Univerzita Hradec Králové

Fakulta informatiky a managementu

DIPLOMOVÁ PRÁCE

2017

Jan Macháček

Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informačních technologií

Synchronizace dat aplikací pro mobilní zařízení

Synchronizace dat mezi mobilním zařízením a vzdáleným serverem s využitím REST a OAuth

Diplomová práce

Autor: Bc. Jan Macháček

Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Pavel Kříž, Ph.D.

Pardubice duben 2017

Prohlášení:

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

V Pardubicích dne 2. dubna 2017 Jan Macháček

Anotace

Hlavním cílem práce je seznámit čtenáře s oblastí synchronizace dat mobilních zařízení. Na začátku práce jsou popsány způsoby synchronizace a nástroje, které se dají pro tento účel použít. Dále je podrobně popsán použitý synchronizační algoritmus, REST API sloužící pro komunikaci mezi serverem a klientskou aplikací a protokol OAuth zajišťující bezpečné ověřování identity uživatele při přístupu k privátním datům prostřednictvím REST API. Další část obsahuje podrobný popis komponent a programových prostředků platformy Android, které lze použít pro vytvoření a správu účtu a pro synchronizaci dat.

Pro demonstraci funkčnosti popsaných algoritmů a postupů byl navržen a vytvořen jednoduchý systém skládající se z aplikace pro OS Android, serveru pro ukládání dat a webového klienta pro registraci uživatele a pro přístup k privátním datům. Uživatel má možnost využít svůj Google účet pro přihlášení do mobilní i webové aplikace.

Součástí práce je CD se zdrojovými kódy mobilní aplikace, serveru a webového klienta.

Annotation

Data synchronization between mobile device and remote server using REST and OAuth

The main objective of the thesis is to introduce the area of data synchronization in mobile devices. At the beginning, the thesis describes synchronization methods and tools that can be used for this purpose. This is followed by a detailed descriptions of the synchronization algorithm REST API used for communication between the server and the client application and the OAuth protocol that ensures safe user authentication when accessing the private data through the REST API. The next part contains a detailed description of the components and software means of the Android platform that can be used for creation and management of the account and data synchronization.

To demonstrate the functionality of the described algorithms and procedures, a simple system was designed and created consisting of an application for the Android OS, a server for data storage and a Web client for user registration and access to private data. The user has the option to use their Google account to log into the mobile and Web applications.

The thesis includes a CD with source codes of the mobile application, server and Web client.

Obsah

1. Úvod ... 1
2. Cíle práce ... 2
   - 2.1. Online aktualizace databáze astronomických objektů ... 2
   - 2.2. Astronomický deník ... 2
   - 2.3. Webové rozraní ... 2
   - 2.4. Údaje o počasí ... 3
   - 2.5. Přihlášení pomocí sociálních sítí ... 3
3. Synchronizace dat mobilních zařízení ... 4
   - 3.1. Dostupná řešení ... 5
   - 3.2. mBaaS ... 5
   - 3.3. Vlastní řešení ... 6
   - 3.4. Typy synchronizace ... 7
   - 3.5. Algoritmus synchronizace ... 8
   - 3.5.1. Vytváření GUID ... 9
   - 3.5.2. Counter ... 10
   - 3.5.3. Řešení konfliktů ... 10
   - 3.5.4. Testování ... 11
   - 3.6. API ... 11
   - 3.6.1. Soap ... 12
   - 3.6.2. REST ... 12
   - 3.7. OAuth ... 14
   - 3.7.1. Role ... 14
   - 3.7.2. Tokeny ... 15
   - 3.7.3. Způsoby autorizace ... 15
4. Implementace na straně serveru ... 18
   - 4.1. Server ... 18
   - 4.1.1. REST ... 18
   - 4.1.2. OAuth ... 19
   - 4.2. API ... 20
   - 4.2.1. Autentizace klienta ... 20
   - 4.2.2. Authorizační token ... 21
   - 4.2.3. Odmítnutí požadavku serverem ... 22

4.2.4. Refresh token ... 22
4.3. Komunikace se serverem ... 22
4.4. Získání aktuálních dat pro katalog astronomických objektů ... 23
5. Implementace v prostředí Android ... 25
5.1. Komunikace se serverem ... 25
5.1.1. Internetové spojením ... 25
5.1.2. Asynchronní procesy ... 26
5.1.3. Odeslání HTTP požadavku ... 28
5.1.4. Zpracování odpovědi serveru ... 28
5.2. Autentizace ... 29
5.2.1. Získání access_tokenu ... 29
5.2.2. Získání dat pomocí access_tokenu ... 30
5.2.3. Získání nového access_tokenu s použítím refresh_tokenu ... 30
5.3. Synchronizace dat ... 31
5.3.1. Autentizace ... 31
5.3.2. SyncAdapter ... 37
5.3.3. Průběh synchronizace ... 44
5.4. Vlastní Implementace synchronizace ... 45
5.4.1. Synchronizace katalogu objektů ... 45
5.4.2. Synchronizace deníku ... 46
5.4.3. Údaje o počasí ... 48
5.4.4. Přihlášení pomocí Googlu ... 49
5.4.5. Implementace na straně serveru ... 53
6. Webová aplikace ... 55
6.1. Popis a implementace ... 55
6.2. Přihlášení pomocí Googlu ... 55
7. Vývoj a testování ... 57
8. Závěr a doporučení ... 58
9. Seznam použitých zdrojů ... 60
Příloha 1: Obsah přiloženého CD ... 62

1. Úvod

Počet přenosných zařízení komunikujících přes internetovou síť na celém světě neustále roste. Stále více lidí používá hned několik takovýchto zařízení. Na každém zařízení může být nainstalováno několik aplikací komunikujících přes internet. Aplikace už nejsou vyvíjeny pouze velkými softwarovými společnostmi. Každý, kdo je vybavený počítačem má možnost vytvořit svojí aplikaci a nabídnout jí lidem na celém světě. Softwarové nástroje a dokumentace jsou poskytovány zdarma, a tak vzniká obrovské množství nejrůznějších aplikací.

S množstvím aplikací vzniká i obrovské množství dat. Současným trendem je sdílení dat v cloudových úložištích. Mobilní zařízení tak není primárním úložištěm, ale pouze nástrojem k jejich vytváření a používání. Uživatel má tak možnost používat pro přístup k datům například telefon, tablet, ale i webovou stránku na desktopovém počítači. Vývojář, který chce vytvořit takovou aplikaci, tak musí vytvořit celý systém skládající se za několika částí, umístěních na různých místech sítě a naprogramovaných v různých prostředích. Vzniká potřeba nějakým způsobem zajistit vzájemnou komunikaci jednotlivých částí.

Mobilní zařízení, na rozdíl od desktopů, mohou být používána téměř kdekoliv. Ačkoliv je v současné době připojení k mobilní síti poskytující data dostupné téměř kdekoliv, stále existují místa, která nejsou signálem pokryta. Práce v offline módu, muže být u aplikace, která ukládá data na vzdáleném serveru, velikou výhodou.

Privátní data bývají chráněna přihlašovacím jménem a heslem. Uživatel, používající několik různých aplikací, si musí pamatovat všechny své přihlašovací údaje. S rostoucím počtem uživatelů webových služeb je velice výhodné používat pro přihlašování účet některé ze sociálních sítí.

2. Cíle práce

Hlavním cílem práce je návrh a implementace jednoduchého systému pro synchronizaci dat mezi Android aplikací a webovým serverem, který umožní i práci v offline módu a lze se do něj přihlásit pomocí účtu sociální sítě.

Pro demonstraci dosažených výsledků bude použita aplikace AstroCatalag, která byla vytvořena jako součást bakalářské práce autora. AstroCatalog obsahuje databázi astronomických objektů a pomáhá uživateli s jejich vyhledáváním na noční obloze. Při reálném používání vznikl ze strany uživatelů požadavek na online aktualizaci seznamu objektů a na vytvoření deníku, který umožní ukládat záznamy o jednotlivých pozorováních včetně údajů o poloze a počasí.

2.1. Online aktualizace databáze astronomických objektů

Součástí aplikace AstroCatalog je soubor assets.txt. Obsahuje souřadnice, názvy a jiné podrobnosti astronomických objektů, které je možno pomocí aplikace vyhledávat. Po spuštění je nutno naplnit SQLite databázi zařízení daty, která budou parsována z tohoto souboru. Při prvním přístupu k databázi systém vytvoří strukturu tabulek a tu pomocí parseru naplní daty ze souboru assets.txt.

Nevýhodou tohoto řešení je fakt, že data obsahuje samotná aplikace. Pokud vznikne požadavek na aktualizaci dat, je třeba vydat novou verzi aplikace a tu poté nainstalovat. Neexistuje žádná možnost aktualizovat data přímo za běhu aplikace.

Součástí této práce bude doplnění aplikace o funkci, která zajistí on-line aktualizaci těchto dat. Tato aktualizace bude probíhat, pokud bude k dispozici internetové připojení. Aplikace se připojí k serveru, pokud bude k dispozici nová verze dat, dojde k jejich stažení a aktualizaci databáze objektů. To vše bude probíhat bez účasti uživatele, na pozadí, v pravidelných intervalech. Data budou aktualizována pouze jednosměrně, směrem od serveru k aplikaci. Aktuální data budou tedy vytvářena pouze na serveru.

2.2. Astronomický deník

Aplikace bude obsahovat astronomický deník. Uživatel bude mít možnost vytvářet si záznamy o svých pozorováních. Jednotlivé záznamy budou obsahovat časový interval kdy k pozorování došlo, GPS souřadnice, počasí, pozorované objekty atd. Tyto data budou ukládány do interní databáze zařízení. Aplikace bude deník synchronizovat se serverem. Synchronizace bude spouštěna pouze pokud bude k dispozici internetové spojení. Offline mód nebude mít žádný vliv na běh aplikace. Synchronizace bude obousměrná. Data budou upravována jak na serveru, tak v zařízení. Algoritmus musí řešit případné konflikty. Pokud dojde k úpravě dat na serveru, změny se při synchronizaci přenesou do zařízení. Stejně tak budou změny provedené v zařízení distribuovány na server.

2.3. Webové rozraní

Data z uloženého deníku budou přístupná prostřednictvím webové aplikace. Po přihlášení uživatele umožní zobrazit data vytvořená v mobilním zařízení. Uživatel bude moci záznamy upravovat, mazat, nebo vytvářet nové. Veškeré změny se musí po synchronizaci projevit i na mobilní aplikaci propojené registrovaným účtem.

2.4. Údaje o počasí

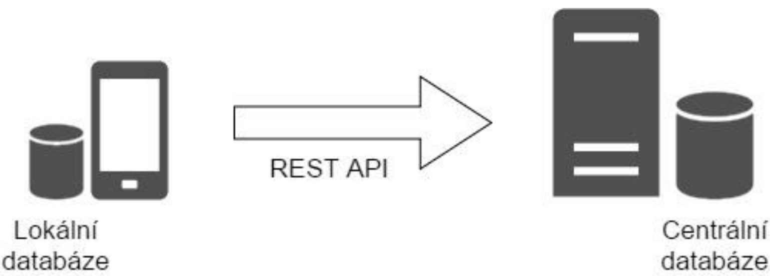
Aplikace bude obsahovat informace o aktuálním počasí v místě, kde se právě uživatel nalézá. Tyto informace budou stahovány z veřejného, volně přístupného API na openweathermap.com. Aktuální počasí bude zobrazováno v zařízení pouze, pokud bude právě k dispozici internetové spojení. Uživatel bude mít možnost uložit tyto informace přímo do záznamu v Astronomickém deníku.

2.5. Přihlášení pomocí sociálních sítí

U aplikací, které vyžadují online autentizaci je v současné době téměř standartní nabízet možnost přihlášení pomocí sociální sítě. V prostředí Android je toto obzvláště vhodné, protože téměř každý uživatel má v zařízení registrovaný emailový účet od Googlu. Ukázková aplikace bude mít schopnost využít tento registrovaný účet k přihlášení do API. Stejná možnost bude i při správě pomocí webového rozhraní.

3. Synchronizace dat mobilních zařízení

Synchronizací se rozumí proces, při kterém dojde ke sjednocení dat mezi mobilním zařízením a vzdáleným serverem. Sever i mobilní zařízení mají vlastní databázi, které jsou na sobě nezávislé. Obě strany nekomunikují přímo, ala prostřednictvím API – veřejného rozhraní, jehož úkolem je poskytnout jednotný přístup k datům uloženým na serveru. Data v mobilním zařízení lze upravovat i tehdy, kdy není dostupné internetové spojení.



Obrázek 1: Synchronizace, zdroj: vlastní

Jednou ze základních otázek je „kdy se má synchronizovat“. Na tuto otázku neexistuje jednoznačná odpověď. Záleží na mnoha aspektech, na typu aplikace, druhu dat, která se mají synchronizovat. Někdy je nutné, aby se data synchronizovala hned, jak jen to bude možné. Jindy stačí spustit synchronizaci pouze jednou denně, na příklad v noci, kdy většina uživatelů má přístroj připojený na nabíječce a nedochází tak ke zbytečné spotřebě baterie. V některých případech synchronizaci spouští uživatel, když to uzná za vhodné.

Další možností je synchronizovat data pouze pokud dojde k jejich aktualizaci. Proces pak může spouštět komponenta, která ukládá data do mobilní databáze. Synchronizaci může spouštět i server, když dojde ke změně dat.

Z výše uvedeného vyplívá, že je třeba řešit nemálo problémů, které tento proces přináší. Například při změně dat, odešle server notifikaci zařízením, o nové verzi dat. Pokud by začala stahovat data všechna zařízení v jeden okamžik, došlo by nevyhnutelně k přetížení serveru. Server by nestíhal odpovídat na požadavky všech zařízení najednou. Dalším problémem je možnost vzniku konfliktů. V některých případech může před spuštěním synchronizace dojít k úpravě stejného záznamu na serveru a současně i v zařízení. Je nutné, aby existoval mechanismus schopný přijatelným způsobem konflikty vyřešit tak, aby nedocházelo ke ztrátě dat.

Důležité je i to, zda k synchronizaci dochází na pozadí, bez vědomí uživatele, nebo zda při probíhající synchronizaci nelze aplikaci používat. Například pokud uživatel upravuje data, mohlo by mezitím dojít na pozadí k aktualizaci právě toho záznamu, který je právě upravován. Při uložení dojde nevyhnutelně ke konfliktu.

Synchronizaci lze rozdělit na dvě části, stahování dat ze serveru (download) a odesílání dat na server (upload). Aplikace může, ale nemusí obsahovat oba tyto způsoby. Některé aplikace potřebují jen stahovat data ze serveru (například aplikace, která zobrazuje stav počasí). Žádná data odesílat zpět není třeba. Jiný případ je, pokud se pouze odesílají data na server (například z důvodu zálohování). V těchto případech mnoho konfliktů není nutno řešit.

Asi nejběžnější případ je ten, kdy se data nejdříve stáhnou ze serveru, provede se jejich uložení v zařízení a poté jsou aktuální data odeslána na server. Je potřeba použít různé způsoby řešení konfliktů tak aby byly splněny všechny požadavky kladené uživateli na konkrétní aplikaci.

Důležité také je, jaké množství dat bude stahováno. Proces synchronizace by měl stahovat nebo odesílat pouze ta data, která jsou nezbytně nutná. Není možné, aby se pokaždé stahovala celá databáze na serveru. Aplikace by si měla na serveru vyžádat pouze ta data, která se změnila od poslední synchronizace.

3.1. Dostupná řešení

Nejzásadnější částí aplikace je schopnost ukládat data i v offline módu a synchronizovat je se serverem v okamžiku, kdy je internetové spojení navázáno. Bude se tedy jednat o formu cloud computingu. Primární data jsou uložena na serveru. Aplikace k nim přistupují prostřednictvím internetového spojení.

Na trhu existují hotová řešení, která lze využít při řešení tohoto požadavku. Jedná se o služby založené na mBaaS.

3.2. mBaaS

mBaaS je zkratka pro „Mobile Backend as a Service“. Skládá se tedy v podstatě ze dvou částí, backend a service. Existuje mnoho definicí. Příkladem může být ta na webu firmy Red Hat poskytující jedno z mnoha možných řešení [1]:

„The Mobile Backend-as-a-Service (MBaaS) securely integrates mobile apps with core enterprise systems, applications, and other services using RESTful APIs and a microservices-based architecture. This high-performance, cloud-based MBaaS manages data storage, scaling, push notifications, analytics, user management, and more, and is key to accelerating and simplifying back-end mobile app development.“

Tato řešení nabízejí vývojářům nástroje pro přístup ke cloudovému úložišti. Není tak potřeba vyvíjet vlastní API pro přístup k online úložišti dat. Vývojář se může soustředit hlavně na vývoj aplikace pro mobilní zařízení. Hlavní výhody použití hotového systému jsou tyto:

Serverové úložiště dat – přístup do správy datového úložiště bývá pomocí webového rozhraní. Vývojář tak může spravovat data, nebo i vytvářet skripty přímo ve webovém prohlížeči.

REST API – veřejné rozhraní pro přístup k serverové části. Aplikace tak má přístup k datům na serveru. Buď přímo nebo prostřednictvím SDK.

SDK – (software development kit). Programové rozhraní pro přístup k REST API. Bývá nabízeno pro několik platforem. Vývojář tak nemusí vyvíjet kód, který komunikuje přímo s REST API, ale může použít komfortní hotovou knihovnu.

Kromě toho nabízí každé řešení další funkce, například správu uživatelů, možnost odesílání správ do aplikace apod.

Na internetu lze najít množství komerčních i open source projektů založených na mBaasS. Některé jsou komerční, jiné volně dostupné – open source. Některé projekty přímo nabízejí hotové cloudové řešení, jiné pouze zdrojové kódy. Následuje popis několika vybraných projektů:

Parse Server

Parse Server je open source projektu Parse provozovaného Facebookem. Tento projekt byl ukončen a zároveň byly uvolněny jeho zdrojové kódy. Volně dostupná verze neobsahuje veškerou funkcionalitu a je nadále vyvíjena a udržována komunitou. Výhodou tohoto řešení je jeho bezplatná dostupnost. Nabízí celou řadu SDK. Nechybí verze pro Android, iOS, JavaScript, .NET, PHP a další. Nevýhodou je nutnost provozovat a spravovat vlastní server, který využívá databázi MongoDB [2].

Firebase

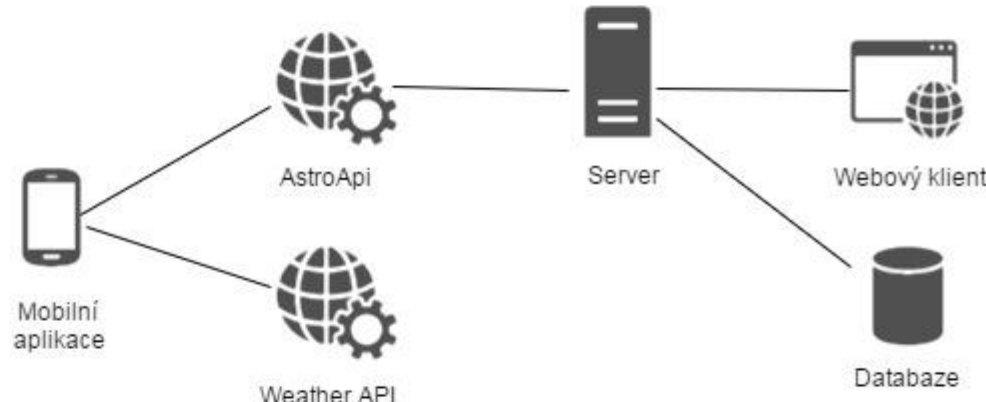
Jedná se o populární alternativu k ukončenému projektu Parse. Tentokrát z dílny Googlu. Nabízí autentizaci pomocí Facebooku, Googlu, Twitteru atd. Není potřeba provozovat žádný backend server. Podporuje offline synchronizaci, což je důležité pro tento projekt. SDK podporuje Android, iOS a C++ [3].

AppSync

Projekt, jehož cílem je nabídnout alternativu ke komerčně nabízeným komplexním systémům založeným na mBaaS. Obsahuje postupy, příklady a terminologii pro vývoj aplikací vyžadujících synchronizaci dat mobilních zařízení. Příklady jsou napsány v jazyce PHP. Implementace ostatních komponent je plně na vývojáři. Poskytuje vývojářům jednoduchý popis algoritmů, které se dají využít při offline synchronizaci dat. Neobsahuje žádné hotové REST API, zdrojové kódy serveru ani žádné SDK [4].

