jakýchkoliv změn. Lze takto ovládat až 300 různých verzí Remote Config parametrů s maximální životností 90 dní.

Zásady a limity

- Nepoužívat Remote Config, pro aktualizace, které vyžadují uživatelskou autorizaci. Jinak může být aplikace vnímána jako nedůvěryhodná (týká se zejména mobilních aplikací).
- Neukládat důvěrná data v Remote Config parametrech nebo jejich hodnotách. Je možné dekódovat libovolné klíče nebo hodnoty uložené v Remote Config nastavení projektu.
- Nepokoušet se obejít požadavky cílové platformy aplikace pomocí vzdálené konfigurace.

Integrace do aplikace

K integraci Firebase Remote Config do aplikace je zapotřebí:

1. Definovat aspekty chování a vzhledu aplikace, které je možné měnit pomocí Remote Config, a převést je do parametrů, které se budou používat v aplikaci.
2. Nastavit výchozí hodnoty pro Remote Config parametry.

3. Přidat logiku pro načítání, čtení a aktivaci hodnot parametrů. Díky této logice může aplikace bezpečně a efektivně načítat hodnoty Remote Config parametrů z backendu a aktivovat je.
4. Definovat hodnoty ve Firebase konzoli nebo přes Remote Config REST API k přepsání defaultních hodnot parametrů. Toto lze učinit před i po spuštění aplikace, protože se jinak použijí výchozí hodnoty.

5.5 Cloud Storage

Cloud Storage for Firebase je navrženo pro ukládání a poskytování souborů (např. fotografie, videa nebo dokumenty). Jedná se o výkonné, jednoduché a nákladově efektivní objektové uložiště.

Dále jsou Firebase SDK pro Cloud Storage integrována s Firebase Authentication, díky čemuž Cloud Storage nabízí jednoduchou a intuitivní autentizaci. Lze také deklarovat bezpečností model, který může být založen na názvu souboru, velikosti, typu dat nebo dalších metadatech.

Síťová komunikace s Cloud Storage probíhá robustně (pokud se přenos zastaví a restartuje, spustí se ze stejného místa, kde se zastavil), což šetří uživatelský čas a v případě mobilních zařízení také mobilní data.

Cloud Storage for Firebase je navrženo pro měřítko v exabytech (v případě opravdu virálních aplikací). Používá stejnou infrastrukturu, která pohání Spotify nebo Google Photos. Soubory jsou ukládány do Google Cloud Storage[14], díky čemuž jsou přístupné z Firebase i z Google Cloudu. Škálování probíhá automaticky.

Integrace do aplikace

K integraci Cloud Storage for Firebase do aplikace je zapotřebí:

1. Integrovat potřebná Firebase SDK for Cloud Storage pomocí Gradlu, CocoaPods nebo skriptu.
2. Vytvořit reference na cestu k souboru ve stylu „images/mountains.png“ (obrázek mountains.png z kolekce images).
3. Použít reference k nahrávání či stahování do nativních typů v paměti nebo na lokální uložiště.
4. Zabezpečit soubory pomocí Firebase Security Rules for Cloud Storage.

[14] https://cloud.google.com/storage

5.6 Cloud Functions

Cloud Functions for Firebase umožňují automaticky spustit kód na backendu v reakci na vyvolané události. Kód je uložen v cloudu Googlu a běží ve spravovaném prostředí (automaticky škálovaném). Není tak potřeba spravovat nebo škálovat vlastní servery.

Funkce mohou reagovat na události generované dalšími Firebase a Google Cloud funkcemi (např. Realtime Database, Remote Config a Authentication), publikací/odběrem zpráv nebo HTTP requesty.

Integrace napříč Firebase službami je zajišťovaná použitím Admin SDK spolu s Cloud Functions, integrace se službami třetích zpráv pak implementací vlastní webhooků. Cloud Functions minimalizují „špagetový" kód a umožňují jednodušší použití Firebase a Google Cloudu uvnitř vlastních funkcí.

Cloud Fuctions jsou také plně izolovány od klientské aplikace, takže si může být vývojář jist, že jsou privátní a dělají přesně to, co mají.

Integrace do aplikace

K integraci Cloud Functions for Firebase do aplikace je zapotřebí:

1. Nainstalovat Firebase CLI a inicializovat Cloud Functions v daném Firebase projektu.
2. Napsat JavaScript funkce k odchytu událostí např. z Firebase služeb nebo HTTP requestů.
3. Použít lokální emulátor k otestování napsaných funkcí.
4. Nasadit napsané funkce pomocí Firebase CLI. Lze také používat Firebase konzoli k zobrazení a prohledávání logů z funkcí.

6 Představení praktické části

Součástí této diplomové práce je také praktická část, která slouží k demonstraci teorie obsažené v předchozích kapitolách. Tato část je popsáno a představena v této kapitole.

6.1 Hlavní kritéria

Praktická část práce byla vypracována na základě několika kritérií, které jsou shrnuty v této podkapitole.

NoSQL databáze

V praktické části musí být použity, alespoň 2 typy NoSQL databází pro demonstraci teorie probrané v kapitole 2.

Zachytávání událostí, ukládání dat a jejich synchronizace

Zachytávání událostí, ukládání dat a jejich synchronizaci na klientská zařízení musí obstarat cloudová Real-Time databáze (viz kapitola 3). Tu musí využívat mobilní aplikace, která bude odebírat data potřebná pro svůj běh. Cloudová Real-Time databáze musí automaticky doručit změny v odebíraných datech do mobilní aplikace, pokud klientské zařízení, na kterém aplikace běží, je ve stavu online.

Cloudové služby Firebase

V praktické části musí být využito Firebase BaaS řešení (viz kapitola 4). Dále musí být použita většina služeb Firebase popsaných v kapitole 5:

- Služba Authentication musí být použita pro autentizaci.
- Služba Real-Time Database musí být použita jako Real-Time databáze, která zachytává události, ukládá data do interní dokumentové NoSQL databázi (viz podkapitola 2.3) a zprostředkovává synchronizaci na klientská zařízení.
- Služba Remote Config musí být použita jako vzdálené uložiště společné konfigurace ve formě NoSQL databáze typu klíč-hodnota (viz podkapitola 2.3).
- Služba Cloud Functions musí být použita pro aktualizaci dat v Real-Time databázi.

Demonstrace funkčnosti mobilní aplikace v emulovaném i reálném prostředí

Funkčnost výsledné mobilní aplikace, vyvinuté během praktické části, musí být demonstrována v prostředí emulátoru (Android Virtual Device) i na reálném mobilní zařízení. Aplikace musí fungovat ve stavu online i offline a musí obsahovat veškeré technologie popsané v předchozích kritériích.

6.2 Zadání

Na základě kritérií bylo vytvořeno zadání pro demonstraci teoretické části práce. Popis a obsah tohoto zadání je obsažen v této podkapitole.

Téma

Jak je uvedeno v podkapitole 3.1, k optimálnímu využití Real-Time databáze dochází u reaktivních aplikací určených k sledování aktuálního stavu určitých událostí. Vzhledem k tomu, že účelem výsledné aplikace, vyvinuté během praktické části, je demonstrace použití Real-Time databáze v praxi, bylo zapotřebí, aby tato aplikace byla reaktivní.

Jako téma této aplikace bylo zvoleno **sledování výsledků fotbalových zápasů**. Aplikací podobného typu existuje celá řada. Tyto aplikace jsou velmi populární, což dokládá fakt, že neexistují pouze obecné aplikace (pro všechny sporty[15] či zaměřené na konkrétní sport[16]), ale vznikají i aplikace zaměřené pouze na konkrétní sportovní události (fotbalové UEFA EURO 2020[17], hokejové Mistroství světa 2019[18]) či soutěže (fotbalová Liga mistrů[19], hokejová NHL[20]).

Mobilní aplikace Livescore pro sledování fotbalových zápasů

Tato neformální systémová specifikace obsahuje koncept činnosti mobilní aplikace Livescore zaměřené na **sledování fotbalových zápasů**.

Hlavní funkcí aplikace je umožnění **živého sledování fotbalových utkání**, seskupených podle lig, **v textové podobě**. Mimo změn výsledků musí aplikace živě zobrazovat také jednotlivé **události zápasu** (góly, asistence, střídání, žluté karty a červené karty) a umožňovat **diskuzi o zápase mezi uživateli aplikace** prostřednictvím chatu. Tyto informace by se měly v aplikaci aktualizovat bez potřeby interakce uživatele.

Mimo živé sledování výsledků by měla aplikace také umožňovat zobrazení **výsledků zápasů** a **podrobností soutěží**. Výsledky zápasů by měly být filtrované podle vybraného dne a soutěže. V podrobnostech soutěže by měla být vlajka dané země, základní informace, rozpis utkání, výsledky utkání, tabulka a tabulka střelců.

[15] https://play.google.com/store/apps/details?id=eu.livesport.LiveSport_cz
[16] https://play.google.com/store/apps/details?id=atpwta.live
[17] https://play.google.com/store/apps/details?id=com.uefa.euro2016
[18] https://play.google.com/store/apps/details?id=de.hit.icehockey
[19] https://play.google.com/store/apps/details?id=com.uefa.ucl
[20] https://play.google.com/store/apps/details?id=com.nhl.gc1112.free

Uživatel by měl mít také možnost **označit si zápasy**, které chce sledovat, **a soutěže**, jejichž podrobnosti ho zajímají.