3.3. Vlastní řešení

Systém, který splňuje veškeré požadavky, se nemůže skládat pouze z jedné aplikace pro mobilní zařízení. Aby byly splněny všechny požadavky, musí se celý systém skládat z několika komponent, které mezi sebou komunikují prostřednictvím internetu.



Obrázek 2: Schéma systému, zdroj: vlastní

Z Obrázek 2: Schéma systému, zdroj: vlastní

je patrné, jakým způsobem budou mezi sebou jednotlivé komponenty propojeny.

Mobilní aplikace bude komunikovat se dvěma veřejnými rozhraními. Jedno bude používat ke stahování katalogu objektů a k synchronizaci deníku pozorování, z druhého bude stahovat data o aktuálním počasí.

AstroAPI je veřejné rozraní poskytující přístup k datům uloženým na serveru. Obsahuje procedury pro synchronizaci dat mezi serverem a mobilní aplikací.

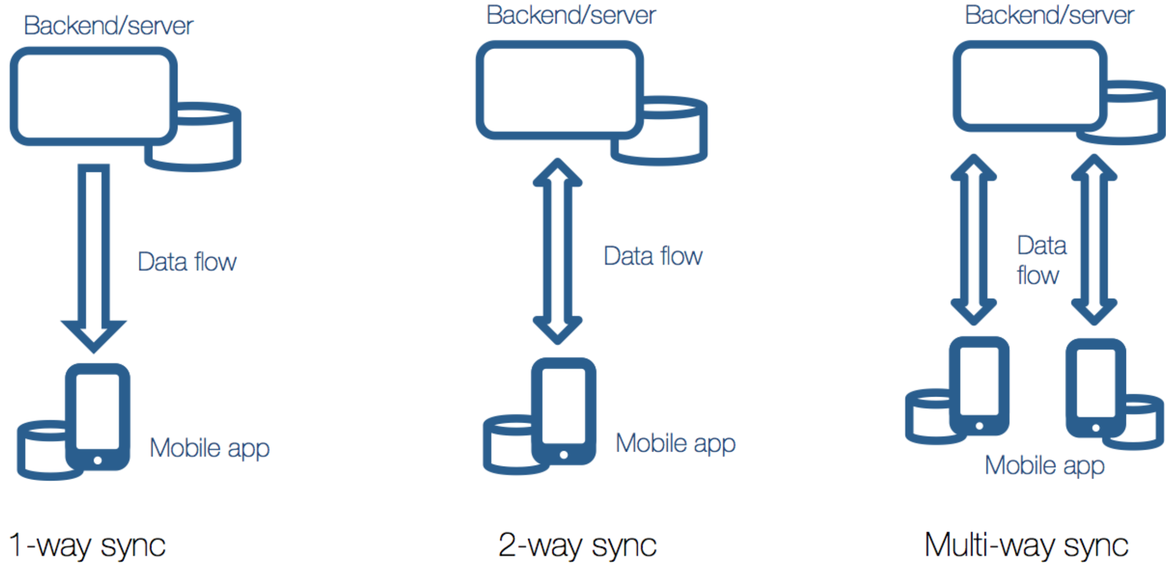
Weather API je veřejné rozhraní třetí strany poskytující údaje o aktuálním počasí na základě odeslaných zeměpisných souřadnic.

Webový klient je aplikace běžící na webovém serveru, která poskytuje přístup k veškerým datům uloženým v zařízení.

Pro potřeby této práce je potřeba vytvořit tyto komponenty:

1. AstroAPI – bude nainstalováno na serveru a založeno na jazyku PHP. Bude poskytovat přístup k datům a obsahovat autorizační systém pro zajištění bezpečnosti.
2. Webový klient – založený na jazyku PHP. Poběží na stejném serveru jako AstroAPI.
3. Mobilní aplikace – bude ukládat data v zařízení a synchronizovat je se serverem. Bude stahovat data o počasí z API třetí strany.

3.4. Typy synchronizace



Obrázek 1: Typy synchronizace, zdroj: [4]

Z obrázku je patrné že synchronizaci lze rozdělit na tři typy, podle směrů kam se data odesílají.

1-way sync – synchronizace probíhá pouze směrem od serveru do zařízení. Tento způsob používají například zpravodajské aplikace. Aplikace požádá server o aktuální data a ten je odešle. Opačný směr není potřeba.

2-way sync – synchronizace probíhá dvěma směry. Klient může data stahovat ze serveru, ale může i data odesílat na server, ale většinou je nutné přihlášení. Synchronizovat lze pouze jedno připojené zařízení.

Multi-way sync – data více než jednoho zařízení jsou synchronizována na server a zpět. Využití je u aplikací, které jsou nainstalovány na několika zařízeních (telefon, tablet…). Pokud dojde ke změně v jednom zařízení, budou následně synchronizována nejen data se serverem, ale i s ostatními připojenými zařízeními se stejným účtem.

Ukázková aplikace, která je součástí této práce, obsahuje dva moduly využívající synchronizaci. Jeden stahuje ze serveru data objektů, které se dají na obloze vyhledat. Jedná se tedy o 1-way synchronizaci. Druhý umožňuje vytvářet deník pozorování, který se dá vytvářet v zařízení i v případě nedostupného internetového spojení. Deník lze zobrazit i editovat i ve webové verzi. V tomto případě se bude jednat o Multi-way synchronizaci.

3.5. Algoritmus synchronizace

Část ukázkové aplikace, která spravuje záznamy v astronomickém deníku, musí pro splnění cílů této práce splňovat následující požadavky:

- Záznam lze vložit jak v zařízení, tak ve webovém rozhraní. Synchronizace musí probíhat oběma směry a záznamy by měly být co nejaktuálnější na obou stranách.
- Při aktualizaci dat nesmí být posílána veškerá data, ale jen data aktuální, která jsou nezbytně nutná. Synchronizace musí probíhat inkrementálně.
- Záznam, který je vytvořený v zařízení, musí být dostupný ve všech ostatních zařízeních, která jsou na účet připojena.
- Musí být umožněno editování záznamu na jakémkoliv zařízení, bez ohledu na to, kde byl vytvořen. Pokud se záznam upraví na dvou zařízeních současně, před provedením synchronizace, dojde nezbytně ke konfliktu, záznam bude existovat ve dvou verzích. Tyto konflikty musí být řešeny při synchronizaci.
- Aplikace musí být funkční i v offline módu. Záznamy budou synchronizovány při nejbližším připojení na internet.

Použitý algoritmus, který splňuje výše uvedené požadavky, vychází z projektu AppSync [4]. Pro jeho pochopení je důležité definovat některé důležité pojmy:

Server – centrální komponenta přístupná pomocí internetu, která zodpovídá za synchronizaci dat mezi jednotlivými zařízeními.

Klient – aplikace běžící na mobilním zařízení umožňující vytvářet a upravovat data astronomického deníku.

Objekt – část dat reprezentující jeden záznam v astronomickém deníku. Obsahuje veškerá data reprezentující jedno uskutečněné pozorování.

Inkrementální synchronizace – synchronizují se pouze data, která byla změněna od poslední úspěšné synchronizace.

Řešení konfliktů - nutnost rozhodnutí mezi dvěma verzemi objektu v případě, že byl objekt změněn ve dvou zařízeních současně před synchronizací se serverem.

Primární klíč – entita, která jednoznačně identifikuje záznam. Záznamy v jedné tabulce nemohou obsahovat stejné primární klíče.

Konflikt primárních klíčů – k tomuto konfliktu dochází, pokud se ve dvou nebo více klientech vytvoří záznam se stejným primárním klíčem.

GUID – unikátní identifikátor objektu vytvářený na serveru nebo v klientovi. Tato hodnota by měla být unikátní napříč všemi klienty a serverem.

Counter – koncept zajišťující informaci o tom, v jakém okamžiku byl záznam změněn. Náleží každému objektu a je zvýšen při každé změně která na objektu vznikne.

Soft delete – pokud je objekt smazán, jeho vlastnost např. „isDeleted“ se nastaví na hodnotu *true*. Nedojde tedy k úplnému odstranění záznamu z databáze.

Timestamp – časové razítko. Každý objekt ukládá do této vlastnosti informaci o přesném čase, kdy byl vytvořen nebo změněn. Lze ho využít při řešení konfliktů.

Předpoklady pro správné fungování synchronizačního algoritmu [4]:

1. Synchronizaci vždy zahajuje klient, nikdy ne server. Důvodem je předpoklad, že pouze aplikace běžící na mobilním zařízení dokáže nejlépe rozhodnout o tom, kdy je nejvhodnější čas pro zahájení synchronizace. Záleží to například na internetovém spojení nebo na provedení určité akce uživatelem.
2. Řešení konfliktů probíhá vždy v zařízení, nikdy ne na serveru. Důvodem je nutnost informovat uživatele o vzniklém konfliktu. Uživatel může být ten kdo rozhoduje, která verze objektu je ta správná. Některé strategie řešení konfliktů nevyžadují rozhodnutí uživatele, ale může být vhodné předat informaci, že k tomu došlo. Konkrétní strategie řešení konfliktů záleží na případu použití.
3. Při synchronizaci provádí mobilní aplikace vždy nejdříve stažení dat (download), po jejich uložení a vyřešení případných konfliktů, je provedeno odeslání aktuálních dat na server (upload).
4. Server není informován o stavu synchronizace jednotlivých klientů. Každý klient si musí zajistit aktualizaci svých dat sám.

3.5.1. Vytváření GUID

Každý objekt, vytvořený v klientem, obsahuje GUID. V některých případech je možné zajistit jeho unikátnost. Například v případě rezervačního systému hotelových pokojů, může být GUID vytvořeno kombinací identifikátoru uživatele, čísla pokoje a data kdy je pokoj, rezervován. Tato kombinace zajistí unikátnost záznamu a umožní spárovat dva záznamy vytvořené v různých zařízeních.

V případě astronomického deníku něco takového možné není. V tomto případě, je nutné pro každý nový záznam vygenerovat řetězec znaků který je pokládán za unikátní napříč zařízeními. Algoritmus generující GUID je odlišný v klientovi i na serveru. Je to způsobeno odlišností platforem, pod kterou aplikace běží. Ve webové aplikaci a v klientovi by mohl být pro nový objekt vygenerován stejný GUID. Dva odlišné záznamy by byly identifikovány jako jeden a při řešení

konfliktů by nevyhnutelně došlo ke ztrátě dat. Ačkoliv je vznik této situace velmi nepravděpodobný, musí existovat způsob, jak tomuto zabránit.

Při odesílání dat na server, je každý záznam, který je od poslední synchronizace vytvořen v klientovi označen příznakem (například „isNew=true“). GUID je mu přiděleno pouze dočasně a není uloženo v databázi. K jeho perzistenci (uložení) dojde až v případě úspěšného dokončení synchronizace. Při ukládání dat na serveru musí být tyto záznamy vloženy do databáze metodou „INSERT“. Pokud by na serveru už existoval objekt se stejným GUID, dojde k chybě a upload bude označen jako neúspěšný. Klient tak při další synchronizaci vygeneruje novým záznamům nová GUID.

3.5.2. Counter

Systém používá dva countery, *server_counter* a *row_counter*.

Server_counter je hodnota uložená na serveru, reprezentující určitý stav systému vzhledem k provedeným změnám.

Row_counter je hodnota uložená přímo u každého objektu. Určuje dobu vzniku, nebo aktualizace vzhledem k *server_counteru*. Při vytváření nového objektu se k objektu do vlastnosti *row_counter* uloží aktuální hodnota *server_counteru*. Pokud se nový objekt vytvoří v klientovi, je jako hodnota row_counteru použita hodnota *serever_counteru* který klient obdržel při poslední úspěšné synchronizaci.

Hodnota *server_counteru* je vždy zvýšena o 1 pokud dojde k úpravě, vložení nebo smazání dat přímo na serveru. Touto hodnotou je reprezentován přesný stav systému a umožňuje jednotlivým klientům identifikovat, zda došlo od poslední synchronizace dat k nějaké změně. *Server_counter* je odeslán klientovi s každými staženými daty. Klient ho uloží ve své databázi a použije ho při další synchronizaci. Server tak odešle klientovy pouze ta data, která mají hodnotu *row_counteru* větší, než hodnota i při posední synchronizaci. Klient používá *server_counter* také pro aktualizaci *row_counterů* objektů upravovaných v klientovi. To je důležité při odesílání na server. Pokud je například hodnota *row_counteru* objektu na serveru větší než u objektu, který je odeslán klientem, znamená to, že objekt byl od poslední synchronizace dat s klientem změněn. Synchronizace tak skončí chybou. Klient v tomto případě musí stáhnout novou verzi dat, provézt aktualizaci a vyřešení konfliktů. Upravená data opět odešle na server.

3.5.3. Řešení konfliktů

Jak bylo popsáno výše, server nikdy konflikty neřeší. To neznamená, že žádné konflikty nemohou vzniknout. Pokud klient pošle na server data obsahující objekty, které byly od poslední synchronizace změněny, dojde nutně ke konfliktu. Server tyto konflikty neřeší. Data, která jsou konfliktní odmítne uložit. Například v případě použití REST API odešle server odpověď s HTTP statusem 409 Conflict. Klient tak musí data stáhnout znovu a vyřešit konflikty sám.

Pokud v klientu dojde k úpravě objektu, je v lokální databázi označen příznakem, například “sync_ok=false”. Při synchronizaci může dojít k situaci kdy mezi objekty staženými ze serveru je i objekt upravený v klientovi a dosud nesesynchronizovaný. Tuto situaci musí aplikace v klientovi vyhodnotit jako konflikt a příslušným způsobem jí vyřešit.

Při řešení konfliktů se dají uplatnit tři základní postupy.

- Priorita serveru - v tomto případě se použije verze objektu ze serveru. Dojde ke ztrátě dat vytvořených uživatelem. Uživatel by měl být o této události informován
- Priorita klienta - v tomto případě se použije verze kterou vytvořil uživatel. Verze objektu ze serveru nebude použita. V další fázi, při odesílání dat na server bude tento objekt odeslán na server, kde bude provedena jeho aktualizace podle příchozí verze. Tím dojde ke ztrátě dat na serveru.
- Priorita timestampu - v tomto případě bude zachována ta verze objektu, jejíž timestamp bude vyšší - byl vytvořen později. Tento způsob se zdá jako nejvhodnější a z hlediska uživatele za nejsnáze pochopitelný. Pro správnou funkci je ale nezbytné zajistit, aby všechna zařízení, včetně serveru, měla synchronizovaný čas.
- Jako další možný způsob řešení konfliktů se jeví možnost ponechat rozhodnutí na uživateli. Ten musí být na konflikt upozorněn a musí mu být dána možnost rozhodnout, která verze bude použita. Existuje mnoho dalších různých strategií řešení konfliktů, které záleží na konkrétních případech.

3.5.4. Testování

Po vytvoření aplikace je třeba vytvořit sadu testů, které ověří správného fungování algoritmu. Dále je třeba ověřit, zda se data někde neztrácí a zda skutečně dojde k úplné synchronizaci na všech připojených zařízeních. Je třeba otestovat zejména tyto případy:

1. Synchronizace ze serveru do klienta: vytvoření a update objektu
2. Synchronizace z klienta na server: vytvoření a update objektu
3. Synchronizace z klienta A do klienta B
4. Zabránění nechtěné synchronizace: klient, který odešle data na server by je už neměl dostat zpět při updatu.
5. Synchronizace smazaných objektů (soft delete)
6. Synchronizace s řešením konfliktů: objekt je upraven současně na serveru i v klientovi
7. Synchronizace s řešením konfliktu na primárním klíči
8. Úplná synchronizace: vytváření a úprava objektů v klientovi v offline módu a jejich plná synchronizace se serverem a ostatními zařízeními.

Po úspěšném provedení testů všech výše uvedených případů je možno synchronizaci pokládat za úspěšnou.

3.6. API

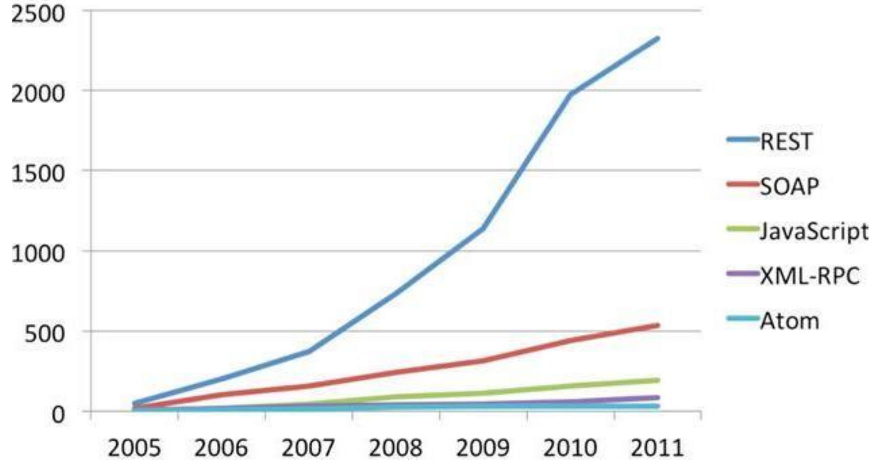
API je zkratka pro Application Programming Interface. Jedná se o software, který usnadňuje komunikaci s jiným softwarem. API umožní vývojáři komunikovat s aplikací pomocí sady

předpřipravených funkcí a procedur. Výhodou API je skutečnost že program, který API používá, bude fungovat i poté, co byl software, jemuž API náleží změněn, nebo aktualizován. Součástí API bývá SDK (Software Development Kit). Je to sada procedur umožňujících komunikovat s API v určitém programovacím jazyku. Typicky provozovatel webové služby, která je přístupná pomocí API, nabízí SDK pro několik různých platforem. Například Firebase nabízí SDK pro Android, iOS a C++. Programátor se tak může rozhodnout, která platforma je po něj nejvýhodnější.

V souvislosti se synchronizací se vzdáleným serverem pomocí internetu, je nutno uvažovat o API, které bude přístupné pomocí protokolu HTTP. Nejpoužívanějšími protokoly komunikujícími přes HTTP jsou SOAP a REST.

3.6.1. Soap

SOAP (Simple Object Access Protocol) je protokol vytvořený firmou Microsoft v roce 1998. Je určený pro komunikaci mezi serverem a klientem pomocí protokolu HTTP nebo SMTP (SMTP se převážně nepoužívá). Pro komunikaci s API používá XML, který definuje formu obsahu zpráv, to je asi největší rozdíl oproti REST, kde je obsah definován přímo v HTTP [5]. SOAP je oproti REST složitý na používání a vyžaduje podporu programových knihoven. To je nejspíš důvodem jeho nízké popularity oproti REST, viz. Obrázek 2: Popularita API řešení, zdroj: [6].



Obrázek 2: Popularita API řešení, zdroj: [6]

3.6.2. REST

„REST is not a protocol, a file format, or a development framework. It’s a set of design constraints: statelessness, hypermedia as the engine of application state, and so on. Collectively, we call these the Fielding constraints, because they were first identified in Roy T. Fielding’s 2000 dissertation on software architecture, which gathered them together under the name “REST.”“ [7, str. 29].

REST umožňuje odesílat data přímo pomocí HTML protokolu. Není tak potřeba vytvářet složité protokoly popisující způsob komunikace. To umožní vývojářům efektivně vytvářet aplikace, které přistupují k datům na vzdáleném serveru i v prostředích, která nedisponují vysokým výkonem nebo rychlým internetovým spojením.

REST funguje téměř stejně jako webový prohlížeč. Zdroj (resource) je reprezentován pomocí URL. Klient tak pouze přistoupí na určitou URL a obdrží požadovaná data. Popularita REST je způsobena jeho jednoduchostí, srozumitelností a možností snadno otestovat jeho funkčnost.

Bezstavovost

Důležitou vlastností REST API je jeho bezstavovost. Pokud server obdrží požadavek od klienta, nemá žádnou spojitost s jakýmkoliv předchozím požadavkem. Klient musí veškeré informace odeslat v každém požadavku. To přináší problémy při autentizaci. Každý požadavek musí nést informaci identifikující klienta.

HTTP metody

Ke zdrojům lze přistupovat pomocí tzv. HTTP metod. Každá metoda reprezentuje záměr, co se s daným zdrojem má provést. Nejčastějšími metodami používanými REST jsou tyto [7, str. 34]:

GET – metoda slouží jako reprezentace zdroje. Zdroj používající GET by měl pouze obdržet data.