Aplikace by měla umožnit fungování autentizovanému i anonymnímu uživateli. Autentizovaný uživatel by měl mít zobrazené stejné seznamy sledovaných zápasů a soutěží ve všech zařízeních, ve kterých se autentizoval, a mělo by se mu automaticky zobrazovat jeho uživatelské jméno v chatu u zápasu.

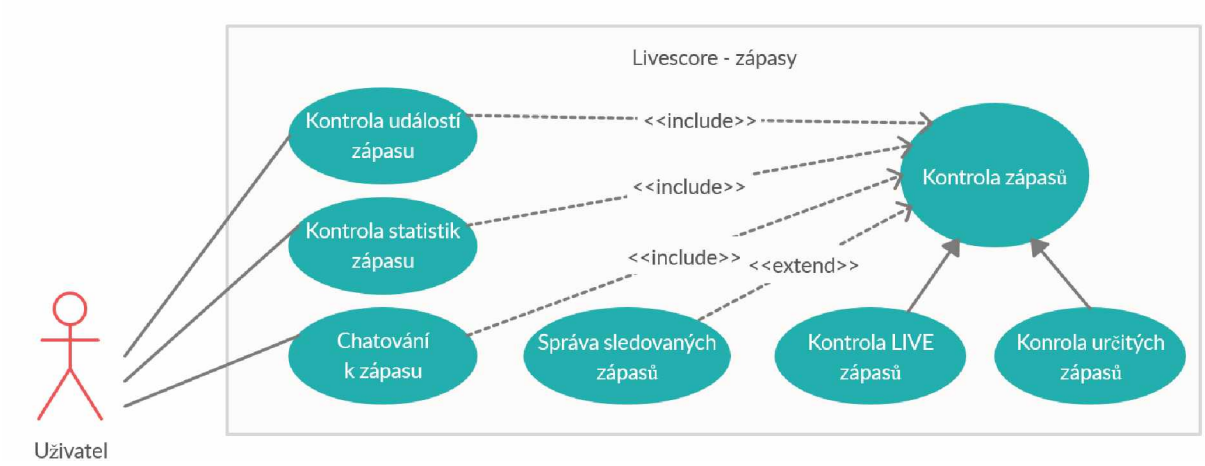
6.3 Případy užití

Na základě funkčních požadavků, definovaných v zadání, byly definováni aktéři a případy užití pro jednotlivé moduly praktické části práce. Klientská část je rozdělena do modulů **zápasy** a **soutěže**, v cloudové části jsou pak moduly **plánování** a **administrace**. V této podkapitole jsou moduly případů užití představeny.

Modul zápasy

Hlavním účelem aplikace je sledování výsledků a událostí zápasů, proto je nejdůležitějším modulem v klientské části modul zápasů (viz obrázek 10). Tento modul slouží ke:

- kontrole právě hraných zápasů,
- kontrole určitých zápasů vybraných podle daných kritérií (viz další podkapitola,
- kontrole informací o zápase,
- kontrole událostí zápasu,
- a kontrole statistik zápasu.



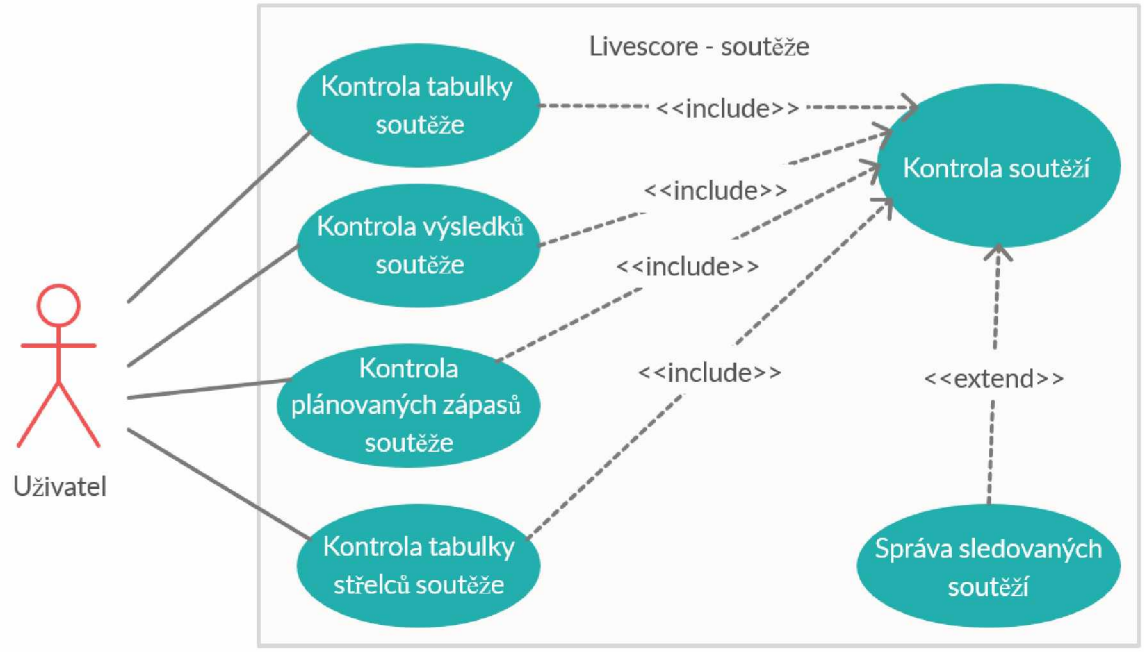
Obrázek 10: Případy užití modulu zápasy[21]

[21] [vlastní]

Modul soutěže

Dalším modulem klientské části je modul soutěží (viz obrázek 11). Tento modul je určen ke:

- kontrole soutěží,
- kontrole tabulky soutěže,
- kontrole výsledků soutěže,
- kontrole plánovaných zápasů soutěže
- a kontrole tabulky střelců soutěže.



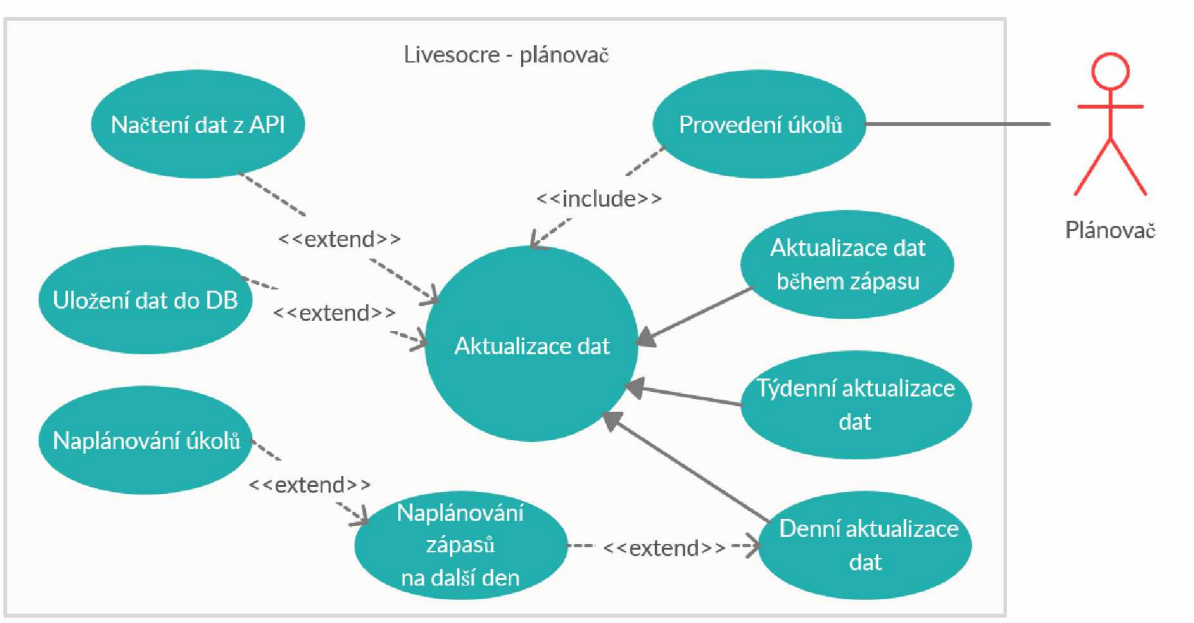
Obrázek 11: Případy užití modulu soutěže[22]

Modul plánování

Následující modul je již součástí cloudové části a zabývá se plánováním úkolů (viz obrázek 12). Jedná se o modul, který má na starost pravidelné načítání dat z API a aktualizací dat v cloudové NoSQL databázi. Na starosti ho má plánovač. Tento modul tedy obstarává:

- provedení úkolů,
- denní aktualizaci dat,
- týdenní aktualizaci dat,
- a aktualizace dat během zápasu.