DELETE – provede smazání jedné nebo více položek zdroje.

POST – je metoda používaná k odeslání dat na určitý zdroj. Je použita pro vytváření nových položek, nebo změnu stavu zdroje.

PUT – metoda provádí aktualizaci jedné nebo více položek zdroje odeslanými daty.

Data, která vrací REST API, nebo která jsou na něj odesílána klientem, jsou ve formátu JSON, nebo XML. V této práci a v ukázkové aplikaci se používá výhradně formát JSON.

HTTP status

Při zpracování odpovědi REST API, obdrží klient kromě payloadu i HTTP status. HTTP status je třímístné číslo připojené ke každé odpovědi serveru. Klient se tak ihned pozná, jakým způsobem byl jeho požadavek na serveru zpracován. Popisuje obsah dat vrácených serverem. Pokud například dojde k chybě, vrátí server HTTP status 400. Klient se tímto způsobem dozví, že data vrácená serverem obsahují popis chyb a může na to adekvátním způsobem reagovat. Po přečtení prvních pár bytů odpovědí, může klient rozpoznat, zda byl jeho požadavek úspěšný či ne.

První číslice HTTP statusu vždy reprezentuje obecnou skupinu stavů. HTTP specifikace definuje pět základních stavů [7, str. 296]:

2xx: Successful – reprezentuje stav, kdy byl klientův požadavek splněn.

3xx: Redirection – klientův požadavek nebyl splněn, ale pokud upraví svůj požadavek, dostane požadované.

4xx: Client Error – klientův požadavek nebyl splněn. Chyba je na straně klienta. Požadavek byl špatně naformátován nebo server nemůže vyhovět požadavkům.

5xx: Server Error – klientův požadavek nebyl splněn. Chyba na straně serveru. Problém není způsobem špatným dotazem, nebo nesplnitelností požadavku. Klient nemá vliv na odstranění chyby.

Nejdůležitějšími kódy, které s v REST API používají jsou tyto:

200 (OK) – všechno je v pořádku, server provedl to, co klient žádal.

301 (Moved Permanently) – server provedl přesměrování z jedné URL na jinou. Zdroj byl přesunut na jinou URL a na ní se bude nadále nalézat. Klient by měl příští požadavek odeslat přímo na URL kam byl přesměrován.

400 (Bad Request) – problém na straně klienta. Tělo dokumentu vráceného serverem neobsahuje očekávaná data, ale popis chyby, ke které došlo. V ideálním stavu, může klient porozumět chybě a příslušným způsobem na ní reagovat.

401 (Unauthorized) – Problém při autorizaci požadavku. Nastává, pokud klient požaduje data bez prokázání identity.

404 (Not Found) - Server nenašel požadovaný dokument.

409 (Conflict) – Obvykle znamená problém při ukládání dat. Například klient odeslal na server objekt, při jehož ukládání došlo ke konfliktu, který server není schopný vyřešit.

500 (Internal Server Error) – Problém nastal na straně serveru. Tělo dokumentu vráceného serverem obsahuje chybovou zprávu. Klient pravděpodobně nemůže ovlivnit úspěšné vyřešení vzniklé situace.

3.7. OAuth

Pokud dochází k synchronizaci dat mezi klientem a serverem, měl by mít klient přístup pouze k datům, která náleží účtu, pod kterým je zaregistrovaný a přihlášený. Kvůli bezstavovosti protokolu REST je nutné, aby klient s každým odeslaným požadavkem prokázal svoji identitu. Z hlediska bezpečnosti, není možné, aby klient sděloval pokaždé své přihlašovací jméno a heslo. Tento problém řeší protokol OAuth.

„The OAuth protocol enables websites or applications (Consumers) to access Protected Resources from a web service (Service Provider) via an API, without requiring Users to disclose their Service Provider credentials to the Consumers. More generally, OAuth creates a freely-implementable and generic methodology for API authentication." [8].

OAuth pochází z roku 2006. V současné době se používá verze 2.0, která je méně komplikovaná, více bezpečná a nabízí podporu i klientů bez webového prohlížeče. V této práci budou veškeré zmínky o OAuth mířeny na verzi OAuth 2.0.

3.7.1. Role

OAuth definuje 4 role:

- **Resource Owner:** jedná se o uživatele, který vlastní data, ke kterým přistupuje.

- **Resource Server:** server na kterém jsou uživatelova data uložena
- **Client:** aplikace, která potřebuje přístup k Resource Serveru (může se jednat o webovou stránku, JavaScript aplikaci, nebo mobilní aplikaci)
- **Authorization Server:** Autorizační server poskytující acces_token klientovi. Acces_token používá klient k odesílání požadavků na API Resource serveru. Resource server může být shodný s Autorizačním serverem.

3.7.2. Tokeny

Tokeny jsou náhodně generované řetězce znaků, generované Autorizačním serverem. Klient si je musí vyžádat.

Existují 2 typy tokenů:

- **access_token:** nejdůležitější token zpřístupňující uživatelova data aplikacím třetích stran. Je odeslán klientem v parametru URL, nebo v hlavičce s každým požadavkem na Resource server. Jeho doba použitelnosti bývá omezená, určuje jí Autorizační server.
- **refresh_token:** tento token bývá poskytován spolu s *access_tokenem* v některých způsobech autentizace. Na rozdíl od *access_tokenu* se neodesílá na Resource server s každým požadavkem. Používá se výhradně k získání nového *access_tokenu* poté, co byla ukončena jeho platnost.

3.7.3. Způsoby autorizace

OAuth definuje 4 druhy autorizace (Authorization Grant Types) v závislosti na druhu a umístění klietnta

1. **Authorization Code Grant** – používá se, když klientem je webová aplikace. Dovoluje získat *access_token* i *refresh_token*.
2. **Implicit Grant** – používá se v případech kdy je klientem script napsaný v JavaScriptu. Tento typ autorizace nedovoluje užití *refresh_tokenu*.
3. **Resource Owner Password Credentials Grant** – v tomto případě je *access_token* získán pomocí klienta, kterému musí uživatel svěřit své přihlašovací údaje. Z toho vyplívá, že musí jít o důvěryhodného klienta. Používá se zejména v případech, kdy je klient vytvořen stejnou autoritou jako Autorizační server.
4. **Client Credentials Grant** – používá se v případě, kdy je klient v roli Resource Ownera. Není potřeba autorizace od uživatele. Např. webová aplikace používající přístup ke cloudovému úložišti.

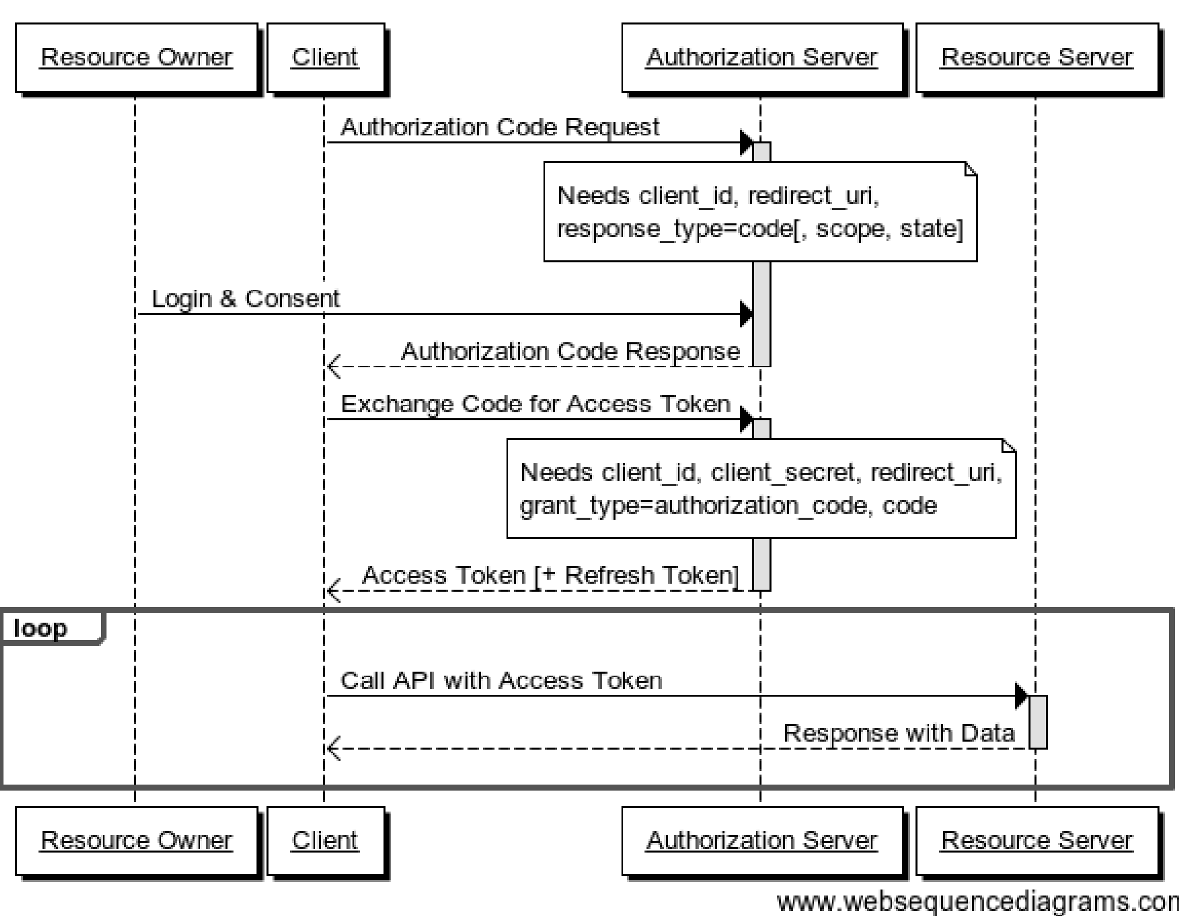
Aplikace, které jsou součástí této práce používají Authorization Code Grant a Resource Owner Password Credentials Grant, proto budou vysvětleny podrobněji.

Authorization Code Grant

Typický scénář užití tohoto způsobu autorizace pro přístup k účtu Googlu:

1. Webová stránka chce zjistit informace z uživatelova Google účtu.

2. Je provedeno přesměrování prohlížeče na webovou stránku Autorizačního serveru (Google).
3. Pokud uživatel potvrdí přístup, Autorizační server odešle autorizační kód klientovi (webové stránce)
4. Klient prostřednictvím autorizačního kódu získá z Autorizačního serveru *access_token*.
5. Webová stránka má nyní přístup k informacím o uživateli z jeho Google účtu.

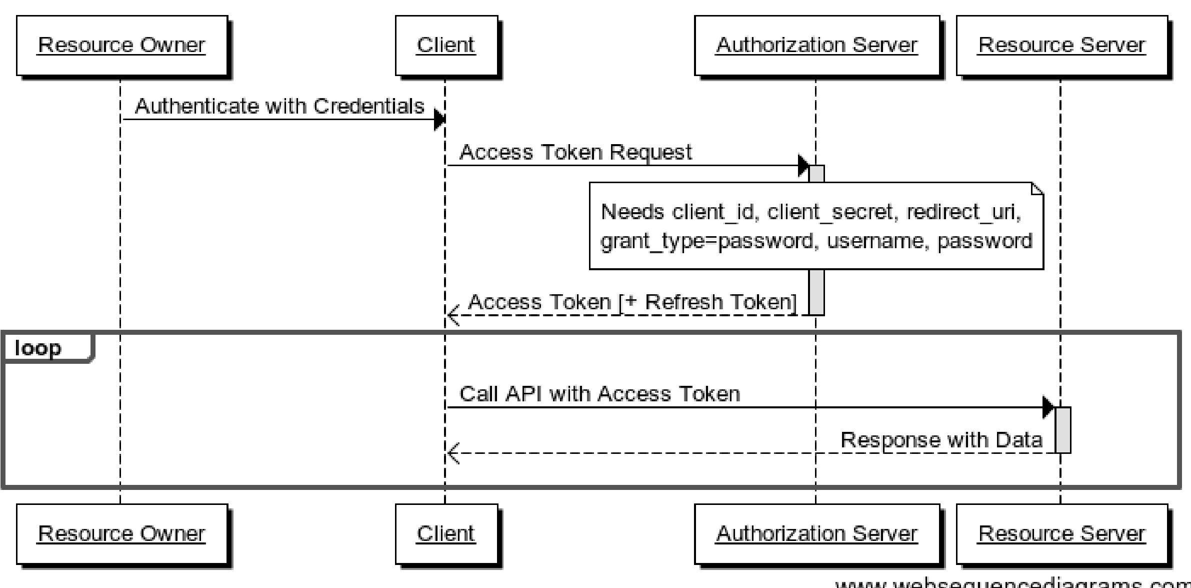


Obrázek 3: Authorization Code Grant Flow, zdroj: [9]

Resource Owner Password Credentials Grant

Tento způsob autorizace je vhodné použít v případě, kdy je třeba synchronizovat aplikace pro mobilní zařízení přihlášené pod jedním účtem. Typický scénář:

1. Mobilní aplikace (klient) požádá uživatele o přístupové údaje k jeho účtu.
2. Uživatel zadá login a heslo, ty jsou odeslány na Autorizační server.
3. Autorizační server ověří přihlašovací údaje a vrátí klientovi *access_token* a *refresh_token*.
4. Klient uloží tokeny a pro přístup k Resource serveru používá *access_token*.



Obrázek 4: Resource Owner Password Credentials Grant, zdroj: [9]

4. Implementace na straně serveru

4.1. Server

Aplikace v mobilním zařízení musí stahovat a ukládat data ze vzdálené databáze. Pro tento účel, potřebuje službu běžící na pozadí (backend service), která zprostředkuje požadavky mezi SQL serverem a aplikací.

Server, běžící na adrese http://astro.8u.cz, je napsán v jazyku PHP. Aplikace v mobilním zařízení je napsána v jazyku JAVA pro prostředí Android. Obě strany tedy používají odlišné technologie. Pro snadnější komunikaci je zřízeno aplikační rozhraní (API) běžící na vzdáleném serveru. API zajišťuje jednotný přístup k funkcím a procedurám potřebným pro požadovanou funkcionalitu aplikace. Architektura REST umožňuje posílat požadavky mezi online aplikacemi pomocí protokolu HTTP. Data jsou posílána ve formátu JSON, který je nezávislý na programovacím jazyku.

4.1.1. REST

Server je napsán v jazyku PHP, pro vývoj byl použit framework Nette [23] s rozšířením Drahak/Restful [10]. Toto rozšíření usnadňuje programování HTTP požadavků a odpovědí.

Framework umožňuje používání modelu MVC. Ve frameworku Nette je controller nazýván presenter. Veškeré požadavky na serveru jsou zpracovávány jedním z presenterů. Který presenter bude požadavek zpracovávat se rozhoduje pomocí třídy *RouterFactory*, kde jsou definovány základní pravidla pro přerozdělování požadavků, tzv. routy.

Rozšíření Drahak/Restfull navíc umožňuje u každé routy definovat, jaký typ požadavku daná routa bude zpracovávat.

```
$router[] = new ResourceRoute("diary[/<id>]", [
  'presenter' => 'Diary',
  'action' => array(
    ResourceRoute::GET => 'list',
    ResourceRoute::POST => 'update'
  )
], ResourceRoute::POST | ResourceRoute::GET);
```

Z kódu je patrné, že presenter *DiaryPresenter* bude zpracovávat pouze požadavky typu POST a GET, které přišly na URL */diary* s nepovinným parametrem *id*. Metoda *actionList()* obslouží požadavky typu GET a metoda *actionUpdate()* obslouží požadavky typu POST.

Pokud je v tomto případě použit nesprávný typ požadavku, např. místo POST je použit PUT, reaguje server odesláním odpovědi s http statusem 405 Method Not Allowed. Zpracování požadavků je tak přehledné a snadno použitelné.

Zpracování požadavků

V presenteru je třeba správně zpracovat tělo HTTP požadavku který byl odeslán. Díky rozšíření Drahak/Restful je toto zpracování velice snadné. Presentery zpracovávající požadavky jsou potomky třídy Drahak\Restful\Application\UI\ResourcePresenter. V metodě zpracovávající požadavek je k dispozici protected proměnná *$input*, která obsahuje data obdržená v HTTP požadavku. Např. na metodu je routerem nasměrován požadavek obsahující v těle HTTP zprávy kód:

```
{
    "login": "JohnDoe",
    "password" : "john1234",
}
```

Hodnota proměnné login a password se získá velice snadno:

```
$login=$input->login;
$password=$input->password;
```

V presenteru je možné data před zpracováním validovat a odpovídajícím způsobem na ně reagovat:

```
if (!isset($this->input->objects)) {
  throw BadRequestException::unprocessableEntity([], 'Wrong data format');
}
```

Pokud se v těle HTTP zprávy nenajde pole objects, je vrácena odpověď obsahující status 422 - Unprocessable Entity.

Odpovědi serveru

Vytváření odpovědí serveru je podobně snadné jako zpracování požadavků.

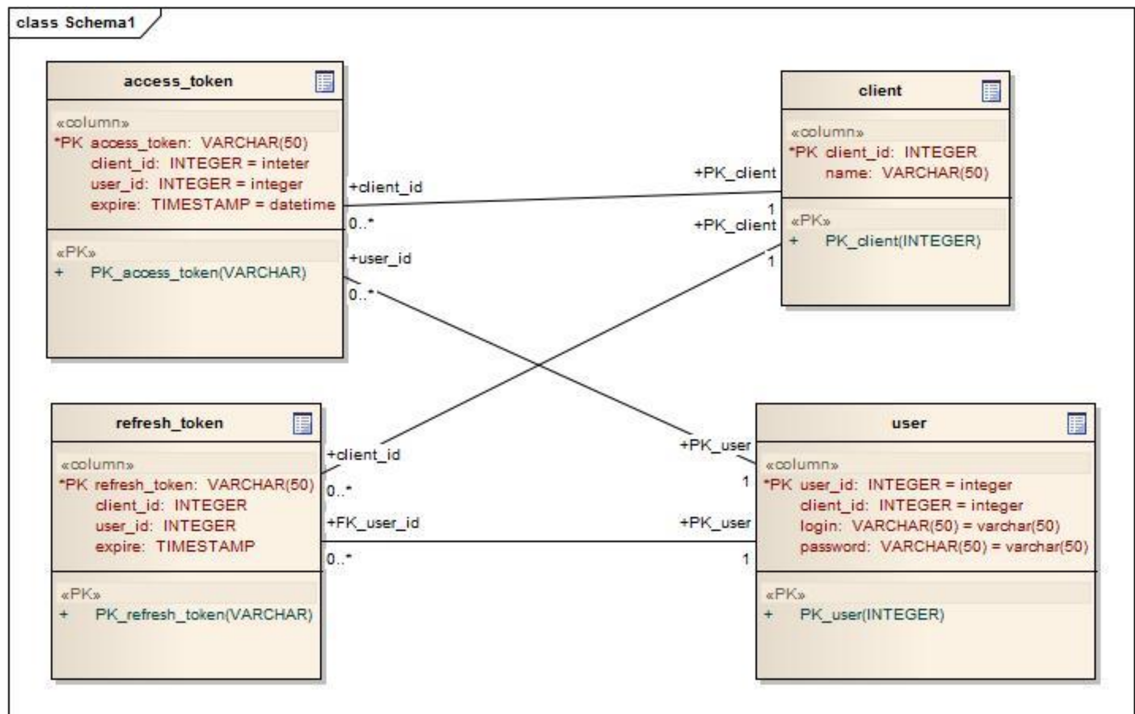
```
$this->resource = ['version' => 1];
```

Uvedený kód vloží do odpovědi serveru proměnnou version s hodnotou 1.

```
{
  "version": 1
}
```

4.1.2. OAuth

Implementace OAute je vyvinuta pro potřeby této práce. Resouce server (server který obsahuje databázi s klientskými daty) slouží zároveň jako Autorizační server. Odpadá tedy nutnost komunikace mezi těmito dvěma servery. Struktura databáze je zřejmá z obrázku.



Obrázek 5:Struktura tříd, zdroj: vlastní

Autorizační tokeny jsou řetězce znaků, které jsou dostatečně unikátní, aby jimi bylo možno identifikovat klienta. Tento řetězec se odešle klientovi a zároveň se uloží do databáze serveru a použije se k identifikaci klienta.