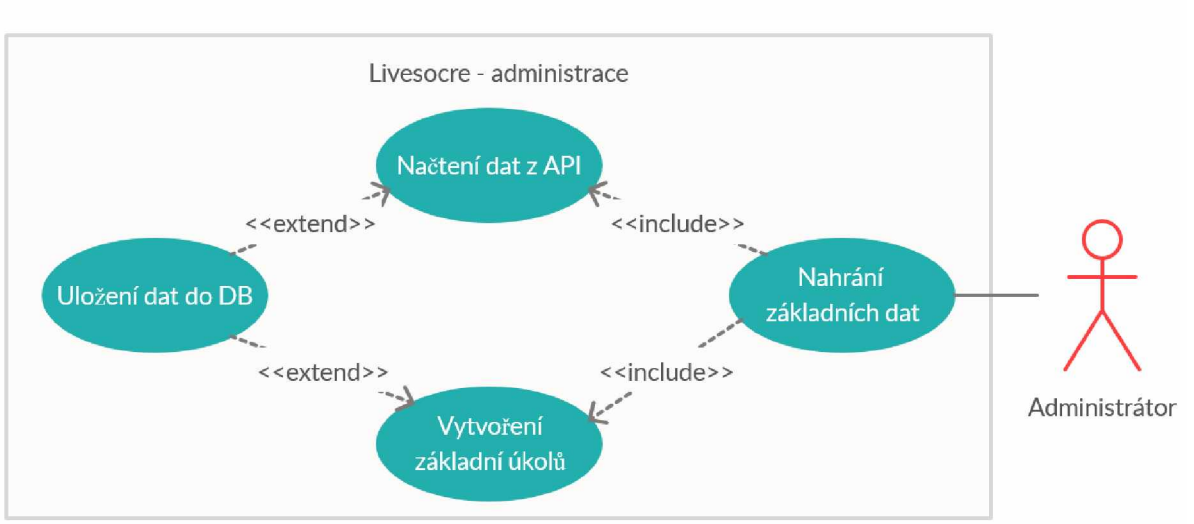
[22] [vlastní]



Obrázek 12: Případy užití modulu úkoly[23]

Modul administrace

Posledním modulem je modul administrace (viz obrázek 13). Ten je důležitý v momentě, kdy probíhá základní inicializace dat nového Firebase cloudu. Administrátor v takové chvíli spustí funkcí, pomocí které načte základní data z API do databáze a vytvoří úkoly pro denní a týdenní aktualizaci dat.



Obrázek 13: Případy užití modulu administrace[24]

[23] [vlastní]
[24] [vlastní]

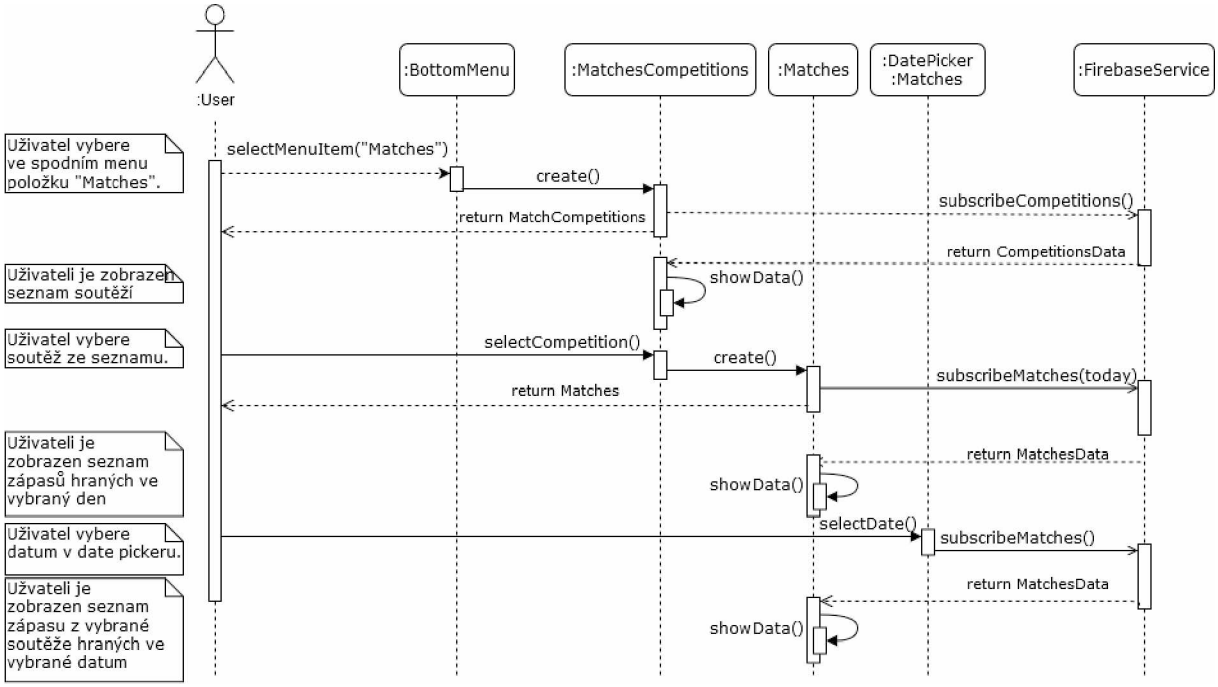
6.4 Realizace případů užití

Dále bylo zapotřebí vytvořit realizace případů užití. V této podkapitole tedy jsou rozebrány scénáře dvou vybraných případů užití.

Kontrola určitých zápasů

Pokud chce uživatel zkontrolovat zápasy určité ligy v určitý den, musí se postupně projit těmito kroky (zjednodušený sekvenční diagram pro ten scénář viz obrázek 14):

1. Uživatel vybere ve spodním menu položku „Matches“.
    a. Vytvoří se (nebo se vezme existující) fragment se seznamem soutěží.
    b. Fragment zavolá svůj view model, aby začal odebírat data pro seznam soutěží.
    c. View model přihlásí odběr dat pro seznam soutěží.
    d. Uživateli je zobrazen fragment.
    e. Do fragmentu je načten seznam soutěží
2. Uživatel vybere soutěž, pro kterou chce zobrazit seznam zápasů.
    a. Vytvoří se (nebo se vezme existující) fragment se seznamem zápasů.
    b. Fragment zavolá svůj view model, aby začal odebírat data pro zápasy vybrané soutěže.
    c. View model přihlásí odběr zápasů vybrané soutěže.
    d. Uživateli je zobrazen fragment.
    e. Do fragmentu je načten seznam zápasů seznam zápasů vybrané soutěže, které se odehrávají dnes (či v nejbližším datu, kdy se hraje).
3. Uživatel v DatePickeru vybere datum, pro které chce zobrazit seznam zápasů vybrané soutěže.
    a. Fragment zavolá svůj view model, aby změnil odběr dat a začal odebírat data pro vybrané datum.
    b. Do fragmentu je zobrazen seznam zápasů vybrané soutěže, které se odehrávají ve vybraném datu.



Obrázek 14: Realizace případu užití "Kontrola určitých zápasů"[25]

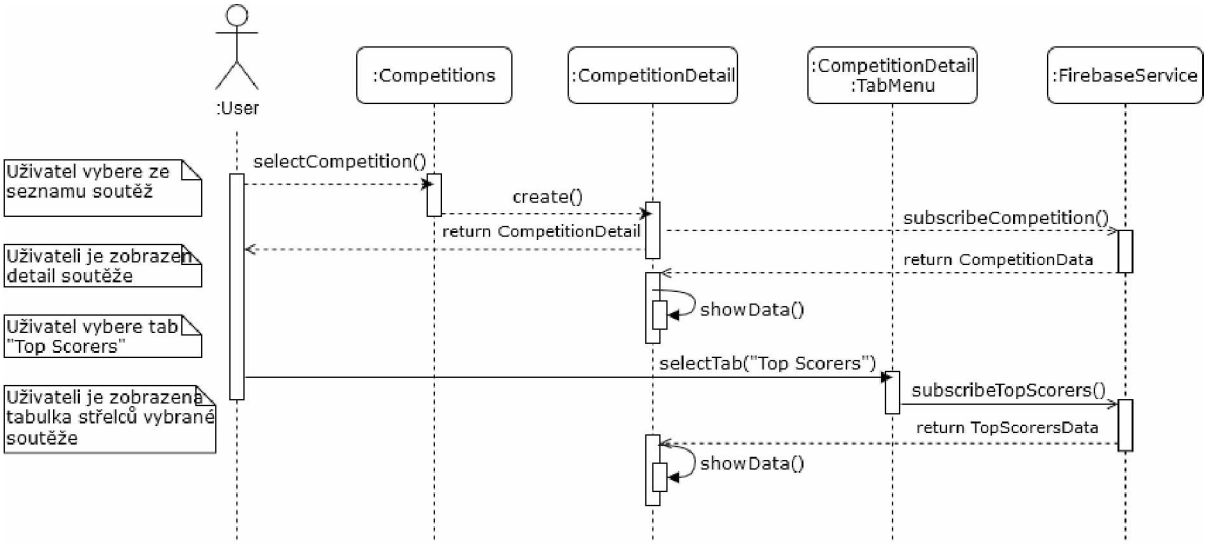
Kontrola tabulky střelců

Pokud chce uživatel zkontrolovat tabulku střelců dané soutěž, musí postupně projít těmito kroky (zjednodušený sekvenční diagram pro tento scénář viz obrázek 14).

1. Uživatel vybere soutěž, pro kterou chce zobrazit tabulku střelců.
    a. Vytvoří se (nebo se vezme existující) fragment s detailem soutěže.
        i. Vytvoří se (nebo se vezme existující) fragment pro události zápasu.
        ii. Fragment zavolá svůj view model, aby začal odebírat data pro danou soutěž.
        iii. View model přihlásí odběru dat pro vybranou soutěž.
        iv. Fragment je zobrazen do fragment s detailem soutěže.
        v. Do fragmentu je načtena tabulka soutěže.
    b. Fragment zavolá svůj view model, aby začal odebírat data pro danou soutěž.
    c. View model přihlásí odběr zápasů vybrané soutěže.
    d. Uživateli je zobrazen fragment.
    e. Do fragmentu jsou načteny informace o soutěži.
2. Uživatel vybere tab „Top Scorers“
    a. Vytvoří se (nebo se vezme existující) fragment pro tabulku.
    b. Fragment zavolá svůj view model, aby začal odebírat data pro danou soutěž.

[25] [vlastní]

c. View model přihlásí odběru dat pro vybranou soutěž.
d. Fragment je zobrazen do fragmentu s detailem soutěže.
e. Do fragmentu jsou načtena tabulka nejlepších střelců.



Obrázek 15: Realizace případu užití "Kontrola tabulky střelců soutěže"[26]

6.5 Architektura

Před začátkem vývoje bylo potřeba s ohledem na integraci cloudové a klientské části stanovit architekturu systému. V této podkapitole je popsána architektura celé systému a detail klientské aplikace.

Celý systém

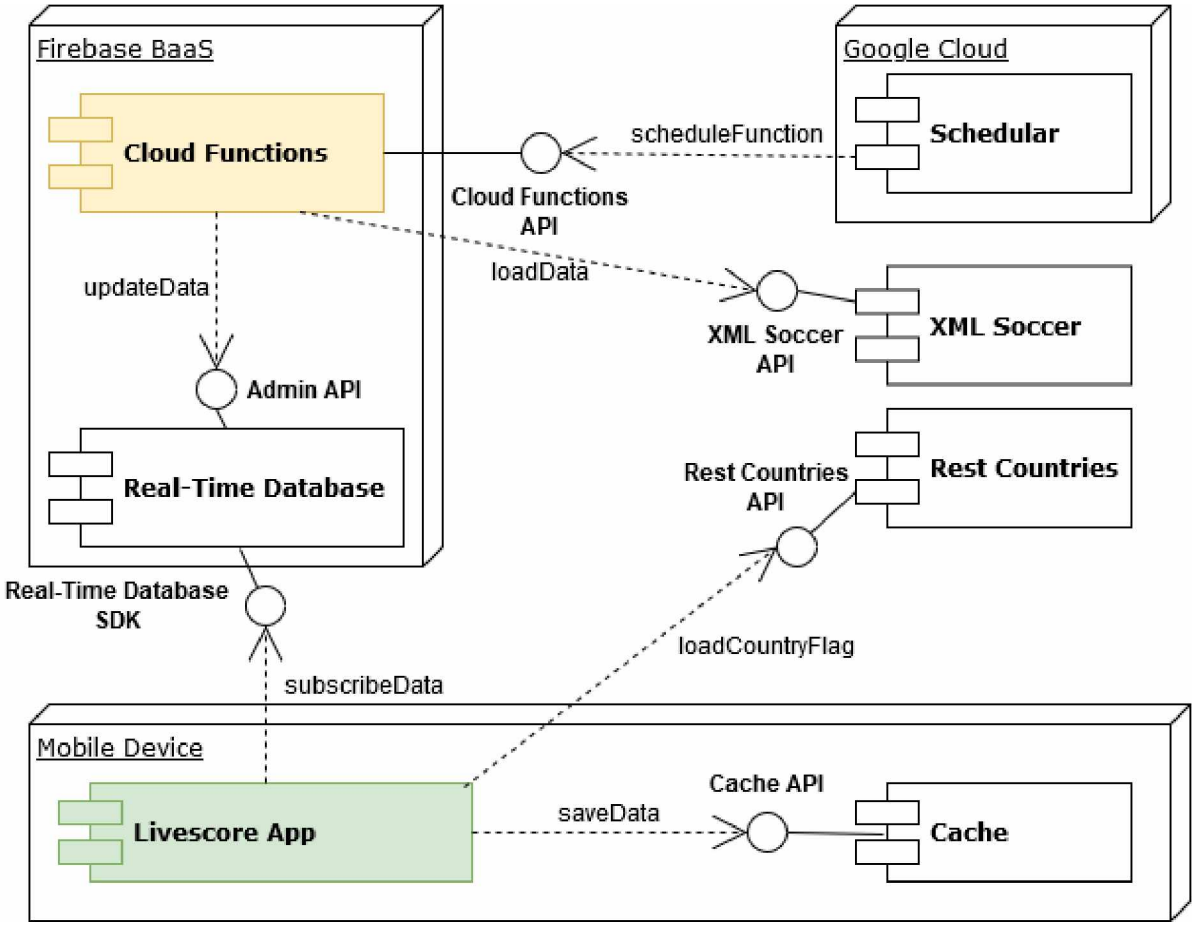
Jak již bylo uvedeno v předchozích podkapitolách, demonstrační aplikace se skládá z klientské a cloudové části. Jako klientská část je definována **mobilní aplikace pro platformu Android** a cloudovou částí jsou **Cloud Functions**, které jsou součástí cloudové služby Firebase.

O veškeré ukládání a aktualizaci dat v databázi se stará plánovač, který každou minutu kontroluje, jestli je třeba spustit nějaký úkol. Součástí každého úkolu bývá načtení dat z **XML Soccer API**[27], uložení dat do databáze (prostřednictvím **Admin API**[28]) a případně naplánování dalších úkolů.

[26] [vlastní]
[27] https://www.xmlsoccer.com/
[28] https://firebase.google.com/docs/reference/admin

Klientská část aplikace pak má za úkol odebírat dat z databáze prostřednictvím **Real-Time Database SDK**[29]. Aby bylo možné odebírat data z databáze, musí nejprve proběhnout autentizace. Data pak jsou případně ukládaná do cache paměti pro rychlejší načítání a pro případ, že by bylo klientské zařízení v režimu offline. Mobilní aplikaci je dále v některých momentech zobrazovat vlajky zemí, které jsou načítány prostřednictvím **Rest Countries API**[30].

Architektura celé aplikace je také zobrazena prostřednictvím diagramu komponent na obrázku 16.



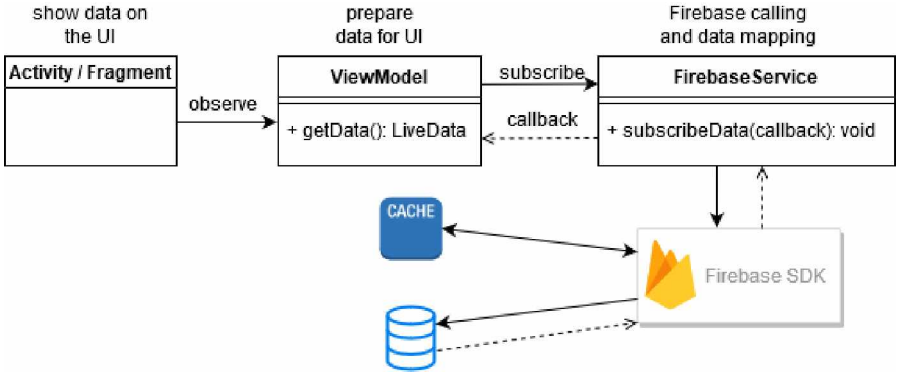
Obrázek 16: Architektura praktické části[31]

[29] https://firebase.google.com/docs/database/android/start
[30] https://restcountries.eu/
[31] [vlastní]: zelenou barvou je znázorněná klientská část, žlutou barvou pak cloudová část.

Klientská část

Klientská část praktické části práce je postavena na architektuře MVVM[32] (viz obrázek 17). Mimo přihlašovací obrazovku běží celá aplikace v jedné aktivitě[33], ve které se pouze postupně mění fragmenty[34], které se starají o zobrazení dat. Navigace mezi fragmenty je pak zajištěna pomocí komponent navigace[35].

Každá aktivita a fragment má pak svůj vlastní view model, od kterého získává data k zobrazení. View model určuje, která data se mají odebírat, a připravuje data pro zobrazení v aktivitě/fragmentu. Třída FirebaseService se pak stará o volání Firebase SDK a o mapování příchozích dat na POJO objekty, se kterými aplikace pracuje.



Obrázek 17: Architektura klientské části[36]

6.6 Obrazovky klientské části

Tato podkapitola je věnovaná konečné podobě klientské části práce, jsou zde postupně zobrazeny jednotlivé obrazovky aplikace spolu se stručným popisem možných činností uvnitř nich. Z důvodu úspory místa jsou většinou spojeny dvě obrazovky do jednoho obrázku a označeny pomocí písmen.