K vytvoření tokenu je použita třída *OpenSSLGenerator*. Objekt této třídy obdrží v konstruktoru název algoritmu, který bude použit pro generování tokenu. V tomto případě je to algoritmus sh256. Metoda *getKey()* vrací token o šířce 40 znaků.

```
public function getKey()
   {
       $bytes = openssl_random_pseudo_bytes(40);
       return hash($this->algorithm, $bytes);
   }
```

Informace o trvanlivosti tokenu je uložena do databáze při jeho vytvoření. Server tedy v budoucnu nemusí rozhodovat o tom, jak dlouho má mít token trvanlivost. Jednoduše se dotáže na tokeny jejichž čas expirace je větší než aktuální čas.

Doba trvání jednotlivých tokenů je uložena v sekundách v konfiguračním souboru config.neon.

```
duration:
      accessToken: 3600
      refreshToken: 2592000
```

Z kódu vyplívá, že *access_token* má trvanlivost jednu hodinu a *refresh_token* jeden měsíc.

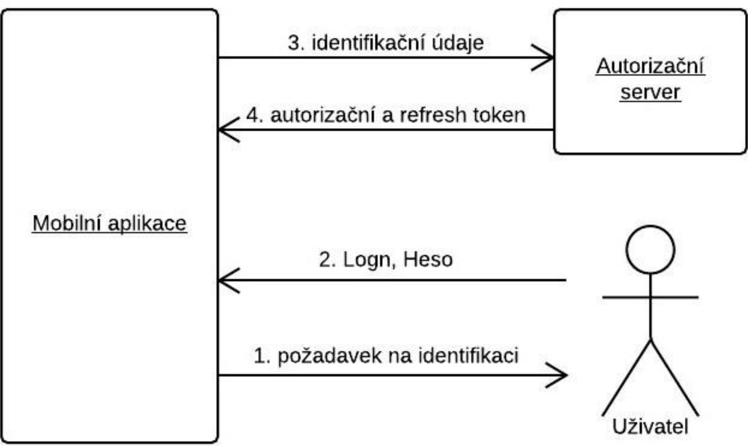
4.2. API

4.2.1. Autentizace klienta

Nativní aplikace je veřejný klient nainstalovaný na mobilní zařízení jeho uživatelem. Přihlašovací údaje (login, heslo) jsou do aplikace zadány uživatelem a mohou být aplikací odeslány na server. Při použití protokolu OAuth se dynamicky mění identifikační údaje – *access_token* a *refresh_token*. Tím je zajištěna dostatečná úroveň bezpečnosti. Identifikační údaje uživatele jsou chráněny před potencionální hrozbou nepřátelských útoků. [11, sec 2.1.]

Přístup na server je dovolen všem klientům. Není tedy povolen pouze aplikacím nainstalovaným v mobilním zařízení. Proto, pro obdržení autorizačního tokenu není třeba s *client_id* aplikace posílat žádné další tajné heslo. Stačí pouze přihlašovací údaje uživatele. API je tedy možno používat různými klienty. Například klient na webovém serveru nebo mobilní aplikace. Tato práce popisuje využití API mobilní aplikací.

Autorizační proces je popsán na schématu:



Obrázek 6: Autorizační schéma, zdroj: vlastní

1. Pokud chce aplikace přistupovat k datům na vzdáleném serveru, musí mít autorizační token. K jeho získání potřebuje uživatelův login a heslo. Uživatel je tedy zobrazením příslušného formuláře vyzván k zadání svých identifikačních údajů.
2. Uživatel zadá své přihlašovací údaje do formuláře a odešle je do aplikace.
3. Aplikace si uloží login a heslo. Odešle je na autorizační server spolu s *client_id*. Požadavek obsahuje *grant_type: password*.
4. Při správném loginu a heslu, vrátí autorizační server nově vytvořený *access_token* a *refresh_token*. Oba tyto tokeny jsou aplikací uloženy pro další použití.

4.2.2. Authorizační token

V 3.7 byly popsány důvody používání autorizačního tokenu. Tato aplikace používá i *refresh_token* k obnovení *access_tokenu* jehož platnost už vypršela.

Když chce aplikace v mobilním zařízení komunikovat se serverem, je třeba nejdříve získat aktuální *access_token*. Sever musí uživatele identifikovat. Proto potřebuje autentizační údaje od klienta, v tomto případě přihlašovací jméno a heslo. Dále potřebuje ID klienta, se kterým komunikuje. Důležitou součástí je *grant_type:password* jehož prostřednictvím oznamujeme serveru jaký typ dat mu posíláme.

```
POST /token
{
	"login": "uzivatel@domena.cz",
	"password" : "heslo",
	"client_id" : 1,
	"grant_type" : "password"
}
```

Pokud přihlašovací údaje souhlasí, obdrží klient odpověď od serveru s HTTP statusem 201 Created:

```
{
	"access_token":
	"e8302c71e745525b5134cd702bd9bf4444950e762470a8b33ca13e6a96a65e60",
	"token_type": "bearer",
	"expires_in": 3600,
```

```
	"refresh_token":
	"2320b0f8a08252a8740b77b93a0a1991c85cb10a94ea4357479007d0f1411b41"
}
```

4.2.3. Odmítnutí požadavku serverem

Při nesprávných přihlašovacích údajích bude odpověď serveru obsahovat HTTP status 401 Unauthorized.

```
{
	"code": 401,
	"status": "error",
	"message": "unauthorized"
}
```

4.2.4. Refresh token

Obdržený autorizační token má trvanlivost jednu hodinu. Po jeho expiraci bude server odpovídat na veškeré požadavky HTTP statusem 401 Unauthorized, stejně jako při neplatných přihlašovacích údajích v 4.2.3.

K získání nového *access_tokenu*, nyní stačí *refresh_token*, který klient obdržel po odeslání svých přihlašovacích údajů. URL je stejná jako při získávání tokenu pomocí loginu a hesla. Tělo požadavku obsahuje *grant_type:refresh_token*.

```
POST /token
{
	"refresh_token":
	"07a49c21ef93c767cb619e6eb608705385c807109269ac7a346c44f8f83a64f2",
	"client_id" : 1,
	"grant_type" : "refresh_token"
}
```

Odpověď serveru je v tomto případě stejná jako při odeslání přihlašovacích údajů.

Trvanlivost *refresh_tokenu* je jeden měsíc. Po jeho expiraci je třeba zaslat serveru opět přihlašovací jméno a heslo jako při prvním získání autorizačního tokenu.

4.3. Komunikace se serverem

Při příchodu požadavku na Resource server, musí být jednoznačně identifikován klient, od kterého požadavek vzešel. Právě k tomuto účelu slouží autorizační token. Ten zastupuje autentizační údaje klienta a nevzniká tak potřeba s každým požadavkem na server posílat zároveň uživatelovo přihlašovací jméno a heslo. Klientovy přihlašovací údaje je potřeba zaslat pouze při prvním získání autorizačního tokenu a poté při expiraci *refresh_tokenu*. Interval mezi zasíláním přihlašovacích údajů na server je tedy shodný s dobou existence *refresh_tokenu*. V tomto případě je to jeden měsíc.

Do URL každého požadavku na Resource server je tedy nutné vložit query [12, sec 3.] obsahující *access_token*.

K získání dat pro naplnění aktuálního katalogu astronomických objektů je použita následující url s query parametrem *access_token*:

```
GET /messierData
?access_token=70af5486b5021562eaba29c852ab6696d3bf47fcabd91a7d25bf0932689e42f0
```

Pokud není autorizační token starší jedné hodiny, vrátí klientovi požadovaná data spolu s HTTP statusem 200 OK.

V opačném případě bude odpověď obsahovat HTTP status 401 Unauthorized jako v 4.2.3.

4.4. Získání aktuálních dat pro katalog astronomických objektů

Po instalaci aplikace neobsahuje katalog pozorovatelných objektů žádné záznamy. V tomto případě klientská aplikace požaduje od serveru zaslání aktuálního seznamu astronomických objektů. Databáze objektů na serveru se může měnit, mohou být přidána nová data nebo se stávající data mohou upravovat. Server z tohoto důvodu obsahuje číslo aktuální verze katalogu. Aplikace nejdříve zjistí číslo verze dat na serveru a porovná ji s aktuální verzí nainstalovaných dat v aplikaci. Pokud je verze dat na serveru větší než ta aplikační, je nutno data aktualizovat.

Zjištění verze katalogu astronomických objektů:

```
GET /messierData/version
{
	"version": 5
}
```

Je patrné, že verze dat na serveru má hodnotou 5.

URL s požadavkem musí obsahovat i query s parametrem *refresh_token*. Důvody jeho použití byly vysvětleny výše a v příkladech nebude dále uváděno.

V případě, že aplikace zjistí, že je potřeba data aktualizovat použije k tomu následující požadavek:

```
GET /messierData
{
  "list": [
    {
      "messier_id": 1,
      "ngc": 1952,
      "constellation": "Tau",
      "type": 3,
      "ra_deg": 5,
      "ra_min": 31.5,
      "dec_deg": "21",
      "dec_min": 59,
      "magnitude": 8.2,
      "distance": 6
    },
    {
      "messier_id": 2,
      "ngc": 7089,
      "constellation": "Aqr",
      "type": 2,
      "ra_deg": 21,
      "ra_min": 30.9,
      "dec_deg": "-01",
      "dec_min": 3,
      "magnitude": 6.3,
      "distance": 50
    },
              .
       .
  ]
}
```

Z odpovědi je patrno, že server poslal seznam objektů včetně jejich dat. Aplikace si tak může aktualizovat svoji interní databázi. Potencionální problém může nastat tehdy, pokud databáze objektů obsahuje stovky nebo tisíce záznamů. V takovém případě by odpověď serveru byla příliš veliká což není požadováno. Z toho důvodu se musí zavézt do odpovědi serveru limit na počet stránek. V tomto případě není nutné, aby si limit záznamů řídila aplikace, která data požaduje. V nastavení serveru je proměnná, která určuje počet záznamů na stránku. Pokud má tato proměnná hodnotu 10, server vždy vrátí prvních 10 záznamů. Při aktualizaci katalogu musí aplikace obdržet všechny záznamy a ne, jen prvních 10. Vzniká tak potřeba nějakým způsobem oznámit serveru, kterou stránku aplikace požaduje. To zajistí zavedení query parametru *page*, který udává číslo stránky balíku dat o limitu 10, který server odešle jako odpověď. Url s požadavkem na 2. stránku záznamů pak bude vypadat takto:

```
GET /messierData?page=2
```

Pokud chce aplikace stáhnout celý objem dat, musí server aplikaci sdělil, kolik stránek budou kompletní data obsahovat a která stránka je poslední. Lze to zajistit např. přidáním dalšího query parametru, který obsahuje počet stránek, nebo informaci, zda je aktuální stránka poslední. Klientská aplikace si tak může určit, zda bude požadovat další stránku, nebo ne.

Serverová aplikace, vkládá tyto informace do hlavičky HTTP odpovědi. Vztah mezi stránkou 1 a stránkou 2 lze chápat jako relaci mezi těmito stránkami. Relace lze v REST API řešit pomocí parametru *Link* HTTP hlavičky. Podle specifikace [29] je Link typ spojení mezi dvěma webovými zdroji, které jsou definovány pomocí jednoznačného identifikátoru. Pokud tedy mobilní aplikace požaduje stáhnutí všech dat, obdrží nejdříve první stránku. HTTP hlavička odpovědi obsahuje parametr Link obsahující URL další stránky a parametrem *rel* určující typ relace.

```
Link: <http://astro.8u.cz/messierData?page=2>;rel="next"
```

Parametr *rel* s hodnotou “next” oznamuje aplikaci, že existuje relace mezi URL odeslanou klientivi a URL v HTTP hlavičce Link. Aplikace tedy má přímo v odpovědi URL další stránky. Může jí tedy například v cyklu pro stažení kompletního balíku dat.

Pro úplnost ještě HTTP hlavička obsahuje parametr *X-Total-Count* určující celkový počet záznamů. Hlavička obsahuje tento parametr jen pro úplnost, ale není aplikací pro mobilní zařízení nijak využit.

Server je napsaný v jazyku PHP a pro vkládání parametrů do HTTP hlaviček je využit framework Nette, který práci s HTTP protokolem výrazně usnadňuje:

```
protected function setPaginationToHeader(){
	$count = $this->paginator->getItemCount();
	$page = $this->paginator->getPage();
	$this->getHttpResponse()->addHeader('X-Total-Count', $count);
	if (!$this->paginator->last) {
		$url = $this->getHttpRequest()->getUrl();
		parse_str($url->getQuery(), $query);
		$query['page'] = ++$page;
		$url->appendQuery($query);
		$link = new Link($url, Link::NEXT);
		$this->getHttpResponse()->addHeader('Link', $link);
	}
}
```

5. Implementace v prostředí Android

Prostředí Android nabízí celou řadu nástrojů, které je možno využít k synchronizaci mezi serverem a mobilní aplikací. Důležitá je schopnost práce s internetovým spojením, schopnost odesílat a přijímat zprávy pomocí protokolu HTTP. Synchronizace by neměla probíhat v hlavním vlákně, Android tuto komunikaci ani nepovoluje (chyba NetworkOnMainThreadException). Z toho důvodu je důležitá schopnost obstarat téměř veškerou komunikace mezi serverem a aplikací ve vlákně spuštěném na pozadí. Neméně důležité jsou i schopnosti zpracovat a vytvářet data uložená v XML, nebo ve formátu JSON. Data obdržená ze serveru je třeba zpracovat a uložit v aplikaci, která je vhodná pro použití v těchto případech. K tomuto účelu lze využít databázi SQLite.

5.1. Komunikace se serverem

Synchronizace dat mezi aplikací v mobilním zařízení a serverem musí probíhat podle pravidel API popsaných v 3.5. Aplikace musí pracovat s HTTP protokolem.

Důležité schopnosti jsou zejména tyto:

- Pracovat s internetovým připojením
- Práce s asynchronními procesy
- Odeslat HTTP požadavek na server
- Zpracovat odpověď severu

5.1.1. Internetové spojením

Před odesláním požadavku na server je třeba zjistit, zda je k dispozici internetové spojení. Bez jeho existence nemá význam volat metody potřebné k synchronizaci.

Pro zajištění možnosti aplikaci používat internetové spojení je třeba, aby manifest obsahoval následující povolení.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Prostředí Android obsahuje klienta *HttpURLConnection*, který podporuje HTTPS, streamování, upload a download, timeout connection, IPv6, atd. [13]. S jeho pomocí je možno otestovat internetové spojení.

```
ConnectivityManager cmr =
    (ConnectivityManager) context.getSystemService(Activity.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = cmr.getActiveNetworkInfo();

if (!(networkInfo != null && networkInfo.isConnected())) {
      sendStatus("Internetoré spojení není k dispozici");
} else {
//Zahájení synchronizace
.
.
.
}
```

5.1.2. Asynchronní procesy

Ne vždy je možno dopředu předpovědět, jak dlouhou dobu zabere vykonání online požadavku a jak dlouho se bude čekat na odpověď. Uživatel by musel čekat na dokončení operace a jeho uživatelské rozhraní by po tuto dobu bylo nepoužitelné, docházelo by k "zamrzání" aplikace. Z toho důvodu je v prostředí Android vyžadováno, aby veškeré online operace běžely asynchronně.

Nejjednoduší způsob, jak toto zajistit, je použít třídu *AsyncTask* framworku Android. Tato abstraktní třída obsahuje metodu *doInBackground()* ve které je možno spustit asynchronní vlákno.

Součástí Activity, která zajišťuje připojení k internetu, může být privátní třída odvozená od *AsyncTask*. Do metody *doInBackground()* se pak vloží kód, který využívá připojení k internetu. Spuštění této třídy je možné po vytvoření instance metodou *execute()* [16, s 258].

```
Synchronization sync = new TestAsyncTask(context);
sync.execute();
```

Spouštění asynchronních procesů

Hlavním tématem této práce je synchronizace dat mezi mobilní aplikací a vzdáleným serverem. Synchronizační proces bude běžet na pozadí, a navíc bude komunikovat pomocí internetu se serverem. Proto je nutné, aby běžel v asynchronním vlákně. Nastává tak několik problémů.

1. Kdy synchronizaci spustit
2. Zajisti jedinečnost procesu synchronizace

Jednou z možností, jak spustit synchronizaci je přenést za ní odpovědnost na uživatele. Aplikace tak bude obsahovat tlačítko, které spustí synchronizaci v čase, kdy to uživatel uzná za vhodné. Toto řešení není vyhovující. Pro větší uživatelův komfort je třeba, aby veškerou synchronizaci řídila aplikace. Uživatel by měl mít pouze možnost synchronizaci zcela vypnout nebo určit intervaly v jakých bude probíhat. Aplikace tedy musí spouštět synchronizaci v předem daných časových intervalech. K tomuto účelu je možno použít metodu *postDelayed()* třídy *Handler* [14]. Spouští se v *AbstractBaseActivity*, tedy třídy, ze které vycházejí všechny ostatní Activity použité v aplikaci.

```
static boolean runing = false;
private Runnable mHandlerTask = new Runnable() {
  @Override
  public void run() {
    runing = true;
      try {
        Date date = new Date();
        Synchronization sync = new Synchronization(getContext());
        sync.execute();
        mHandler.postDelayed(mHandlerTask, 30000);
      } catch (Exception e) {
      e.printStackTrace();
      runing = false;
    }
  }
};

void startRepeatingTask() {
  mHandlerTask.run();
}
```

Proces se spouští pomocí metody *startRepeatingTask()* v metodě *onCreate()*. Každých 30000 ms, tedy každou půl minutu, se spouští v dalším asynchronním vlákně synchronizace. Metoda *onCreate()* je volána systémem pokaždé když je Activity přenesena na popředí, nebo i při změně orientace přístroje. Je třeba splnit požadavek, aby se proces, spouštějící synchronizaci, běžel vždy v jednom vlákně. Pro tento účel je vytvořena v *AbstractBaseActivity* statická proměnná *running*. Při spuštění procesu se její hodnota nastaví na true, při skončení procesu chybou je proměnná opět nastavena na hodnotu *false*. Podle této proměnné může Acitivity rozhodnout, zda se má proces spustit či ne. V reálném případě by interval spouštění synchronizace byl delší a jeho hodnota by se načítala z uživatelského nastavení aplikace. Popsané řešení je funkční a vyhovuje základním požadavkům této práce.

Komunikace mezi vlákny

Synchronizace je spuštěný proces běžící v jiném vlákně než hlavní proces, se kterým komunikuje s uživatelem. Není tedy možné posílat přímo zprávy mezi těmito vlákny. Třída *AsyncTask* obsahuje dvě metody, ve kterých je tako komunikace umožněna.

Metoda *onPostExecute()* běží v hlavním vlákně a je volána po ukončení procesu spuštěného metodou *doInBackground()*, z ní je pak například možné upozornit uživatele na ukončení procesu běžícího na pozadí.

Metodu *publishProgress()* je možné volat přímo z metody *doInBackground()*. Je zde vhodné posílat informace do hlavního vlákna. Uživatel může dostávat informace o stavu procesu běžícím na pozadí.

Další možností, jak posílat zprávy mezi jednotlivými vlákny je odeslání broadcast zprávy z asynchronního vlákna. Toto je vhodné, pokud má zprávu obdržet jen jedna určitá Activity. Například po stažení nových dat ze vzdáleného serveru a jejich uložení do interní databáze přístroje, vznikne požadavek na opětovné načtení seznamu objektů, aby se aktuální data zobrazila uživateli ihned po jejich stažení. Proces v asynchronním vlákně použije *metodu sendBroadcast()*:

```
Intent intent = new Intent();
intent.setAction(ObjectListActivity.REFRESH_OBJECTS);
context.sendBroadcast(intent);
```

Tímto způsobem se odešle zpráva s filtrem *ObjectListActivity.REFRESH_OBJECTS*, který slouží pro identifikaci typu zprávy. Aby zprávu aktivity správně identifikovala a mohla na ní odpovídajícím způsobem reagovat, musí mít registrovaný *BroadcastReceiver*. Activita v tomto případě obsahuje třídu odvozenou od třídy *BroadcastReceiver*.

```
private class ObjectReloader extends BroadcastReceiver {
	@Override
	public void onReceive(Context context, Intent intent) {
		createObjects();
	}
}
```

Na vhodném místě *ObjectListActivity*, například v metodě *onCreate()*, se zaregistruje instance třídy *ObjectReloader* s odpovídajícím filtrem:

```
registerReceiver(new ObjectReloader(), new IntentFilter(REFRESH_OBJECTS));
```

Po obdržení broadcastu s filtrem REFRESH_OBJECTS, je v Activity prostřednictvím *BroadcastReceiveru* spuštěna metoda *createObjects(),* která provede reload objektů a tím zobrazení aktuálních dat.