Autentizace

Uživatel se může aplikaci používat anonymně nebo se do ní může autentizovat pomocí e-mailu či Googlu účtu (viz obrázek 18). Autentizace mu poskytne možnost sdílení seznamu odebíraných zápasů a soutěží napříč zařízeními. Všechny metody autentizace obstarává Firebase a konečná podoba autentizační obrazovky je vygenerována na základě povolených metod autentizace (viz [37]).

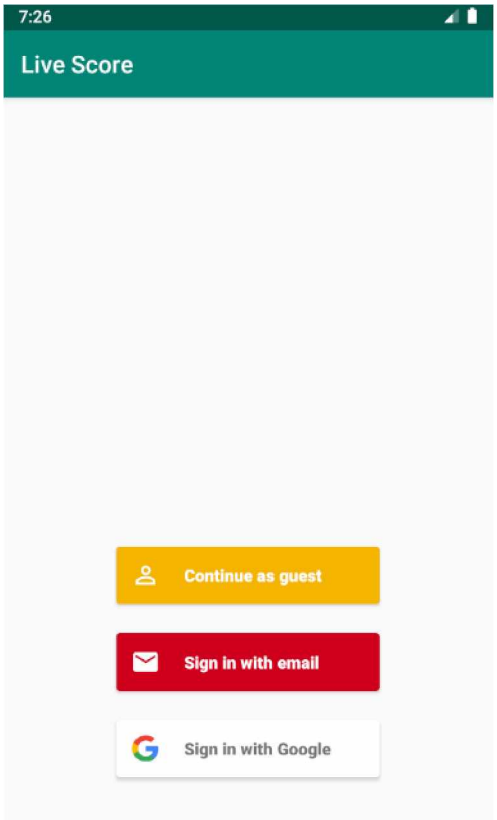
---

[32] https://developer.android.com/jetpack/docs/guide
[33] https://developer.android.com/reference/android/app/Activity
[34] https://developer.android.com/guide/components/fragments
[35] https://developer.android.com/guide/navigation
[36] [vlastní]
[37] https://firebase.google.com/docs/auth/android/firebaseui

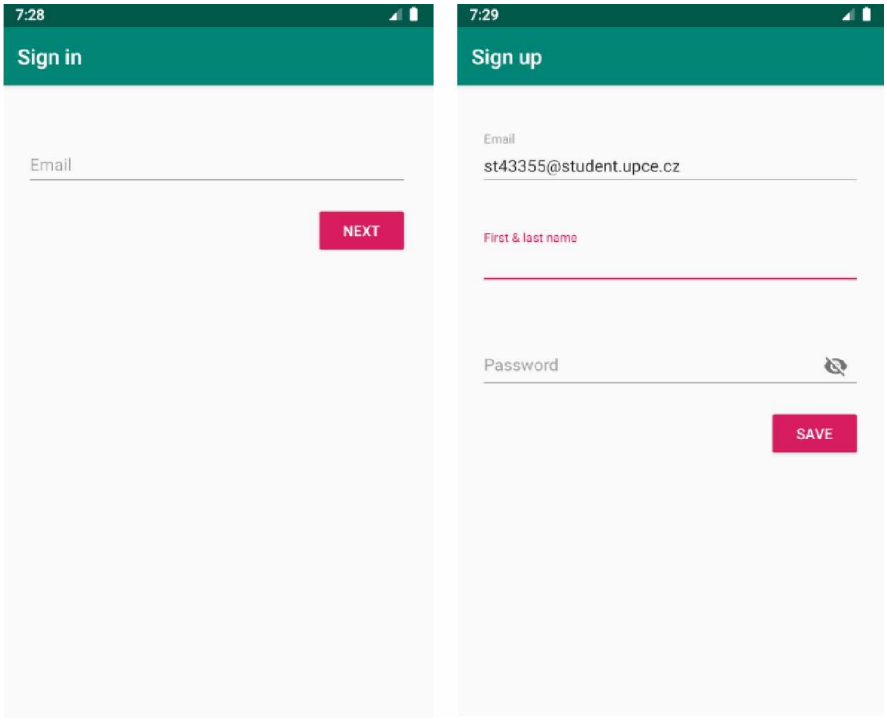
Při autentizaci pomocí e-mailu je uživatel nejprve vyzván k zadání e-mailu, jména a hesla (viz obrázek 19). Při opětovném přihlášení pak uživatel zadá pouze e-mail a heslo (viz obrázek 20a). Pokud uživatel zvolí přihlášení pomocí Google účtu, je přesměrován na přihlašovací obrazovku Googlu (viz obrázek 20b).

Hlavní obrazovky

Po projití autentizací se uživatel dostane do hlavního prostředí aplikace, které je tvořeno horním panelem, spodním hlavním menu a obsahem. Hlavními obrazovkami jsou ty, které zobrazují seznamy soutěží a zápasů (viz obrázek 21). Po vybrání položky se uživateli zobrazí její detail. Dále lze přidat položku do seznamu sledovaných zápasů či soutěží, pokud uživatel klepne na hvězdu. U seznamu zápasů lze také vybrat datum, pro které mají být zápasy zobrazeny.

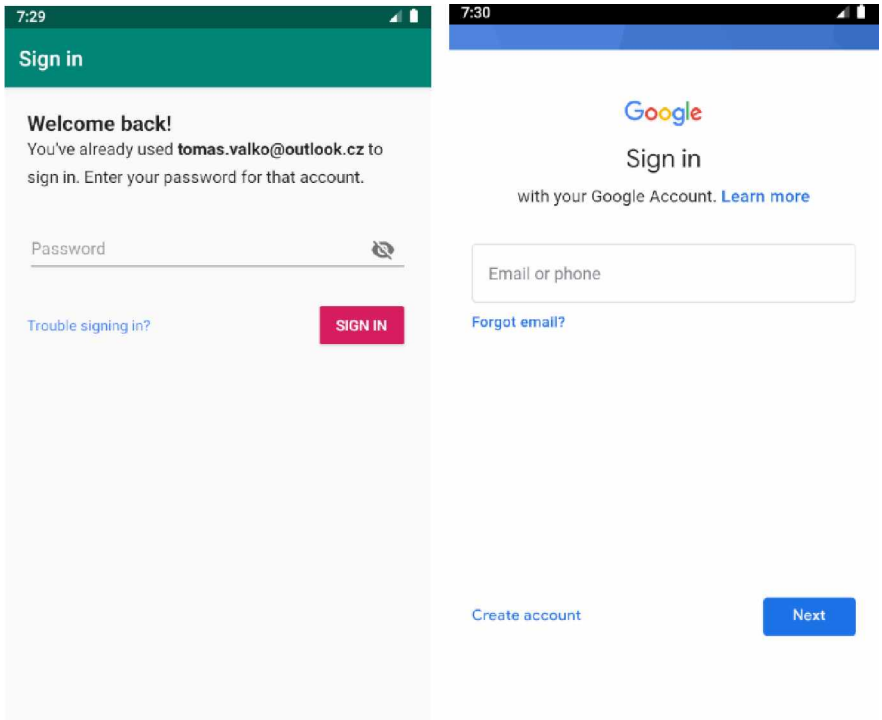


Obrázek 18: Firebase přihlašovací obrazovka



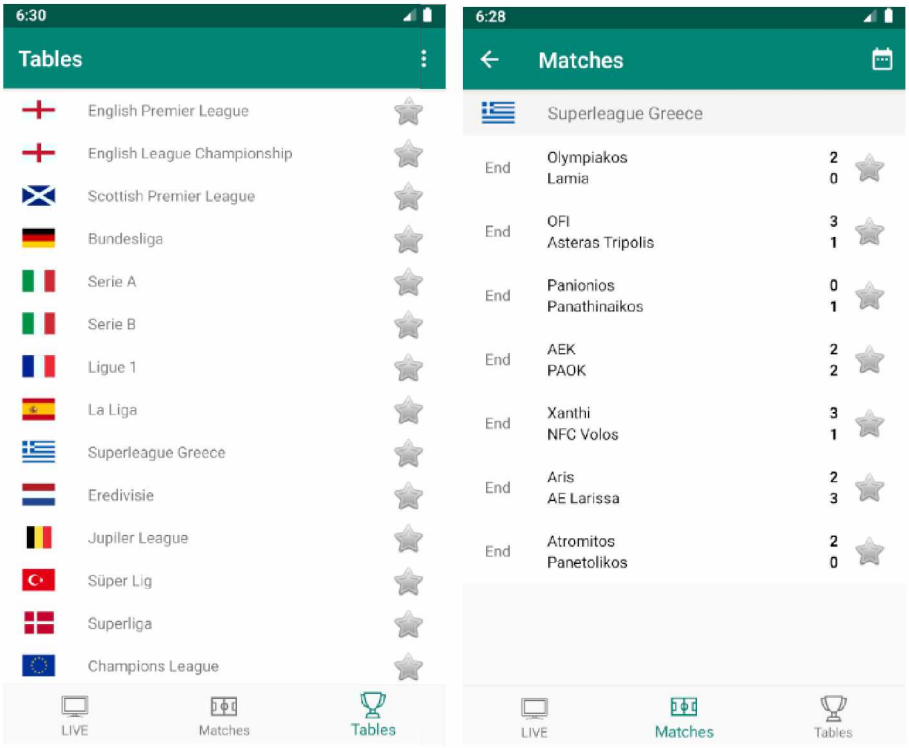
a) b)

Obrázek 19: Firebase přihlašovací obrazovka – e-mail



a) b)

Obrázek 20: Firebase přihlašovací obrazovka – e-mail a Google účet



a) b)

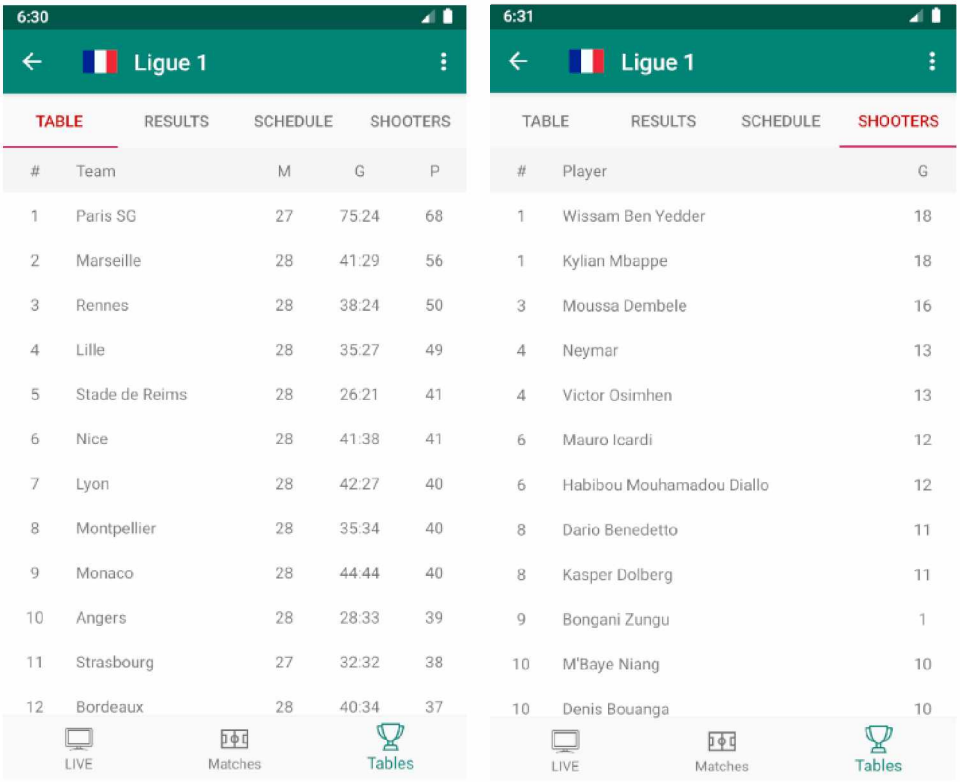
Obrázek 21: Obrazovky pro seznam zápasů a seznam soutěží

Obrazovky detailů

Dále následují obrazovky zobrazující detaily soutěží a zápasů. Ty jsou primárně tvořeny obsahem zobrazovaným v tabech.

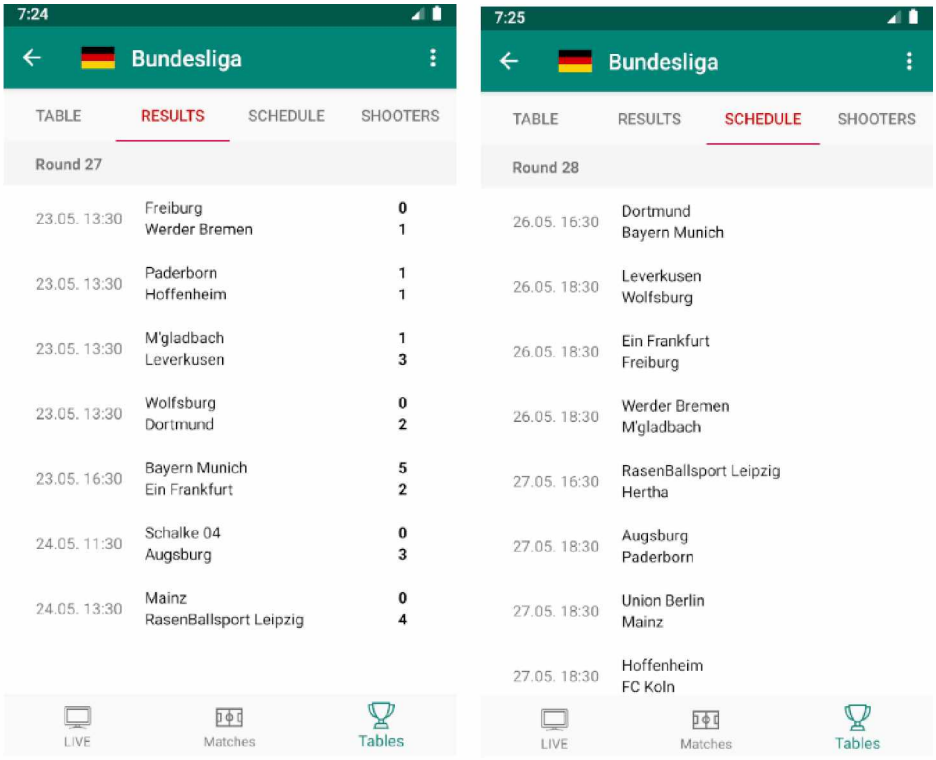
U soutěže je nejprve v horním panelu zobrazena vlajka země soutěže a její název. Dále si lze na jednotlivých tabech zobrazit tabulku soutěže (obr. 22a), tabulku střelců (obr. 22b), poslední výsledky (obr. 23a) a následující plánované zápasy (obr. 23b).

U zápasu lze nejprve vidět názvy týmů, datum, skóre a případně aktuální čas zápasu. Dále si lze v jednotlivých tabech zobrazit události zápasu a statistiky.



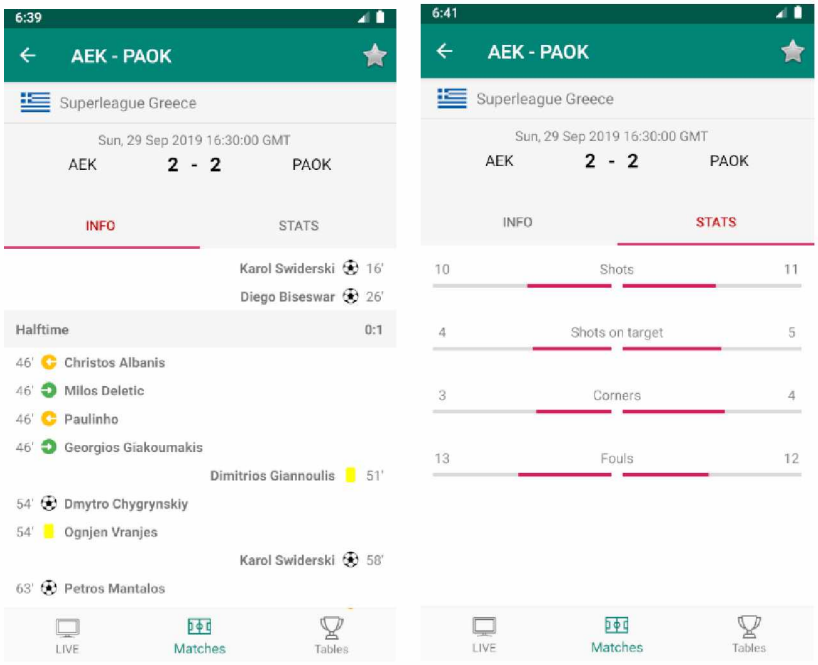
a) b)

Obrázek 22: Obrazovky pro tabulku soutěže a tabulku střelců soutěže



a) b)

Obrázek 23: Obrazovky pro výsledky soutěže a plánované zápasy



a) b)