5.1.3. Odeslání HTTP požadavku

V prostředí Android lze pro odesílání HTTP požadavků použít třídu *HttpURLConnection*, která podporuje HTTPS, streamování, odesílání a stahování dat apod. Instanci lze získat z třídy *java.net.URL* metodou *openConnection().*

```
URL url = new URL(apiUrl);
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
```

Typ spojení se definuje metodou *setRequestMethod()* s parametrem POST, GET, PUT, atd.

```
conn.setRequestMethod("POST");
```

Při práci s REST API je důležité nastavit správný „Content-Type“ a znakovou sadu. K tomuto účelu lze použít metodu setRequestProperty():

```
conn.setRequestProperty("Content-Type", "application/json; charset=UTF-8");
```

Ukázková aplikace používá pro získání *access_tokenu* pomocí přihlašovacích údajů metodu *accessTokenByCredentials(),* které je předáno uživatelské jméno a heslo.

```
public static HttpURLConnection accessTokenByCredentials(
             String login, String password) throws IOException, JSONException {

       URL url = new URL(Config.API_URL + Config.API_TOKEN);
       HttpURLConnection conn = (HttpURLConnection) url.openConnection();
       conn.setRequestMethod("POST");
       conn.setDoInput(true);
       conn.setDoOutput(true);
       conn.setRequestProperty("Content-Type", "application/json; charset=UTF-8");
       DataOutputStream wr = new DataOutputStream(conn.getOutputStream());
       JSONObject obj = new JSONObject();

       obj.put("login", login);
       obj.put("password", password);
       obj.put(Config.API_GRANT_TYPE, "password");
       obj.put("client_id", Config.API_CLIENT_ID);

       wr.writeBytes(obj.toString());
       wr.flush();
       wr.close();

       return conn;
}
```

Metoda vytvoří POST požadavek a do jeho těla vloží potřebná data:

```
POST /token
{
    "login": "prihlasovaci_jmeno",
    "password" : "heslo_uzivatele",
    "client_id" : 1,
    "grant_type" : "password"
}
```

5.1.4. Zpracování odpovědi serveru

Odpověď serveru se získá metodou *getInputStream()* třídy *HttpUrlConnection.*

```
InputStream in = new BufferedInputStream(conn.getInputStream());
```

API vrací odpovědi ve formátu *JSON*, proto je nutno *InputSteam* převézt na vhodný formát. V tomto případě je vhodné použít knihovnu *IOUtils*, která je určená ke zpracování streamu.

```
InputStream in = new BufferedInputStream(conn.getInputStream());
String json = org.apache.commons.io.IOUtils.toString(in, "UTF-8");
```

Řetězec je třeba převézt na objekt třídy *JSONObject* pro snadnější přístup k obsahu:

```
JSONObject jsonObject = new JSONObject(json);
String accessToken = jsonObject.getString("access_token");
String refreshToken = jsonObject.getString("refresh_token");
```

5.2. Autentizace

Pokud aplikace odešle požadavek na server, musí jí autentizovat, dokázat serveru že odeslaná zpráva pochází opravdu od uživatele za kterého se vydává. Server, který odpovídá na požadavky je nazýván Resource server a obsahuje data, která dokáže poskytovat, ukládat nebo aktualizovat pomocí REST API rozhraní. V případě použití autorizace pomocí OAuth2.0, musí Resource server ověřit platnost odeslaného *access_tokenu*. Resource server musí být ve spojení s Autorizačním serverem, který ověřuje platnost tokenů. Obdržený *access_token* je odeslán do Autorizačního serveru a podle odpovědi je určena jeho platnost. Aby mohl uživatel úspěšně komunikovat s Resource serverem, musí být jeho identita nějakým způsobem ověřena na Autorizačním serveru (musí mít založený účet). V některých případech, kdy není vyžadována přílišná robustnost, je Autorizační server a Resource server spojován do jednoho. Ušetří se tak systémový čas na komunikaci se vzdálenou databází Autorizačního serveru.

Pro zajištění autentizace musí aplikace být schopná zajistit nyto procesy:

1. Získání *access tokenu*
2. Získání dat pomocí *access_tokenu*
3. Získání nového *access_tokenu* s použítím *refresh_tokenu*

5.2.1. Získání access_tokenu

Access_token lze z Autorizačního serveru získat několika způsoby viz. 3.7. V případě aplikace nainstalované na mobilním zařízení lze použít jednouchý Client Credentials Grant [17]. Tento způsob je vhodný pro ověřené klienty, tedy pro mobilní nebo desktopové aplikace u kterých je ověřena jejich autenticita. Uživatel těmto aplikacím uděluje právo, aby mohli komunikovat s Resource serverem v jeho zastoupení.

Pro získání *access_tokenu* je třeba ověřit platnost aplikace a uživatele.

Aplikace je ověřena, pomocí *client_id* nebo *client_secret*, což je zpravidla unikátní řetězec znaků generovaný Autorizačním serverem. Bývá uložen přímo v aplikaci a je k dispozici ihned po jejím nainstalování.

Ověření uživatele je provedeno odesláním přihlašovacího jména (loginu) a hesla uživatele, v jehož zastoupení má aplikace komunikovat.

Pokud autorizační server obdrží tyto informace, vygeneruje zprávu obsahující *access_token* a *refresh_token*. Aplikace musí obě informace uložit v paměti zařízení a používat je pro komunikaci s Resource serverem.

Autorizační požadavek se odesílá na server metodou POST, proto je nutné, aby aplikace byla schopna vytvořit HTTP POST, odeslat ho na server a zpracovat jeho odezvu.

5.2.2. Získání dat pomocí *access_tokenu*

Pokud má aplikace uložený *access_token*, může bez dalších ověřování uživatele komunikovat přímo s Resource serverem. *Access_token* se připojí jako parametr do HTTP požadavku, který se odesílá na server. Na serveru pak dojde k ověření platnosti tokenu. Pokud je Resource server zároveň i autorizačním serverem, bývá *access_token* uložen v databázi spolu s časovým otiskem vypršení platnosti. Server tak ověřuje, zda se v databázi nalézá *access_token* a zda jeho platnost ještě nevypršela. Součástí uloženého tokenu může být i identifikační číslo uživatele. *Access_token* tak může sloužit zároveň jako identifikace uživatele.

Pokud chce aplikace v prostředí Android stahovat nebo odesílat data na server, musí před odesláním HTTP požadavku načíst z úložiště dat v zařízení aktuální *access_token* získaný při ověřování uživatele (viz. 5.2.1) a připojit ho jako parametr do URL.

5.2.3. Získání nového *access_tokenu* s použitím *refresh_tokenu*

Při komunikace s Resource serverem nutně dojde k ukončení platnosti *access_tokenu*. Doba jeho platnosti je kvůli zajištění bezpečnosti omezená. Aplikace v tomto případě obdrží chybu s HTTP Statusem 401 Unauthorized. Tím dává server na vědomí, že obdržený *access_token* již nadále není platný a nelze ho použít k autentizaci. Je třeba získat platný *access_token*.

Z kapitoly 5.2.1 je patrné, že při první úspěšné autentizaci, odešle server spolu s *access_tokenem* ještě *refresh_token*. Lze ho použít k získání nového aktuálního *access_tokenu*. Platnost *refresh_tokenu* trvá zpravidla delší časový úsek, než je tomu u *access_tokenu*, může být i neomezená.

Pokud tedy aplikace obdrží HTTP Status 401 Unahthorized, musí získat nový *access_token* pomocí *refresh_tokenu*.

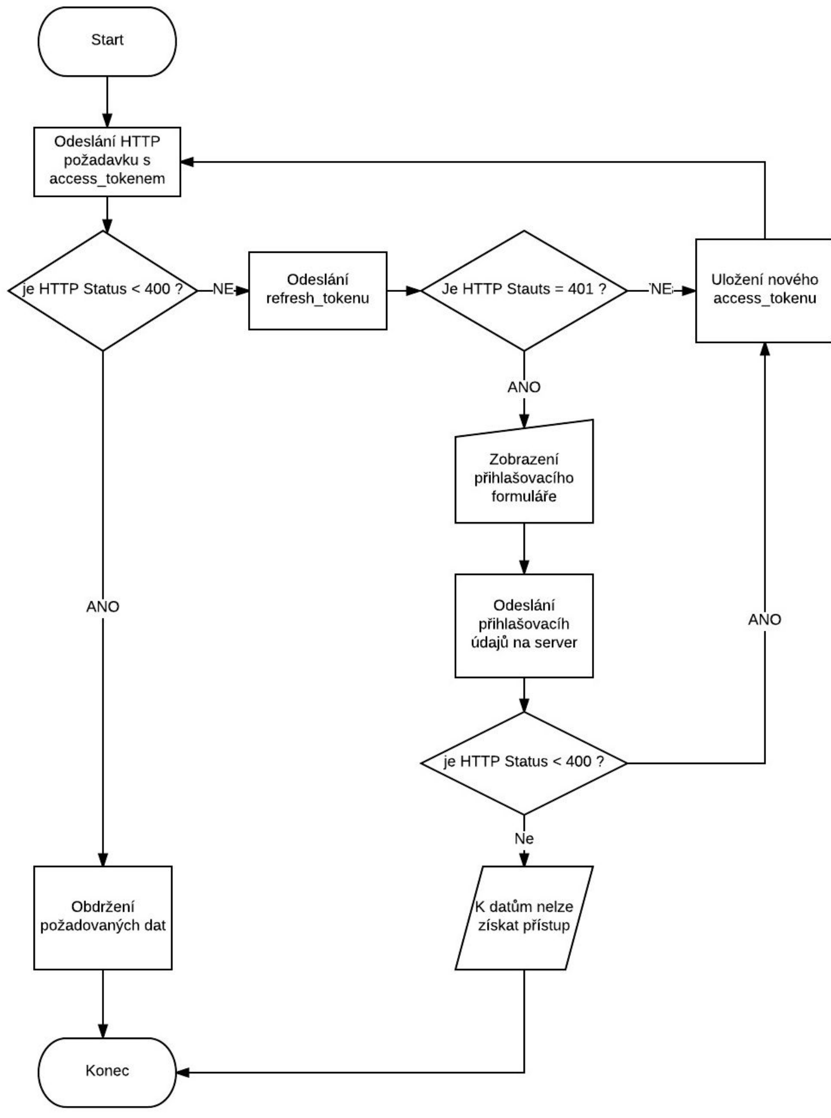
```
POST /token
{
    "refresh_token": "07a49c21ef93c767cb619e6eb6087053859269ac7a346c44f8f83a64f2",
    "client_id" : 1,
    "grant_type" : "refresh_token"
}
```

Při úspěchu vrátí server nově vygenerovaný access_token, který je třeba uložit a používat při všech dalších požadavcích na sever.

Refresh_token nemusí mít vždy neomezenou platnost, proto se může stát, že server odmítne požadavek splnit. Odpověď pak vypadá takto:

```
{
  "code": 401,
  "status": "error",
  "message": "refresh token not found or expired"
}
```

Server oznamuje neplatnost *refresh_tokenu*. V tomto případě je třeba, aby aplikace zobrazila uživateli formulář se zadáním přihlašovacího jména a hesla jako v kapitole 5.2.1



Obrázek 7: Získání access tokenu, zdroj: vlastní

5.3. Synchronizace dat

Řešení synchronizace a autentizace popsané v 5.10 a 5.2 je přímé, ale do určité míry naivní. Daleko lepší robustnost poskytuje využití přímo prostředí Android, které obsahuje komponenty, které tyto případy řeší. Aplikace, která je součástí této práce vyžívá pro autentizaci systémový *AccountAuthenticator* a pro synchronizaci *SyncAdapter*.

5.3.1. Autentizace

Prostředí Android nabízí k autentizaci vestavěný *AccountAutenticator*. S jeho pomocí lze vložit účet přímo do správce účtů v nastavení přístroje. Ihned po instalaci aplikace je možno v systémové nabídce přístroje najít možnost přidání účtu, v tomto případě typu *AstroCatalog*. Pokud není žádný účet tohoto typu v přístroji vytvořen, zobrazí aplikace při svém prvním spuštění formulář se zadáním přihlašovacího jména a hesla. Při úspěšném vytvoření účtu, aplikace uloží autentizační údaje (tokeny) přímo do *AccounManageru*, odkud mohou být kdykoliv znovu vyvolány a použity.

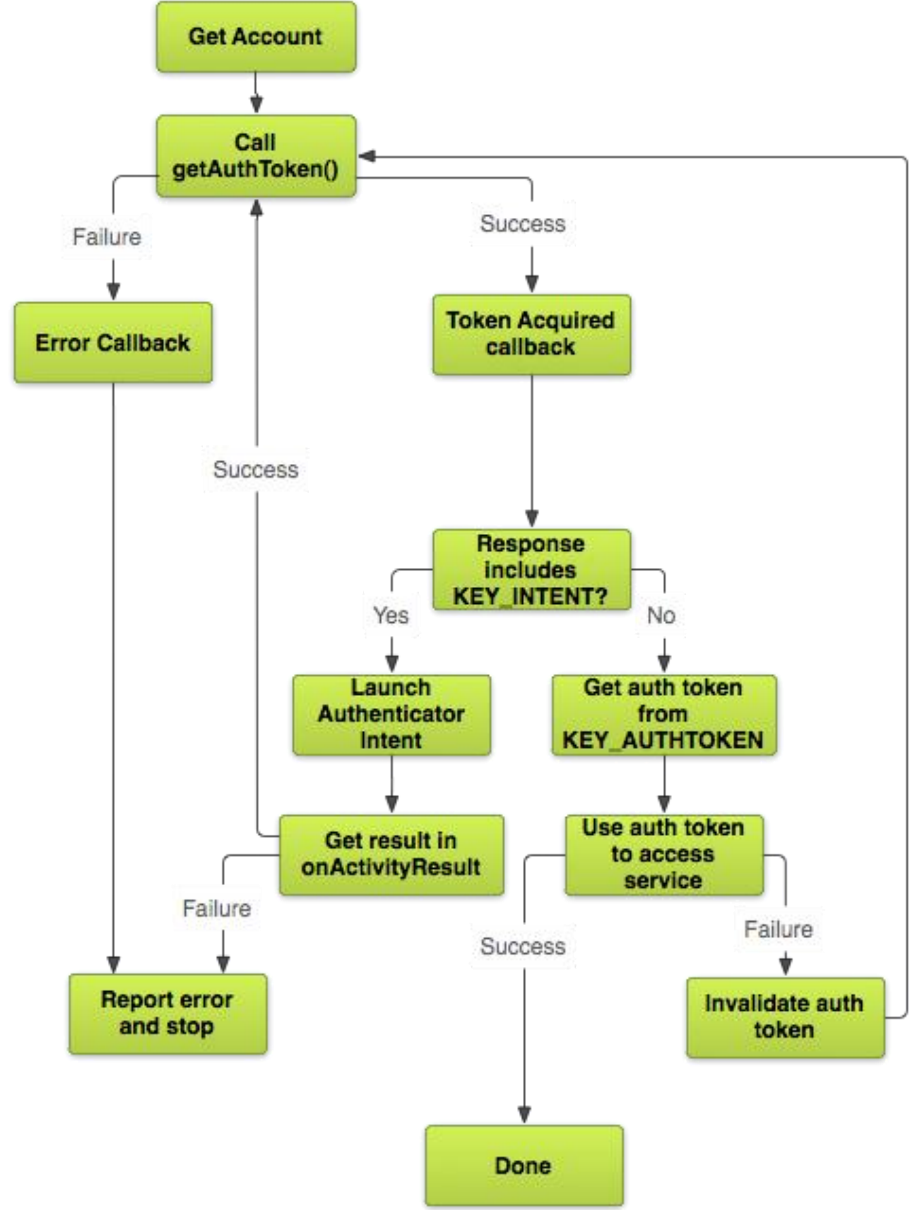
Aplikace komunikuje se serverem odkud stahuje nebo odesílá data. Na serveru je pro ověření použit protokol OAuth. Aplikace potřebuje pro komunikaci se serverem aktuální token.

Pro tento proces jsou důležité tři komponenty:

- AccountManager
- AccountAuthenticator
- AccountAuthenticatorActivity

AccountManager

AccountManager je součástí Android API a dovoluje aplikaci zobrazit seznam účtů registrovaných v systému. Aplikace může spravovat účty a ukládat hesla. Pro synchronizaci je velice důležitá schopnost zjistit *acces_token* a pracovat s ním ve stylu OAuth. To znamená, že není nezbytně nutné ukládat hesla k on-line službám v aplikaci. Detailní použití OAuth prostřednictvím *AccountManageru* zobrazuje Obrázek 8: Získání access_tokenu, zdroj: [18].



Obrázek 8: Získání access_tokenu, zdroj: [18]

AccountManager zajišťuje přístup k centrálnímu registru online účtů [19]. Uživatel zadá svoje přihlašovací údaje pouze jednou, tím zajistí přístup k online zdrojům bez nutnosti dalšího zadáváni uživatelského jména a hesla. Různé online služby mohou používat různé způsoby přihlašování, *AccountManager* může podle typu účtu používat různé autentikátory třetích stran.

Jedním z parametrů metody *addAcount()* je accoutnType, podle něhž je určen správný autenticator který se použije při vkládání nového účtu. Pokud aplikace potřebuje získat aktuální token, zavolá metodu *getAuthToken()*, která vrátí *access_token* uložený v aktivním účtu. Za určitých okolností je třeba *access_token* zneplatnit (invalidovat). Metoda *invalidateAuthToken()* zneplatní aktuální *access_token*. Toto se děje například tehdy, pokud server při požadavku vrací chybu s HTTP Statusem 401 Unauthorized. Znamená to, že došlo k expiraci *access_tokenu*.

AccountAuthenticator

Před použitím *AccountAuthenticatoru* je třeba věnovat zvýšenou pozornost správnému nastavení Manifestu.

Je třeba vložit několik oprávnění, která zajistí aplikaci práva manipulovat s účty.

android.permission.AUTHENTICATE_ACCOUNTS - uděluje aplikaci právo vytvořit účet uživatele.

android.permission.WRITE_SYNC_SETTINGS - systém Android rozlišuje mezi právem vytvořit nový účet a právem měnit jeho nastavení. Proto je potřeba udělit ještě toto oprávnění, bez kterého by aplikace nemohla měnit nastavení účtu, který byl právě vytvořen.

android.permission.GET_ACCOUNTS – dovoluje aplikaci zobrazit seznam známých účtů, které se váží ke konkrétní službě.

android.permission.MANAGE_ACCOUNTS – dovoluje editovat účty, které nebyly vytvořeny aplikací. [20, str. 146]

V prostředí Android je účet daného typu definován existencí servisní komponenty, která musí splňovat několik základních požadavků. Deklarace služby identifikuje komponentu, kterou Android použije pro vytvoření, konfiguraci a autentizaci jednoho nebo více uživatelských účtů deklarovaných uživatelských typů (account_type). Aby systém rozpoznal, kterou službu má použít pro který uživatelský typ, musí být v manifestu deklarovány následující části:

- služba, která v metodě *onBind()* vrací instanci třídy *AbstractAccountAuthenticator*
- *intent-filter* pro *android.accounts.AccountAuthenticator*
- referenci na soubor s metadaty *AccountAuthenticatoru*
- soubor s metadaty musí obsahovat element *account-authenticator*
- *account-authenticator* element musí obsahovat atribut *android:accountType*

Dokud neexistuje account authenticator, neexistuje account type. Pokud neexistuje account type, nelze založit účet. Pokud neexistuje účet, nelze použít synchronizační službu k synchronizaci dat.

V manifestu je služba definována pomocí intent-filteru, to znamená, že službu mohou používat externí aplikace. Je to jeden z hlavních důvodů proč se intent-filter deklaruje. Pro account authenticator je však lepší zakázat jeho použití externími aplikacemi. Je toho dosaženo atributem exported="false". Toto je nejbezpečnější nastavení authenticatoru [20, str. 147].

```
<service android:name="cz.uhk.machacek.Service.AuthenticatorService"
      android:exported="false">
      <intent-filter>
            <action android:name="android.accounts.AccountAuthenticator" />
      </intent-filter>
      <meta-data
                android:name="android.accounts.AccountAuthenticator"
                android:resource="@xml/authenticator" />
</service>
```

Odkaz na soubor s metadaty, který je definován elementem meta-data, musí obsahovat atribut *name* s hodnotou *android.accounts.AccountAuthenticator*. Element musí také obsahovat atribut *resource* odkazující na soubor s metadaty.

res/xml/authenticator.xml

```
<account-authenticator xmlns:android="http://schemas.android.com/apk/res/android"
    android:accountPreferences="@xml/pref_auth"
    android:accountType="@string/accountType"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:smallIcon="@drawable/ic_launcher" />
```

res/values/strings.xml

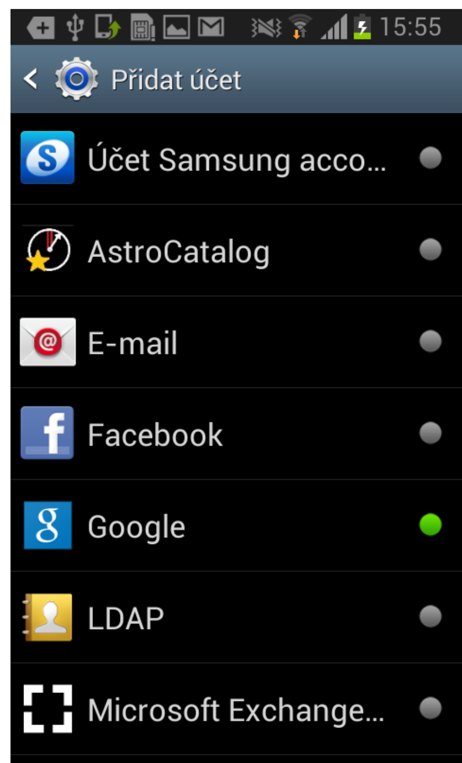
```
<!-- accountType -->
<string name="accountType">cz.uhk.machacek.astroCatalog</string>
```

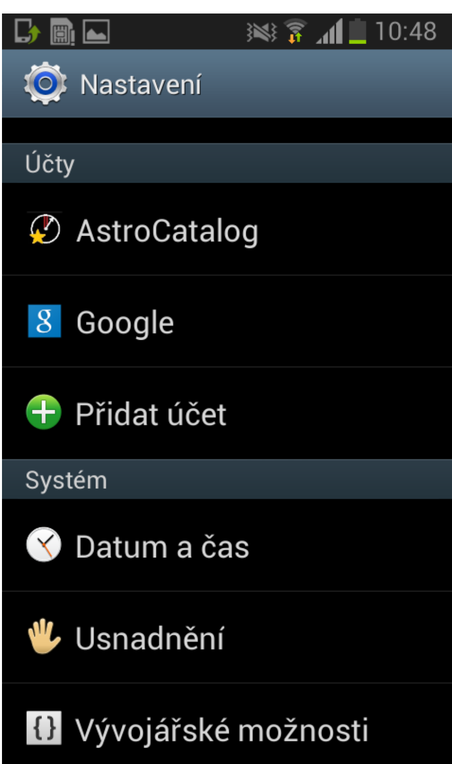
Soubor s metadaty obsahuje několik položek. Jejich význam je patrný z pojmenování atributů. Jsou to typ, název a ikony účtu. Dále obsahuje odkaz na soubor s nastavením formuláře, který se bude zobrazovat v nastavení účtu.

Použití AccountAuthenticatoru

V nastavení zařízení je seznam účtů, které jsou v zařízení vytvořeny. Nabídky „Přidat účet" zobrazí všechny typy účtů, které lze v zařízení vytvořit. Ukázková aplikace používá typ účtu s labelem AstroCatalog. Po nainstalování se objeví v seznamu účtů, které lze vytvořit viz. Obrázek 9: Správa účtů nalevo. Vedle ikony a názvu je indikátor pro stav účtu. Pokud v zařízení není vytvořen ještě žádný účet tohoto typu, je indikátor neaktivní. Pokud uživatel klikne na tento účet, *AccountAuthenticator*, který zobrazí formulář pro vytvoření účtu. Tento formulář lze explicitním voláním *AccountManageru* zobrazit i přímo z aplikace. V případě ukázkové aplikace je po jejím spuštění zkontrolováno, zda se v zařízení nalézá aktivní účet typu *AstroCatalog*. Pokud tomu tak není, je zobrazen přihlašovací formulář.

Po úspěšném vytvoření nového účtu je zobrazen v seznamu aktivních účtů viz. Obrázek 9: Správa účtů vpravo.





Obrázek 9: Správa účtů, zdroj: vlastní

Při instalaci, aplikace je nový typ účtu definován v manifestu. Další informace (label, ikona) obsahuje soubor s metadaty. Pokud je vše správně nastaveno, zobrazí framework nový účet v seznamu.

Implementace AccountAuthenticatoru

Vytvoření nového účtu zajistí AccountManager, který je součástí Android frameworku. Voláním metody addAccount() je spuštěn proces, který propojuje AccountAuthenticator a odpovídající službu. Framework zná typ účtu, který se má vytvořit a podle něho najde odpovídající službu.

```
public class AuthenticatorService extends Service {
    @Override
    public IBinder onBind(Intent intent) {
        AstroAccountAuthenticator authenticator =
                              new AstroAccountAuthenticator(this);
        return authenticator.getIBinder();
    }
}
```

AccountAutenticator musí být potomek třídy *AbstractAccountAuthenticcator* a musí obsahovat metodu *getBinder()*, která vrací objekt implementující rozhraní *IBinder*. Tuto metodu volá služba v jeho *onBind()* metodě.

Ze svého abstraktního předka, implementuje AccountAuthenticator sedm metod: *addAccount*, *getAuthToken*, *updateCredentials*, *hasFeatures*, *confirmCredentials*, *editProperties* a *getAuthTokenLabel*. Pro využití autorizačního procesu pomocí OAuth jsou nezbytné pouze dvě z těchto metod.

Metoda *addAccount()* zajistí vytvoření správného účtu. Pomocí *Intentu* je spuštěna odpovídající *AutenticatorActivity*, která zobrazí formulář pro zadání přihlašovacích údajů. Po jejich obdržení, uloží získané tokeny spolu s nově vytvořeným účtem do *AccountManageru*.

```
public Bundle addAccount(AccountAuthenticatorResponse accountAuthenticatorResponse, String accountType, String authTokenType, String[] strings, Bundle bundle) throws NetworkErrorException {

        final Intent intent = new Intent(context, AuthenticatorActivity.class);
        intent.putExtra(AuthenticatorActivity.ACCOUNT_TYPE, accountType);
        intent.putExtra(AuthenticatorActivity.AUTH_TYPE, authTokenType);
        intent.putExtra(AuthenticatorActivity.NEW_ACCOUNT, true);
        intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE,   accoun-
tAuthenticatorResponse);

        final Bundle bund = new Bundle();
        bund.putParcelable(AccountManager.KEY_INTENT, intent);
        return bund;
    }
```

Metoda *getAuthToken()* je volána pouze v případě, že platnost *access_tokenu* je explicitně zrušena metodou *AccountAutenticator.invalidateAuthToken()*. Pokud aplikace vyžaduje na *AccountAuthenticatoru access_token* a ten je invalidován (zneplatněn) volá se právě tato metoda. Její tělo by mělo obsahovat kód, který zajistí získání nového *access_tokenu*.

Například pokud aplikace používá k autorizaci OAuth, je k získání nového *access_tokenu* použit *refresh_token* uložený v aktivním účtu. Pokud se stane, že tato operace selže kvůli ukončení platnosti *refresh_tokenu*, je možno prostřednictvím *AccountAuthenticatorActivity* zobrazit formulář pro zadání přístupového jména a hesla. Pomocí těchto údajů je z autorizačního serveru získán nový *access_token* a *refresh_token*. Ty jsou pak uloženy pomocí *AccountManageru* k příslušnému účtu uživatele. Toto vše je prováděno v metodě *addAccount()*.

AccountAutenitcatorActivity

Tutu třídu implementuje Activity, která je vytvořena a volána modulem *AccountAutenticator*. Jejím úkolem je zobrazit formulář s požadovanými přihlašovacími údaji. Poté co je uživatel zadá, jsou odeslány do API, pro získání *access_tokenu* a *refresh_tokenu*. Tokeny jsou uloženy pomocí *AccountManageru* k příslušnému uživateli.

AccountAuthenticator musí do Activity předat požadavek následujícím způsobem:

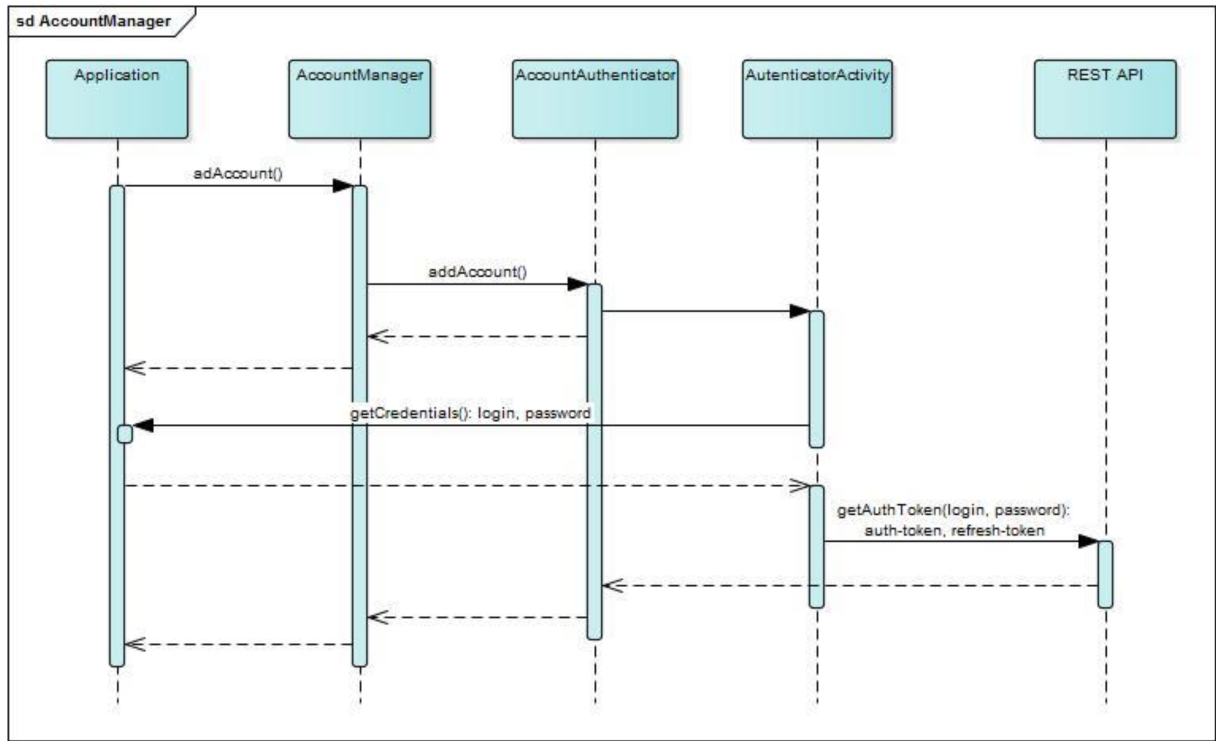
```
intent.putExtra(KEY_ACCOUNT_AUTHENTICATOR_RESPONSE, response);
```

Activita po spuštění zajistí vše potřebné, připojí se k serveru pomocí autentizačních údajů, uloží vrácené tokeny, vytvoří účet uživatele apod. Data, která je nutno vrátit zpět do AccountAuthenticatoru, jsou vložena do třídy *Bundle()* a poslána metodou *setAccountAuthenticatorResult(Bundle data)*. Po ukončení všech akcí je třeba volat metodou *finish()*, která zajistí předání dat a ukončení Activity [15].

Vytvoření účtu

Po úspěšném vytvoření, je možno aktivní účet požadovaného typu nalézt v seznamu účtů, jak je patrno z Obrázek 9: Správa účtů.

Životní cyklus vytvoření účtu



Obrázek 10: Vytvoření účtu, zdroj: vlastní

- Aplikace požádá *AccountManager* o vytvoření účtu
- *AccountManager* určí správný *AccountAutenticator* a volá na něm metodu *addAccount()*
- *AccountAutenticator* zavolá *AutenticatorActivity*, která zobrazí formulář s přihlašovacími údaji
- Uživatel zadá své přihlašovací údaje
- *AutenticatorActivity* odešle přihlašovací údaje do API a získá tokeny
- *AccountManager* vytvoří nový účet a uloží k němu získané tokeny

5.3.2. SyncAdapter

SyncAdapter je plug-in zajišťující synchronizaci aplikací na pozadí. Je velice vhodné ho použít, pokud aplikace potřebuje synchronizovat svá data se vzdáleným serverem. Je registrován přímo v systému Android, který podle potřeby dokáže spouštět.

Aplikace, které synchronizuje svá data, by měla využívat tento plug-in spíše, než používat vlastní řešení založené na procesech spouštěných v paralelních vláknech. Jeho použití má několik výhod:

1. **Úspora baterie**
   Systém dokáže spouštět synchronizaci v okamžiku, kdy běží i on-line synchronizace ostatních aplikací instalovaných v zařízení. Tím se zabrání probuzení zařízení ze svého spánku pro provádění jediného synchronizace.

2. **Použití jednotné interface**
   Všechny *SyncAdaptery* v zařízení jsou přístupné z menu *Nastavení* u typu účtu, na který jsou navázány. Uživatel tak může z jednoho místa měnit preference synchronizace, sledovat, zda nedošlo při synchronizaci k problémům, nebo synchronizaci deaktivovat.
3. **Povědomí o obsahu**
   Pokud se používá *ContentProvider* pro přístup a manipulaci s daty, může *SyncAdapter* sledovat, zda došlo k úpravě dat a spouštět synchronizaci se serverem pouze pokud je to skutečně potřeba.
4. **Retry mechanismus**
   SyncAdapter obsahuje mechanizmus kterým je možno opakovaně spouštět synchronizační procesy při kterých došlo k problémům pomocí prodloužením střední doby prodlevy po každém pokusu na dvojnásobek.

Způsoby synchronizace

Velikou výhodou modulu *SyncAdapter* je plánování synchronizace. Nabízí několik způsobů, jak zajistit čas a periodu synchronizace. Lze ho nastavit tak aby se spouštěl v pravidelných časových intervalech, nebo v určitou část dne. Také ho lze spustit po provedení určité operace s daty v content provideru. Neměl by být spouštěn jako přímý výsledek nějaké uživatelské akce, protože tím by byla modulu znemožněna jedena z největších výhod - spouštění synchronizace samotným Android frameworkem.

Synchronizace pomocí *SyncAdapteru* lze spouštět čtyřmi možnými způsoby:

1. spuštění serverem
2. spuštění po změně dat v zařízení
3. spuštění v pravidelných intervalech
4. spuštění po provedení akce uživatelem

Spuštění serverem

Synchronizace se spustí poté, co aplikace obdrží zprávu ze serveru, která indikuje, že data na serveru byla změněna.

Pro spuštění *SyncAdapteru*, musí server odeslat speciální zprávu do *BroadcastReceiveru* v aplikaci. Jako odpověď na tuto zprávu se volá *ContetntResolverer.requestSync()* a ten udělí povel frameworku, aby se spustil SyncAdapter.

Google Cloud Messaging (GCM) poskytuje obě potřebné části, serverovou i aplikační. Používání GCM pro spouštění synchronizace je vhodnější než dotazování se serveru, zda nemá nějaká změněná data. Dotazování se serveru na změněná data, vyžaduje běžící službu, která je stále aktivní. GCM používá *BroadcastReceiver*, který je aktivován pouze pokud přijde zpráva

ze serveru. Tím je zajištěno, že se synchronizace bude spouštět pouze tehdy, kdy je to nutné (data na serveru jsou změněná).[1]

Spuštění po změně dat v zařízení

Synchronizace se spustí poté, co byla změněna přímo v zařízení. Umožní upravená data ze zařízení odeslat na server. Tato volba je zvláště vhodná tam, kde mají být neustále aktuální data. Podporu této volby obsahuje *ContentProvider* svázaný s tímto typem účtu. K tomu je třeba registrovat *observer* pro *ContentProvider*. Když jsou data změněna, *ContentProvider* volá *observer*. V *observeru* je volána metoda *requestSync(),* která spustí příslušný SyncAdapter.

Pro vytvoření observeru, je nutno extendovat třídu ContentObserver, který obsahuje dvě důležité metody, *onChange()* a *onCreate().*

V metodě *onCreate()* je registrován observer k příslušnému *ContentProvideru* [30].

```
@Override
protected void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);
	mResolver = getContentResolver();
	mUri = new Uri.Builder()
			.scheme(SCHEME)
			.authority(AUTHORITY)
			.path(TABLE_PATH)
			.build();
	TableObserver observer = new TableObserver(false);
	mResolver.registerContentObserver(mUri, true, observer);
}
```

Metoda *onChange()* je volána *ContentProviderem* při změně dat. V ní je třeba volat metodu *requestSync()*, která spustí synchronizaci.

Spuštění v pravidelných intervalech

Synchronizace se bude spouštět po uplynutí předem daného časového termínu, nebo bude spuštěna v konkrétní hodinu dne. Synchronizaci lze například spouštět v noci, kdy je server nejméně zatížený a kdy většina uživatelů svoje zařízení nepoužívá. V tento čas se spouští synchronizace i jiných procesů a stahování aktualizací. Synchronizaci je však nutno nastavit tak, aby nebyla spuštěna v ten samý okamžik na všech zařízeních, což by mohlo způsobit zahlcení serveru požadavky.

Pro spuštění této periodické synchronizace, je třeba volat metodu *addPeriodicSync().* Tím se zajistí spouštění po uplynutí zadaného počtu sekund. Uplynulá doba se může lišit o několik sekund. Je to způsobeno tím, že se Android framework pokouší spojit synchronizaci ostatních účtů do jednoho časového okamžiku. Framework se tak pokouší ušetřit spotřebu baterie. Synchronizace se nespustí, pokud není k dispozici internetové spojení.

[1] Když GCM spouští SyncAdaptér odešle přes broadcast zprávu s povelem k synchronizaci, je třeba počítat s tím, ža SyncAdapter spustí synchronizaci najednou ve všech zařízeních, která mají nainstalovanou tuto aplikaci. V tento okamžik, budou všechny tyto aplikace požadovat po serveru aktuální data, což může způsobit velkou zátěž systému na serveru. Aby se tomuto předešlo, je vhodné zajisti spouštění synchronizaci v každém zařízení v jiném časovém okamžiku.

Metoda *addPeriodicSync()* sama o sobě nezajistí spouštění procesů v určitou část dne. Pokud je nutno spouštět synchronizaci v určitý čas, doporučuje se pro tento účel použít třídu *AlarmManager* a její metodu *setInexactRepeating()* pro nastavení konkrétního času. Zároveň je třeba zajistit, aby se synchronizace nespouštěla v přesně danou sekundu dne [22].

Spuštění po provedení akce uživatelem

Spuštění *SyncAdapteru* jako odezvu na určitou akci uživatele je nejméně preferovanou metodou. Framework je navrhnut tak, aby co nejvíce šetřil baterii zařízení. I když se synchronizace spouští periodicky, pokouší se spojit více akcí dohromady. Pokud je synchronizace spouštěna při ukládání dat, je vhodné spustit i synchronizaci, protože energie z baterie je právě používaná k ukládání dat.

Pokud je tedy dovoleno uživateli spustit synchronizaci mimo tyto případy, vede to k neefektivnímu použití energie v okamžiku, kdy to není nezbytně nutné. Obecně by aplikace měla spouštět synchronizaci pouze v případě aktualizace data ne serveru nebo v zařízení, nebo v pravidelných časových intervalech bez zásahu uživatele.

Pokud je přesto nutno spustit synchronizaci přímo uživatelem, je třeba volat metodu *requestSync()*.

Přímé spuštění může obsahovat následující parametry:

SYNC_EXTRAS_MANUAL – okamžité spuštění synchronizace. Ignoruje se nastavení metodou *setSyncAutomatically()*.