Obrázek 24: Obrazovky pro události zápasu a statistiky zápasu

6.7 Integrace pomocí Firebase Assistant

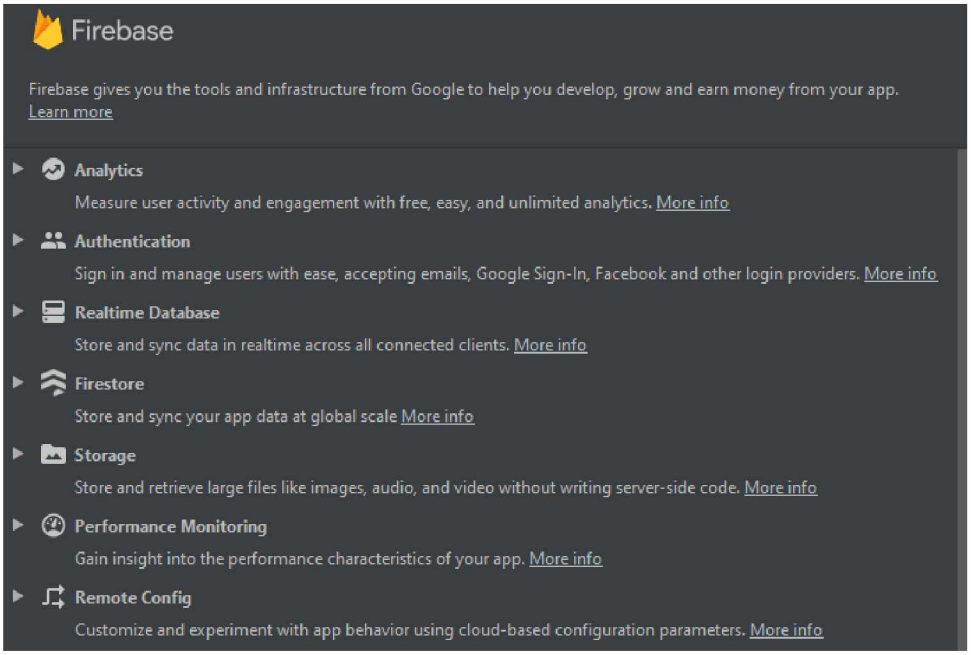
Nejjednodušší cestou, jak integrovat funkce, které jsou součástí cloudové služby Firebase, do Android aplikace, je použití nástroje Firebase Assistant (viz obrázek 18). Tento nástroj je integrován jako součást vývojového prostředí Android Studio a poskytuje interaktivní cestu pro integraci Firebase.

K otevření tohoto nástroje je potřeba v horním menu Android Studia zvolit položku Tools a poté položku Firebase. Následně se na pravé straně prostředí objeví lišta s otevřeným nástrojem Firebase Assistant.

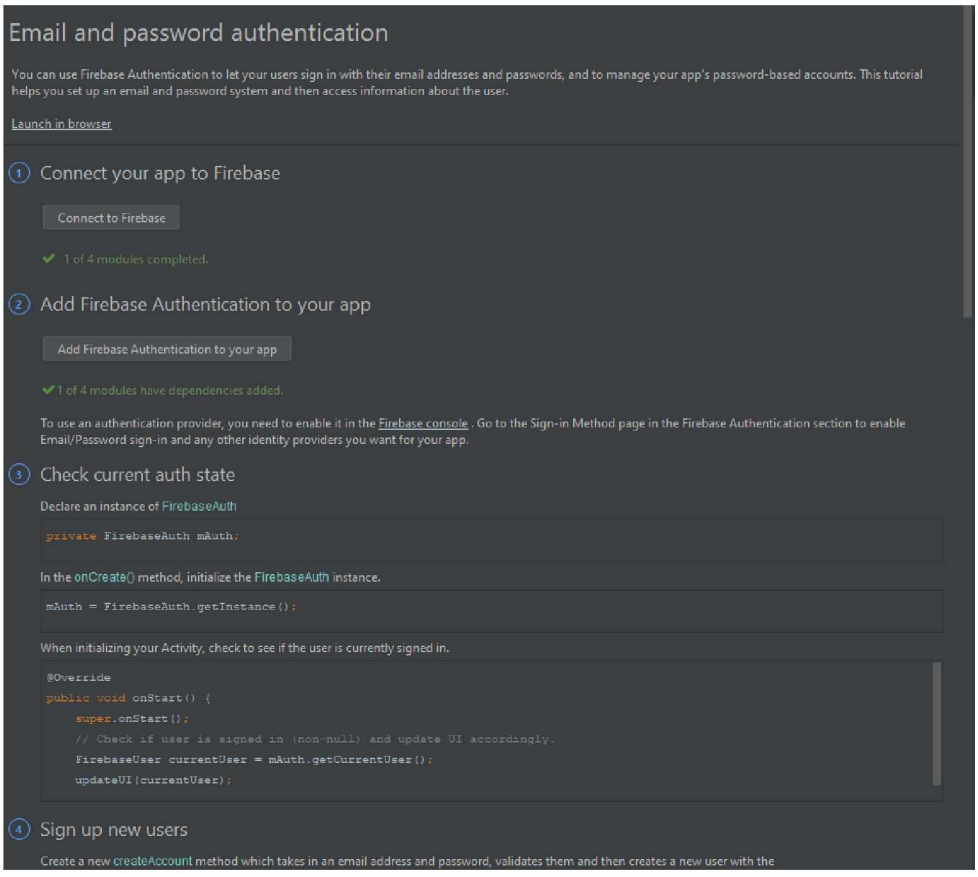
V nástroji si vývojář zvolí nejprve část Firebase, kterou chce do Android aplikace integrovat, a následně lišta zobrazí obsah věnované dané části. Ten obsahuje seznam kroků, které je potřeba provést k integraci této části Firebase do Android aplikace (viz obrázek 19).

První položka vždy slouží k připojení Android aplikace ke službě Firebase. Po jejím vybrání se otevře panel v prohlížeči s webovou Firebase konzolí, kde se vývojář přihlásí, zvolí daný projekt a připojení proběhne automaticky.

Po vybrání druhé položky se zobrazí závislosti, které je potřeba přidat do Android aplikace pro možnost. Po přidání závislostí je pak možné v aplikaci používat SDK pro danou část Firebase.



Obrázek 25: Prostředí nástroje Firebase Assistant



Obrázek 26: Prostředí nástroje Firebase Assistant zobrazující část věnovanou autentizaci

Další položky pak již jen zobrazují příklady kódu pro integraci konkrétních funkcí dané části Firebase. Při použití tohoto nástroje se tedy integrace částí Firebase stává jednoduchou a interaktivní.

Závěr

Hlavním cílem této diplomové práce bylo prozkoumání možností a vlastností, které nabízí Real-Time databáze pro využití v praxi. To vše pak mělo být demonstrováno na mobilní aplikaci pro platformu Android, která bude tuto databázi využívat.

V teoretické části práce byly postupně rozebrány pojmy důležité k vysvětlení problematiky Real-Time databází. Nejprve proběhlo srovnání databázových systémů, kdy byly vyjmenovány výhody a nevýhody relačních, NoSQL a hybridních DBMS. Na to navázalo představení NoSQL databází, kdy byly rozebrány především základní principy těchto databází a představeny jednotlivé typy. Dále bylo popsáno fungování Real-Time databází, rozebrán rozhodovací strom Real-Time dotazu a představeny populární řešení těchto databází. Nakonec byla představena forma cloudové služby (BaaS), pomocí které bývá Real-Time databáze poskytována, a jedno konkrétní BaaS řešení (Google Firebase), které bylo následně použito při implementaci demonstrační aplikace.

V praktické části práce byla teorie demonstrována pomocí mobilní aplikace pro platformu Android zabývající se sledováním výsledků a událostí fotbalových utkání. Klientská část aplikace byla implementována v programovacím jazyce Java s pomocí architektury MVVM. Jednalo se především o jednotlivé obrazovky, autentizaci, odebírání dat z cloudové databáze včetně jejich mapování do POJO objektů, mockování dat na začátku vývoje a načítání SVG obrázků vlajek z webového API. Cloudová část pak byla zpracována pomocí Cloud Functions, které jsou součástí Firebase BaaS, v programovacím jazyce JavaScript pro platformu Node.js. Zde bylo využito plánovací funkce, která se spouští každou minutu a stará se o načítání data, plánování dalších úkolů a ukládání dat do databáze. Integrace Firebase v klientské části proběhla pomocí nástroje Firebase Assistant integrovaném ve vývojovém prostředí Android Studio.

Vývoj praktické části práce byl v práci postupně popsán. Nejprve byla stanovena kritéria pro demonstrační aplikaci, podle kterých bylo vymyšleno řešení vhodné pro demonstraci řešené teorie. Dále byly definovány jednotlivé případy užití pro každý modul aplikace spolu se scénáři některých z nich. Poté byla vytvořena architektura celkové aplikace s ohledem na integraci klientské a cloudové části. Dále byla navržena architektura klientské části s ohledem na odběr, mapování, transformaci a zobrazování dat z databáze. Nakonec byly představeny jednotlivé obrazovky aplikace.

Dále rozvíjet by šla hlavně demonstrační aplikace, kde by mohly být využity například některé další mechanismy zabezpečení přístupu k datům, přidány další metody autentizace, přidán chat pro další demonstraci fungování Real-Time databáze či rozšíření aplikace o notifikace.

Seznam použitých zdrojů

[1] What does ACID mean in Database Systems? *Database.Guide* [online]. [cit. 2020-05-24]. Dostupné z: https://database.guide/what-is-acid-in-databases/

[2] Ajax. MDN web docs [online]. [cit. 2020-05-24]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

[3] Application Domain. *ScienceDirect* [online]. [cit. 2020-05-24]. Dostupné z: https://www.sciencedirect.com/topics/computer-science/application-domain

[4] LILE. Journal of systems management. *Journal of systems management*. 1993, **44**(12). ISSN 0022-4839.