SYNC_EXTRAS_EXPEDITED – spustí synchronizaci okamžitě. Pokud by se nepoužil tento parametr, synchronizace by se spustila až během několika sekund, protože by se framework pokoušel optimalizovat využití baterie.

```
public void onRefreshButtonClick(View v) {
      Bundle settingsBundle = new Bundle();
      settingsBundle.putBoolean(
               ContentResolver.SYNC_EXTRAS_MANUAL, true);
      settingsBundle.putBoolean(
               ContentResolver.SYNC_EXTRAS_EXPEDITED, true);
      ContentResolver.requestSync(mAccount, AUTHORITY, settingsBundle);
}
```

Implementace SyncAdapteru

Aby mohl *SyncAdapter* správně pracovat, musí být splněno několik požadavků:

1. musí existovat *AccountAuthenticator*
2. musí být vytvořený *ContentProvider* pro správu dat
3. musí být spuštěná služba, která *SyncAdapter* spouští
4. je třeba zajistit správné nastavení manifestu a souboru s metadaty

Pro *SyncAdapter* je nutno v manifestu udělit tato tři oprávnění:

android.permission.INTERNET – umožňuje přístup na internet, bez tohoto oprávnění se SyncAdapter nebude spuštět.

android.permission.READ_SYNC_SETTINGS – umožňuje aplikaci číst nastavení synchronizace

android.permission.WRITE_SYNC_SETTINGS – zpřístupňuje aplikaci nastavení synchronizace. Je potřebná pokud aplikace nastavuje čas, kdy se bude synchronizace spouštět.

```
<service
      android:name=".Service.AstroDiarySyncService"
      android:exported="true" >
      <intent-filter>
            <action android:name="android.content.SyncAdapter" />
      </intent-filter>
      <meta-data
            android:name="android.content.SyncAdapter"
            android:resource="@xml/sync_adapter_diary" />
</service>
```

V ukázce kódu výše je zobrazen příklad nastavení *SyncAdapteru* v *manifestu*. Je podobné jako nastavení *AccountAuhenticaoru*, obsahuje *intent-filter* a odkaz na soubor s metadaty. Atribut *android:SyncAdapter* má stejnou hodnotu jak pro *intent-filter* tak pro odkaz na metadata: *android.content.SyncAdapter*. Po instalaci aplikace systém Android zaregistruje tuto deklaraci a zpřístupní jí jako službu, která je navázaná na typ účtu definovaný v souboru s metadaty.

```
<?xml version="1.0" encoding="utf-8"?>

<sync-adapter xmlns:android="http://schemas.android.com/apk/res/android"
    android:contentAuthority="cz.uhk.janMachacek.astro.diaryProvider"
    android:accountType="@string/accountType"
    android:userVisible="true"
    android:allowParallelSyncs="false"
    android:isAlwaysSyncable="true" />
```

Soubor s metadaty obsahuje nastavení popisující chování komponenty. Je zde definován typ účtu se kterým je *SyncAdapter* svázaný, řetězec pro tzv. content provider authority se který je asociovaný s aplikací a další parametry popisující chování a viditelnost komponenty uživatelem.

android:accountType – typ účtu se kterým je *SyncAdapter* svázaný. Musí být stejný jako typ účtu který používá *AccountAuthenticator*. Umožňuje frameworku správným způsobem propojit.

android:contentAuthority – URI authority označující ContentProvider. Jeho hodnota musí být stejná jako hodnota atributu android:authrorities v elementu <provider> v manifestu.

android:userVisible – nastavuje viditelnost účtu *SyncAdapteru*. Standardně je ikona typu účtu zobrazovaná v Nastavení přístroje v sekci účty i když žádný účet ještě není vytvořen. Při nastavení na hodnotu *false*, bude typ účtu zobrazován pouze pokud je nějaký účet v zařízení již vytvořen.

android:allowParallelSyncs – dovoluje běh několika instancí SyncAdapteru současně. Toto nastavení je vhodné použít v situaci, kdy k jednou typu je vytvořeno více než jeden účet. Synchronizace tak bude probíhat současně pro každý účet.

android:supportsUploading – dovoluje aplikaci odesílat data, pokud je *false*, budou se při synchronizaci pouze stahovat data.

android:isAlwaysSyncable – pokud je nastaveno na *true*, bude čas synchronizace ovládat framework. Pokud je potřeba ovládat čas synchronizace aplikací, jeho nastavení je *false* a spouští se metodou *requestSync().*

Syncrhonizace pomocí SyncAdapteru

SyncAdapter sám o sobě žádnou synchronizaci nezajistí. Veškerou funkcionalitu je třeba implementovat do metody *onPerformSync()*, která je zděděná od abstraktního *předka AbstractThreadedSyncAdapter*. Tato metoda je volána frameworkem při zahájení synchronizace. Implementace se může lišit podle druhu aplikace a způsobu synchronizace. Zde by se mělo řešit několik hlavních úkolů, které jsou shodné pro všechny implementace:

- Spojení se serverem
- Stahování a odesílání dat
- Řešení konfliktů
- Čištění dat a keší po synchronizaci

Důležitými parametry předanými do metody *onPeformSync()* je *Account* a *ContentProviderClient.*

Account je důležitý pro získání dat ze serveru, tedy pro download. Jedná se o objekt reprezentující účet uživatele, jehož synchronizace byla právě spuštěna.

Druhým důležitým parametrem jen contentProviderClient, což je veřejné rozhraní do ContentProvideru, který je prostřednictvím contentAuthority svázán se SyncAdaptérem. Po stažení dat v prvním kroku je lze uložit jako nová data nebo je aktualizovat.

Parametrem *syncResult* je předán objekt třídy *SyncResult*, která slouží k předávání zpráv o stavu synchronizace mezi *SyncAdapterem* a *SyncManagerem*. Pokud například dojde k při synchronizaci k problémům s autentizací, zvýší se proměnná *syncResult.numAuthExceptions* o jednotku. *SyncManager* je tak upozorněn na problém. K dalšímu spuštění synchronizace dojde až po uplynutí časového úseku, který je závislý na počtu neúspěšných synchronizací. Pomocí proměnné *syncResult.delayUntil* lze nastavit počet sekund kdy bude nejdříve spuštěn další pokus o synchronizaci.

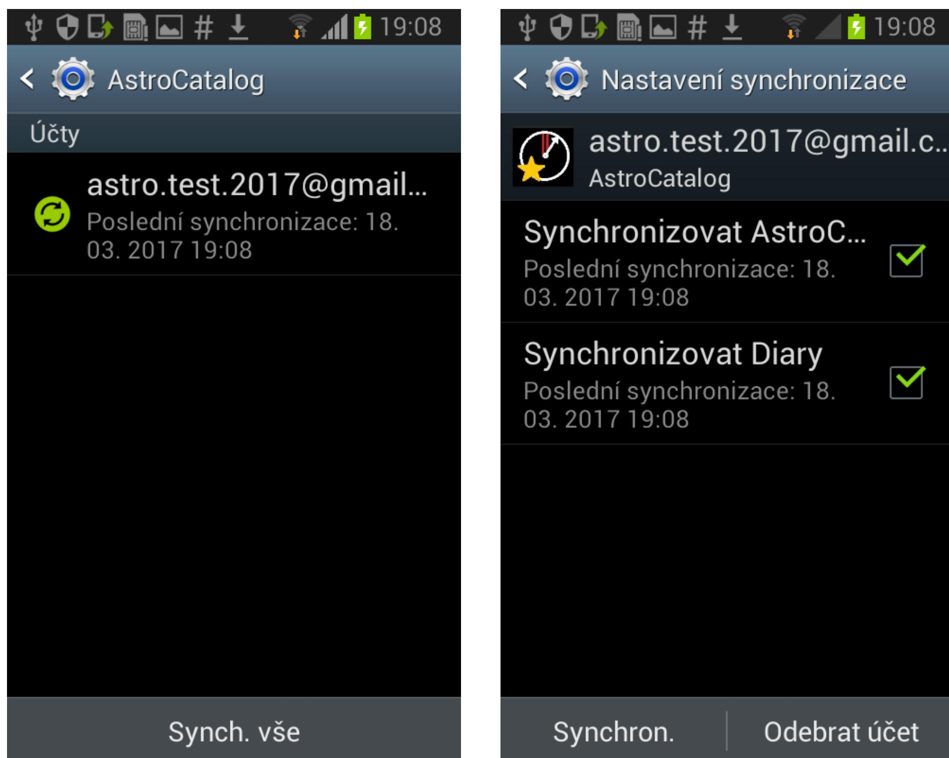
Použití OAuth

Android framework obsahuje podporu pro autentizaci pomocí OAuth. Metoda onPerformSync() by měla obsahovat algoritmus popsaný v Obrázek 7: Získání access tokenu. Pro komunikaci se serverem je v tomto případě nutné odeslat na server aktuální access_token. Ten je spojený s účtem, který je předán do metody *onPerformSync()*. Prostřednictvím *AccountManageru*, který vyžaduje instanci objektu *Account*, lze získa *access_token*, který byl při přihlášení uživatele uložen k jeho účtu atd.

```java
public class DiarySyncAdapter extends AbstractThreadedSyncAdapter {

  private final AccountManager mAccountManager;

  public DiarySyncAdapter(Context context, boolean autoInitialize) {
    super(context, autoInitialize);
    mAccountManager = AccountManager.get(context);
  }
  @Override
  public void onPerformSync(
                Account account,
                Bundle bundle,
                String s,
                ContentProviderClient contentProviderClient,
                SyncResult syncResult) {

    String authToken =
        mAccountManager.blockingGetAuthToken(account, "baerer", true);

    try{
    /* spuštění synchronizace */
    .
    .
    .
    } catch (AccessTokenExpiredException e) {
            mAccountManager.invalidateAuthToken(getContext().get-
String(R.string.accountType), authToken);
             syncResult.numAuthExceptions++;
             syncResult.delayUntil = 30;
    }
  }
}
```

V ukázce kódu výše je příklad implementace OAuth do *SyncAdapteru*. Nejdříve je získán *access_token*. Pokud při synchronizaci dojde k problémům s autorizací, je vyhozena výjimka AccessTokenExpiredException. Při chybě je *access_token* označen jako nevalidní – *accountManager. invalidateAuthToken()*, tím je zajištěno jeho obnovení při příštím volání metody *accountManager. blockingGetAuthToken().* Pomocí *syncResult.numAuthExceptions++* je *SyncManageru* oznámen problém se synchronizací, *syncResult.delayUntil=30* nastaví spuštění příští synchronizace nejdříve za 30 sekund.

5.3.3. Průběh synchronizace



Obrázek 11: nastavení synchronizace, zdroj: vlastní

Synchronizaci lze ovládat přímo v „Nastavení“ přístroje. Pokud proběhla instalace aplikace úspěšně a byl vytvořen účet lze ovládat synchronizaci z menu, které je přístupné ze seznamu aktivních účtů - Obrázek 9: Správa účtů vpravo. Na tomto místě lze sledovat stav synchronizace, vypnout ji či zapnout. Aby se tato volba vůbec zobrazovala, je třeba mít v metadatech SyncAdapteru nastavenou hodnotu *android:userVisible=true*. V opačném případě by synchronizace probíhala na pozadí, uživatel by jí nemohl žádným způsobem zobrazit nebo ovládat.

První obrazovka zobrazuje nastavení pro jeden účet. Je zde název účtu a datum poslední synchronizace. Pokud synchronizace právě probíhá, je to patrné z indikátoru na levé straně viz. Obrázek 11: nastavení synchronizace vlevo. Zde může být i odkaz na další obecné nastavení synchronizace. Tato obrazovka přísluší jednomu určitému účtu a souvisí s *AccountAuthenticatorem*. Pokud aplikace používá i *SyncAdapter*, lze se dostat proklikem z účtu přímo do nastavení synchronizace viz. Obrázek 11: nastavení synchronizace vlevo.

Ukázková aplikace používá dvě komponenty pro synchronizaci Deník a Objekty. Každá komponenta má svůj vlastní *SyncAdapter*, službu a *ContentProvider*. Obě jsou svázány s jedním typem účtu, proto se objevují společně na jedné obrazovce. Zobrazují podobné informace jako předešlá obrazovka. Uživatel zde může vypnou synchronizaci každého modulu, pokud je checkbox zaškrtnutý, synchronizace se bude spouštět automaticky.

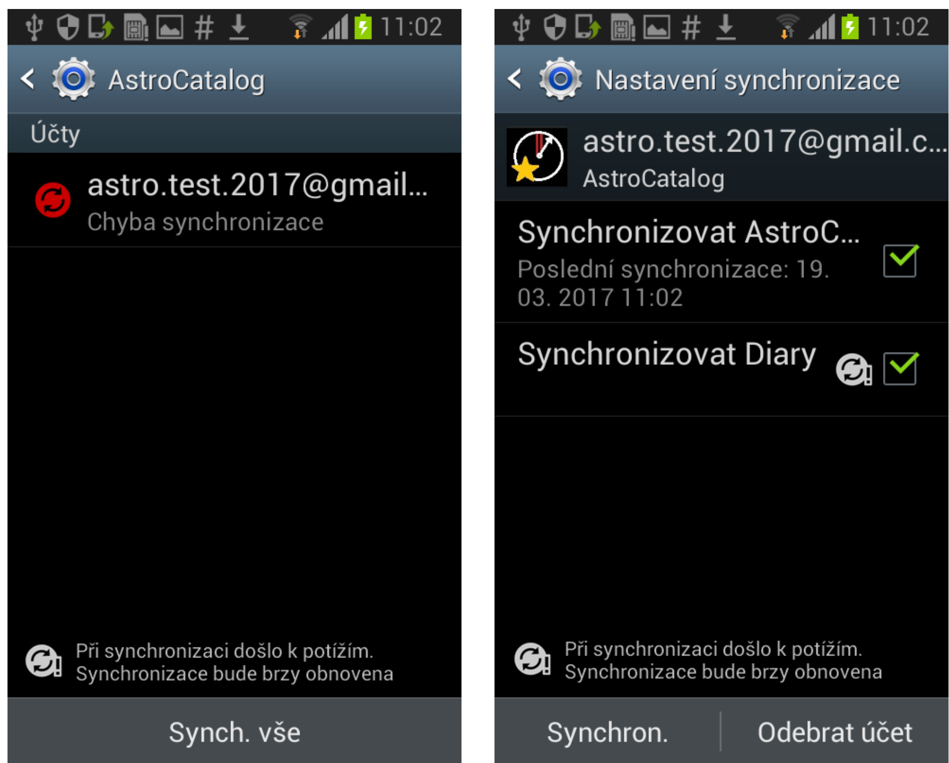
Ve většině případů je vhodné ve výchozím nastavení zapnout po instalaci aplikace automatickou synchronizaci (zaškrtnout checkbox). Toho lze dosáhnout ihned po vytvoření účtu metodou *setSyncAutomatically():*

```
context.getContentResolver().setSyncAutomatically(account, AstroCon-
tract.DIARY_AUTHORITY, true);
```

Podobně lze nastavit periodu mezi jednotlivými synchronizacemi:

```
context.getContentResolver().addPeriodicSync(account, AstroContract.DIARY_AUTHORITY,
new Bundle(), PERIOD_OF_SYNC_SEC);
```

Pokud dojde při synchronizaci k potížím, je na to SyncManager upozorněn objektem třídy --*SyncResult*. Stav synchronizace se na obou obrazovkách změní. U účtu zčervená indikátor. V detailu je patrné, který modul má problémy při synchronizaci.



Obrázek 12: Problém při synchronizaci, zdroj: vlastní

5.4. Vlastní Implementace synchronizace

Ze zadání je patrné, že pro synchronizaci bude aplikace používat dva moduly: synchronizace katalogu objektů a synchronizace astronomického deníku.

5.4.1. Synchronizace katalogu objektů

V tomto případě se jedná o jednosměrnou synchronizaci, data jsou stahována pouze jedním směrem, ze serveru do klienta. Data objektů jsou na serveru označena číslem verze. Při aktualizaci dat dojde k navýšení čísla verze. Klient tak má možnost stáhnout data pouze v případě, že byla data na serveru aktualizována. Po stažení dat, je číslo verze uloženo. Při synchronizaci se klient zeptá serveru, zda existuje nová verze dat, pokud je verze dat na serveru shodná s uloženým číslem verze v klientovi, není třeba nic stahovat.

Synchronizace je spouštěna metodou *onPerformSync() třídy CatalogSyncProvider*. Tato metoda je periodicky spouštěna *SyncManagerem*, probíhá v ní celý proces synchronizace katalogu vesmírných objektů.

Po nainstalování aplikace, nebo při aktualizaci verze na serveru dojde ke stažení celého katalogu ze serveru. V 4.4 je popsáno, jakým způsobem poskytuje API data. Odpověď serveru obsahuje v hlavičce parametr Link, který se použije pro získání celého katalogu. Data jsou získávána pomocí třídy MessierData, která volání API zapouzdřuje. Z odpovědi serveru je třeba přečíst hlavičku obsahující Link na další stránku. Kompletní data jsou získána v privátní metodě getData(), která rekurzívně volá sama sebe:

```
String headerLink = conn.getHeaderField("Link");

if (null != headerLink) {
       String[] link = headerLink.split(";");
       if (link[1].matches("rel=\"next\"")) {
        String nextUrl = link[0].substring(1, link[0].length() - 1);
        getData(nextUrl, astroObjects);
       }
}
```

Když je z API získána nová verze dat, jsou starší data vymazána z databáze a nahrazena novými. Pokud k tomu dojde, je třeba odeslat broadcastovou zprávu:

```
Intent intent = new Intent();
intent.setAction(ObjectListActivity.REFRESH_OBJECTS_LIST);
context.sendBroadcast(intent);
```

V ObjectListActivity, která zobrazuje seznam objektů, je nutné tento broadcast receiver zaregistrovat:

```
registerReceiver(
       new RefreshBroudcastReceiver(),
       new IntentFilter(REFRESH_OBJECTS_LIST)
       );
```

V privátní třídě *RefreshBroudcastReceiver* je tak možné obnovit seznam objektů podle aktuálních dat.

5.4.2. Synchronizace deníku

Synchronizace deníku probíhá podle algoritmu popsaném v 3.5.

Vytváření a editace záznamů

Jeden záznam je reprezentován jedním řádkem v tabulce *diary* lokální databáze. Aby mohly být uloženy informace, které uživatel očekává a aby byla zajištěna synchronizace, musí každý záznam v tabulce obsahovat tyto atributy:

id – jedinečný identifikátor v rámci lokální tabulky

guid – globální identifikátor, je vytvářen až při synchronizaci

from – timestamp zahájení pozorování

to – timestamp ukončení pozorování

latitude – zeměpisná šířka

longitude – zeměpisná délka

log – záznam pozorování

weather – záznam o počasí

row_counter – údaj reprezentující globální verzi objektu

timestamp – čas poslední modifikace

deleted – 1 pokud byl záznam smazán (soft delete)

sync_ok – 1 pokud byl objekt synchronizován

Záznamy jsou vytvářeny a upravovány v *DiaryEditActivity*. Pro jejich zadávání jsou použity standardní prvky frameworku Android.

Po otevření obrazovky s formulářem pro vkládání záznamu deníku, je automaticky doplněna zeměpisná šířka a délka. Pokud je k dispozici internetové spojení, je doplněn i záznam o aktuálním počasí. Podrobný způsob, jakým je počasí doplněno je popsán v kapitole 5.4.3.

Jeden záznam obsahuje čas začátku pozorování a čas konce pozorování. Při ukládání do databáze, musí systém zkontrolovat, zda jsou data validní – obsahují reálný časový interval (datum zahájení musí být nižší než datum ukončení pozorování). Pokud data nejsou validní, zobrazí se uživateli zpráva a záznam není uložen.

Stejně tak je kontrolována verze objektu. Při editaci existujícího záznamu, který má serverem přidělen *row_counter*, se načte jeho hodnota do paměti a zobrazí se formulář pro editaci záznamu. Pokud během editace dojde k synchronizaci a editovaný objekt je aktualizován novou verzí ze serveru, vznikne při ukládání konflikt verzí. Z toho důvodu je těsně před uložením objektu porovnána hodnota *row_counteru*, načtená do paměti při začátku editace, s aktuální hodnotou. Pokud se verze liší, došlo ke konfliktu, záznam nebude uložen a uživatel o tom bude informován.