[5] HUB, Miloslav. Autentizace v přímém bankovnictví: Význam autentizace v přímém bankovnictví. *Scientific papers of the University of Pardubice: Series D Faculty of Economics and Administration* [online]. Pardubice: Ústav systémového inženýrství a informatiky, FES, Univerzita Pardubice, 2003, 8 roč., s. 39-43 [cit. 2017-04-14]. ISSN 1211-555X. Dostupné z: http://dspace.upce.cz/bitstream/handle/10195/32263/CL404.pdf

[6] FLAMÍK, Martin. *Bezpečnost v elektronickém bankovnictví*. Pardubice, 2010. Bakalářská práce. Univerzita Pardubice. Fakulta ekonomicko-správní. Ústav systémového inženýrství a informatiky.

[7] Hashovací tabulka. *ITnetwork.cz* [online]. Praha: David Čápka, 2013 [cit. 2020-05-25]. Dostupné z: https://www.itnetwork.cz/navrh/algoritmy/algoritmy-vyhledavani/algoritmus-vyhledavani-hashovaci-tabulka

[8] History of Linux explained. *Everything.explained.today* [online]. ©2009-2016 [cit. 2017-04-05]. Dostupné z: http://everything.explained.today/History_of_Linux

[9] TORRES, Alexandre, Renata GALANTE, Marcelo S. PIMENTA a Alexandre Jonatan B. MARTINS. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and software technology* [online]. listopad 2017, roč. 82, s. 1-18 [cit. 2017-04-22]. ISSN 0950-5849. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0950584916301859

[10] ARNOŠT, Pavel. Co je to Open Source Software. *Root.cz* [online]. Internet Info, s.r.o., 22. srpna 2001 [cit. 2017-04-05]. ISSN 1212-8309. Dostupné z: https://www.root.cz/clanky/co-je-to-open-source-software

[11] Offline. *It-slovnik.cz* [online]. ©2008-2017 [cit. 2017-04-05]. Dostupné z: http://itslovnik.cz/pojem/offline

[12] SQL. In: *Britannica Academic* [online]. Encyclopædia Britannica, 15. ledna 2009 [cit. 2017-04-05]. Dostupné z: http://academic.eb.com/levels/collegiate/article/SQL/438617#

[13] Srovnání distribuovaných "NoSQL" databází s důrazem na výkon a škálovatelnost[online]. Praha, 2012 [cit. 2013-04-05]. Dostupné z: http://isis.vse.cz/zp/111109. DIPLOMOVÁ PRÁCE. Vysoká škola ekonomická v Praze.

[14] Špagetový kód. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 14. května 2014, aktualizováno 5. října 2016 [cit. 2017-04-15]. Dostupné z: https://cs.wikipedia.org/wiki/Špagetový_kód

[15] SARBJEET, Singh. Software Testing. *International journal of advanced research in computer science* [online]. [b.j.], 1. října 2010, roč. 1, č. 3, s. 403-406 [cit. 2017-04-21]. ISSN 0976-5697. Dostupné z: http://search.proquest.com/docview/1443701695/F3BFDD3F196649F6PQ/1?accountid=17239

[16] HLAVA, Tomáš. Fáze a úrovně provádění testů. *Testování softwaru: Portál zabývající se problematikou ověřování kvality software. Manuální i automatizované testování.* [online]. 21. srpna 2011 [cit. 2017-04-21]. Dostupné z: http://testovanisoftwaru.cz/tag/akceptacnitestovani

[17] HARMAN, Mark. Why Source Code Analysis and Manipulation Will Always Be Important. In: SCAM 2010: *International Working Conference on Source Code Analyisis and Manipulation*, Timişoara 12.-13. listopadu 2010 [online]. Timişoara: [b.j.], 2010 [cit. 2017-04-12]. Dostupné z: http://www0.cs.ucl.ac.uk/staff/M.Harman/scam10.pdf

[18] HOLUBOVÁ, Irena, Jiří KOSEK, Karel MINAŘÍK a David NOVÁK. *Big Data a NoSQL databáze*. Praha: Grada, 2015. Profesionál. ISBN 978-80-247-5466-6.

[19] CONOLLY, Thomas, Carolyn E. BEGG a Richard HOLOWCZAK. *Mistrovství - databáze: profesionální průvodce tvorbou efektivních databází*. Brno: Computer Press, 2009. ISBN 978-80-251-2328-7.

[20] NoSQL vs SQL — Which Database Type is Better For Big Data Applications. *Analytics India Magazine* [online]. Bengaluru: Analytics India Magazine, 2017 [cit. 2019-02-26]. Dostupné z: www.analyticsindiamag.com/nosql-vs-sql-database-type-better-big-data-applications

[21] The SQL vs NoSQL Difference: MySQL vs MongoDB. *XplentyBlog* [online]. San Francisco: medium.com, 2017 [cit. 2019-02-26]. Dostupné z: medium.com/xplenty-blog/the-sql-vs-nosql-difference-mysql-vs-mongodb-32c9980e67b2

[22] 10 things you should know about NoSQL databases. *TechRepublic* [online]. San Francisco: CBS Interactive, 2010 [cit. 2019-02-26]. Dostupné z: www.techrepublic.com/blog/10-things/10-things-you-should-know-about-nosql-databases

[23] Úvod do JSON. *Json.org* [online]. 200-? [cit. 2019-03-02]. Dostupné z: www.json.org/json-cz.htm

[24] FROZZA A. A., R. d. S. MELLO, F. d. S. d. COSTA, An Approach for Schema Extraction of JSON and Extended JSON Document Collections. In: *2018 IEEE International Conference on Information Reuse and Integration (IRI), Salt Lake City 6.-9. července* [online]. Salt Lake City, 2018 [cit. 2019-03-02]. Dostupný z: ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8424731

[25] *BSON* [online]. 2009 [cit. 2019-03-05]. Dostupné z: bsonspec.org

[26] ŘÁDA, Jan. *Aplikace NoSQL databází v oblasti podnikových informačních technologií*. Praha, 2017. Bakalářská. Unicorn College. Katedra informačních technologií.

[27] HOUŽVIČKA, Tomáš. *Aplikace NoSQL databází*. Praha, 2012. Diplomová. Bankovní institut vysoká škola Praha. Katedra matematiky, statistiky a informačních technologií.

[28] WINGERATH, W., N. RITTER a F. GESSERT. *Real-Time & Stream Data Management: Push-Based Data in Research & Practice*. 1. New York, NY: SpringerBriefs in Computer Science, 2018. ISBN 978-3-030-10554-9.

[29] WINGERATH, Wolfram. A Real-Time Database Survey: The Architecture of Meteor, RethinkDB, Parse & Firebase. *Baqend Blog* [online]. San Francisco: medium.com, 2017 [cit. 2019-03-18]. Dostupné z: https://medium.baqend.com/real-time-databases-explained-why-meteor-rethinkdb-parse-and-firebase-dont-scale-822ff87d2f87

[30] GREIF, Sacha. An Introduction To Latency Compensation. *Discover Meteor* [online]. Osaka & Melbourne: Tom Coleman and Sacha Greif, 2015 [cit. 2019-03-31]. Dostupné z: www.discovermeteor.com/blog/latency-compensation

[31] LARDINOIS, Frederic. Google Acquires Firebase To Help Developers Build Better Real-Time Apps. *TechCrunch: Startup and Technology News* [online]. Dublin: Verizon Media, 2014 [cit. 2019-04-01]. Dostupné z: techcrunch.com/2014/10/21/google-acquires-firebase-to-help-developers-build-better-realtime-apps/?guccounter=1&guce_referrer_us=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_cs=Xwbj3wQ1Uqc8q4ba7oSQwA

[32] MARTÍNEZ, Gabriela a Pablo LOPÉZ. *Real-Time Databases and Firebase* [online]. Brusel, 2018 [cit. 2019-04-01]. Dostupné z:

cs.ulb.ac.be/public/_media/teaching/infoh415/student_projects/2019/firebase.pdf. Studentský projekt. École Polytéchnique Bruxelles, Université Libre de Bruxelles.

[33] Scale with Multiple Databases. *Firebase* [online]. Mountain View: Google, 2019, 2019 [cit. 2019-04-01]. Dostupné z: firebase.google.com/docs/database/usage/sharding

[34] BAPTISTE, Jamin. Reasons Not To Use Firebase. *Crisp* [online]. Nantes: Crisp IM, ©2019, 2016 [cit. 2019-04-01]. Dostupné z: crisp.chat/blog/why-you-should-never-use-firebase-realtime-database

[35] MÁJSKÝ, Michal. *Analýza současných řešení Backend-as-a-Service pro vývoj mobilních a webových aplikací*. Praha, 2016. Diplomová. Katedra softwarového inženýrství, Fakulta informačních technologií, ČVUT.

[36] TŘASKALÍK, Michal. *Využití realtime databází v Ionic frameworku*. Zlín, 2018. Diplomová. Fakulta aplikované informatiky, Univerzita Tomáš Bati ve Zlíně.

[37] LANE, Kin. *Overview Of The Backend as a Service (BaaS) Space* [online]. [cit. 2019-04-10]. Dostupné z: www.integrove.com/wp-content/uploads/2014/11/api-evangelist-baas-whitepaper.pdf

[38] Firebase Guides. *Firebase* [online]. Google [cit. 2020-05-25]. Dostupné z: https://firebase.google.com/docs/guides

Přílohy