Nově uložený záznam ještě nemá přiděleno *guid* a atribut *sync_ok* má hodnotu 0. To indikuje, že objekt ještě nebyl synchronizován se serverem. Záznam, který je pouze upraven má *guid* ale atribut *sync_ok* má také hodnotu 0. Ve výpisu záznamů tak může uživatel identifikovat stav objektu. Každý záznam ve výpisu má v pravém horním rohu ikonu označující stav synchronizace, červená barva značí že objekt ještě nebyl synchronizován, zelená znamená že objekt nebyl vytvořen ani editován od poslední synchronizace. Pod ikonou je *guid*. Na Obrázek 13: Výpis deníku je zobrazen výpis objektů. První záznam má červenou ikonu a nemá *guid.* Znamená to, že jde o nově vytvořený objekt. Druhý záznam má červenou ikonu, ale má i *guid*, což indikuje že objekt byl upraven a jeho změny ještě nebyly odeslány na server. Ostatní záznamy mají zelenou ikonu – jsou ve stejné verzi jako objekty na serveru.

Obrázek 13: Výpis deníku, zdroj: vlastní

Podobně jako v kapitole 5.4.1 je celá synchronizace prováděna v metodě *onPerformSync()* tentokrát však třídy *DiarySyncAdapter*. Třída obsahuje dvě privátní metody *syncFromServer()* a *syncToServer()*. První stáhne data ze serveru a uloží je do zařízení. Druhá má za úkol odeslat na server nová data. Pokud v první metodě dojde k chybě, nebude se provádět druhá. Pokud obě metody proběhnou v pořádku, označí se nové nebo aktualizované objekty v zařízení jako synchronizovaná, *sync_ok=1*. Pokud při problémech s odesíláním, dojde na serveru ke konfliktu, zůstane u aktualizovaných objektů *sync_ok=0*, systém se je pokusí odeslat při další synchronizaci.

Pokud dojde při synchronizaci ke konfliktu, ze serveru přijde nová verze objektu, který ještě nebyl synchronizován, je porovnán timestamp obou verzí objektu a použit ten aktuálnější.

5.4.3. Údaje o počasí

Údaje o počasí jsou získávány z veřejného API serveru OpenWeatherMap [31]. Tento web nabízí přístup k údajům o aktuálním počasí kdekoliv na Zemi. Při registraci je vygenerován API Key který umožní přistupovat k REST API. Klíč je spolu s ostatními parametry odeslán metodou GET:

```
http://api.openweathermap.org/data/2.5/weather?units=metric
       &appid=edcc91c97986f06f830222a5aabf9084&lat=49.798729&lon=16.1848778
```

Pokud uživatel zobrazí formulář pro vložení nového záznamu do deníku, je spuštěn *AsyncTask*, ve kterém jsou stažena aktuální data počasí. Volání API je zapouzdřeno v metodě *getWeatherData():*

```
public static JSONObject getWeatherData(double latitude, double longitude) throws
IOException, JSONException {

        String lat = "&lat=" + Double.toString(latitude);
        String lon = "&lon=" + Double.toString(longitude);
        String weatherUrl = AstroContract.weatherUri + "&appid="
             + AstroContract.weatherApiKey + lat + lon;
        Log.d("astro", "WEATHER> " + weatherUrl);
        URL url = new URL(weatherUrl);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        InputStream in = new BufferedInputStream(conn.getInputStream());
        String json = org.apache.commons.io.IOUtils.toString(in, "UTF-8");

        JSONObject jsonObject = new JSONObject(json);
        return jsonObject;
}
```

Z ukázky je patrné, že pro zjištění aktuálního počasí je nejdříve nutné znát aktuální polohu. Až po jejím úspěšném zjištění, je možno odeslat požadavek do API. Po úspěšném zjištění dat se doplní do příslušného políčka otevřeného formuláře.

5.4.4. Přihlášení pomocí Googlu

Až doposud popisovaná aplikace používá k přihlašování a autentizaci vlastní autorizační server, který je shodný s Resource serverem. Součástí této práce je i oblast přihlašování pomocí sociálních sítí. Tato kapitola popisuje způsob integrace přihlášení pomocí účtu Googlu.

Google disponuje velice rozsáhlým API, které zpřístupňuje nejrůznější služby třetím aplikacím. Pro potřebu přihlášení stačí pouze přístup k informacím o uživateli, k jeho emailu. Ten sám o sobě poslouží jako jedinečný identifikátor uživatele.

Google používá ve svém autorizačním systému kromě *access_tokenu* a *refresh_toknenu* ještě další dva kódy, které jsou důležité pro tuto aplikaci.

ID Token

ID Token je používán pro bezpečné přihlašování bez verifikace na Google autorizačním serveru. Je to řetězec znaků zakódovaný pomocí base64. Obsahuje JSON se základní informace o uživateli v závislosti na Scope použité při jeho vytvoření. Google poskytuje knihovny, které ho dokáží dešifrovat a použít ho pro identifikaci uživatele. Jeho platnost je omezená, podobně jako u access_tokenu [21]. Po dešifrování může vypadat například takto:


```
{
        "azp": "171814397882-qoafbodpid52h7lh0pc98bruc9vv16vs.apps.googleusercontent.com",
        "aud": "171814397882-qoafbodpid52h7lh0pc98bruc9vv16vs.apps.googleusercontent.com",
        "sub": "117546833415194912034",
        "email": "astro.test.2017@gmail.com",
        "email_verified": "true",
        "at_hash": "uwz1yPLOyl15RqaPB7DP6g",
        "iss": "https://accounts.google.com",
        "iat": "1490864098",
        "exp": "1490867698",
        "name": "Jan Macháček",
        "picture": "https://lh3.googleusercontent.com/-6WzVtOGv358/AAAAAAAAAAI/AAAAAAAAACU/kf6WS-jTII8/s96-c/photo.jpg",
        "given_name": "Jan",
        "family_name": "Macháček",
        "locale": "cs",
```

```
        "alg": "RS256",
        "kid": "d6eb2094ac2b7f5763dd34ca277a3450efbdb6a9"
}
```

Authorization code

Authorization code se používá pro získání *access_tokenu* a *refresh_tokenu*. Doba jeho užití je velice krátká. Pro zjištění tokenů se dá použít pouze jednou. Poté je zneplatněn.

Registrace aplikace do Google API

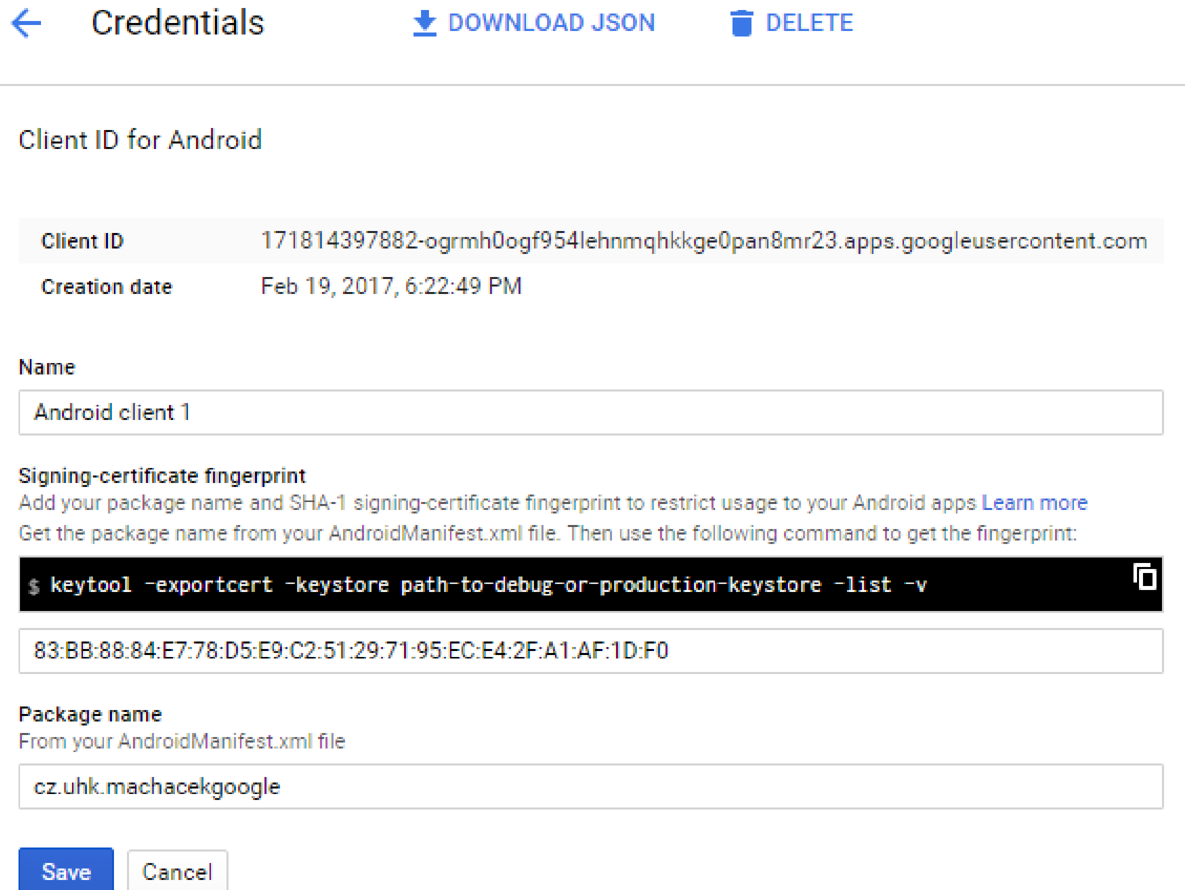
Aplikace, která má mít přístup do Google API musí mít přiděleno *Client ID a Client secret*. Obě tyto položky jsou dostupné po registraci aplikace do Google API console [32]. Zde je třeba vytvořit *Credentials pro OAuth client ID*. Google nabízí vytvoření *Client ID* pro několik typů aplikací. V tomto případě se jedná o aplikaci typu Android. Je třeba zadat ještě namespace aplikace – atribut „package“ v Manifestu. K dokončení je dále nutno vygenerovat SH1 otisk, sloužící pro jedinečnou identifikaci aplikace. Otisk se vygeneruje v konzoli příkazem:

```
keytool -exportcert -keystore path-to-debug-or-production-keystore -list -v
```

jeho podoba může být například takováto:

```
83:BB:88:84:E7:78:D5:E9:C2:51:29:71:95:EC:E4:2F:A1:AF:1D:F0
```

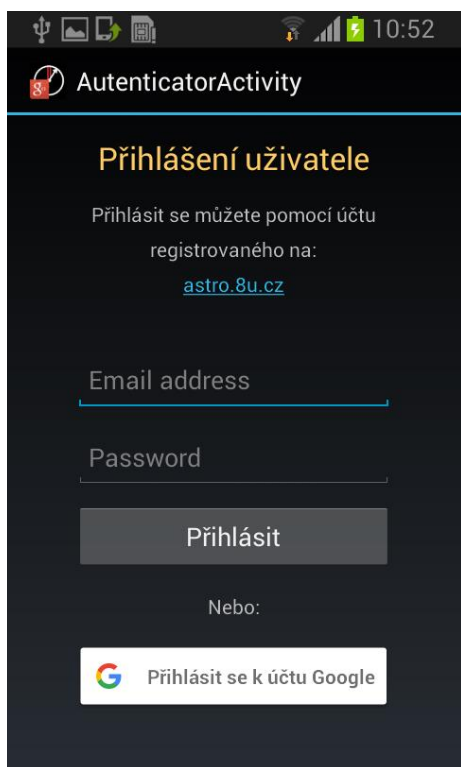
Po uložení formuláře je možné stáhnout soubor .json obsahující *Client ID, Client secret* a další potřebné údaje sloužící k identifikaci aplikace.



Obrázek 14: registrace aplikace do Google API manageru, zdroj: vlastní

Přihlášení uživatele

Téměř každé zařízení se systémem Android je přihlášeno k účtu Google. Tuto vlastnost lze využít v aplikaci, uživatel jen potvrdí, že se chce přihlásit pomocí svého účtu, nemusí zadávat přihlašovací údaje a aplikace je nebude nikam odesílat. Do přihlašovacího formuláře je přidáno tlačítko, které spustí proces autentizace pomocí Google účtu.



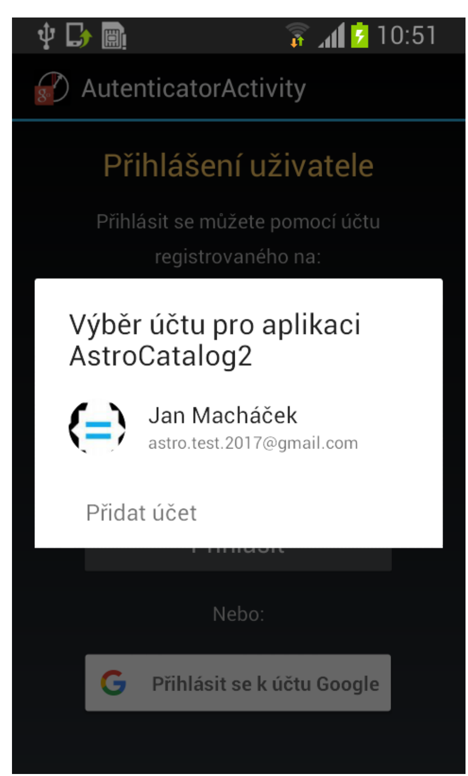
Obrázek 15: Přihlašovací formulář s Google tlačítkem, zdroj: vlastní

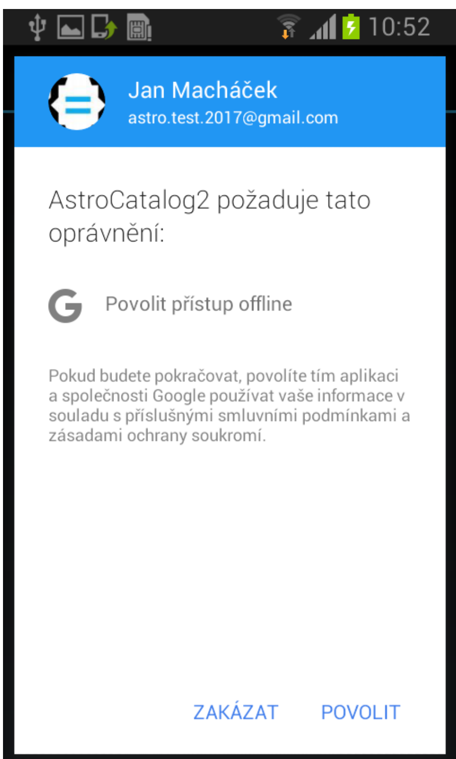
K přihlášení aplikace k účtu lze použít třídy *GoogleSignInOptions* a *GoogleApiClient*, které jsou součástí knihovny play-services:

```
GoogleSignInOptions gso = new GoogleSignInOptions.Builder(
             GoogleSignInOptions.DEFAULT_SIGN_IN)
      // ziskani údajů uživatelova účtu
      .requestScopes(new Scope(Scopes.PLUS_ME))
      //nastavení api_client_id a požadavku na získání autorizačního kódu pro
      offline přístup
      .requestServerAuthCode(AstroContract.API_CLIENT_ID, true)
      .requestEmail()
      .build();

// Vytvoření GoogleApiClient s přístupem do Google Sign-In API
mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addApi(Auth.GOOGLE_SIGN_IN_API, gso)
        .build();
```

Ukázka kódu 1: Vytvoření GoogleApiClient

GoogleApiClient je vytvořen pomocí parametrů definovaných v objektu *GoogleSignInOptions*. Metoda *requesetScopes()* určuje Scope kam má mít uživatel přístup, v tomto případě jsou to základní informace o uživatelově účtu. První parametr metody *requestServerAuthCode()* slouží pro zadání *Client ID*, druhý parametr slouží pro určení, zda bude API vracet i *refresh_token*, který je v tomto případě velice důležitý.





Obrázek 16: přihlášení pomocí účtu Googlu, zdroj: vlastní

Po kliknutí na přihlašovací tlačítko, je uživatel vyzván k výběru účtu, ke kterému se chce přihlásit. Poté musí udělit aplikaci právo k offline přístupu k jeho účtu. Offline přístup je velice důležitý. Znamená to, že aplikace potom bude moci pomocí *refresh_tokenu* sama aktualizovat *acces_token* bez přítomnosti uživatele, synchronizace tak bude probíhat i v době kdy je zařízení na kterém aplikace běží neaktivní.

Pokud vše proběhne bez problémů, je vytvořen objekt třídy *GoogleSignInAccount*. Reprezentuje účet právě přihlášeného uživatele. Lze použít pro zjištění informací o uživateli, obsahuje například jméno uživatele, email, obrázek spojený s účtem atd., dle použité Scope. Obsahuje i ID token, který se dá použít pro přihlášení do API. Pokud by aplikace přistupovala k API pouze při aktivitě uživatele, nebylo by třeba zjišťovat žádné další údaje, pro identifikaci by se mohl požít ID token.

Tato aplikace však synchronizuje data na pozadí, často i při neaktivitě uživatele. Pokud by se používal pouze ID token, došlo by po uplynutí životnosti k jeho zneplatnění. Nový ID token lze tímto způsobem získat pouze při přítomnosti uživatele (!). Pro zajištění synchronizace je třeba obstarat *refresh_token*, který lze využít k obnovení ID tokenu.

Získání refresh_tokenu

Refresh_token je nezbytně nutný pro přístup do API v offline módu. Synchronizace tak může probíhat i bez přítomnosti uživatele. V předchozí kapitole je popsán postup získání instance třídy *GoogleSignInAccount*. Metodou *getServerAuthCode()* lze získat *Authorization code* a ten pak použít pro zjištění *refresh_tokenu*.

Třída *GoogleAuthorizationCodeTokenRequest* z knihovny api-client, zajistí veškerou funkcionalitu.

```
GoogleTokenResponse googleTokenResponse = new GoogleAuthorizationCodeTokenRequest(
        new NetHttpTransport(),
        JacksonFactory.getDefaultInstance(),
        "https://www.googleapis.com/oauth2/v4/token",
        "171814397882-qoafbodpid52h71h0pc98bruc9vv16vs.apps.googleusercontent.com",
        "zN_3GYmPfnSAnlz0ChErRI4M",
        "4/T2HdYFfsAMUR52FoTg_X_a5XKEM7SYIC0YUKDKdJNnM",
        null).execute();
```

Do konstruktoru objektu je třeba zadat URL API, Client ID, Client secret a Authorization code. Výsledkem je objekt třídy *GoogleTokenResponse* obsahující *access_token*, *refresh_token* a *ID token*. Metoda *execute()* v podstatě zapouzdřuje volání API:

```
POST https://www.googleapis.com/oauth2/v4/token

        code:4/T2HdYFfsAMUR52FoTg_X_a5XKEM7SYIC0YUKDKdJNnM
        client_id:171814397882-qoafbodpid52h71h0pc98bruc9vv16vs.apps.googleusercon-
        tent.com
        client_secret:zN_3GYmPfnSAnlz0ChErRI4M
        redirect_uri:
        grant_type:authorization_code
```

Odpovědí serveru je:


```
{
  "access_token":    "ya29.GlseBHRki-dHg2_6wm6YS_fRlJk31alp6TEeNMboBsLuA_vKE9XVzzC--
gDmkOWsIBnfI78I0KS9xq7-jx0r2k-258g8iD97t-W5NNsrOn_whEEEaSuZwSc4Zjas",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "1/9oCEBxrKifgBO0Vy_T7RQeOQn31ozA3SyqBBHU1C-5M",
  "id_token":                                                "eyJhbGciOiJSUzI1NiIsImt-
pZCI6ImQ2ZWIyMDk0YWMyYjdmNTc2M2RkMzRjYTI3N2EzNDUwZWZiZGI2YTkifQ.eyJhenAiOiIx-
NzE4MTQzOTc4ODItcW9hZmJvZHBpZDUyaDdsaDBwYzk4YnJ1Yzl2djE2dnMuY …"
}
```


Tento JSON response je převeden na objekt *GoogleTokenResponse*.

Všechny získané tokeny je nutné uložit do *AccountManageru* pod příslušným účtem. Pokud při synchronizaci dojde k expiraci ID tokenu, lze použít získaný *access_token* k získání nového ID tokenu.

5.4.5. Implementace na straně serveru

Server, na kterém je umístěno REST API pro přístup k datům uživatele, slouží zároveň jako autorizační server. Při implementaci přihlášení pomocí Google účtu je třeba na to odpovídajícím způsobem reagovat. Server tak musí být schopen autorizovat i požadavky od uživatelů kteří použijí k přihlášení svůj účet u Googlu. V takových případech je možno k ověření použít autorizační server Googlu. V tomto případě se ověřuje pouze platnost emailové adresy, není třeba získávat z API Googlu žádné další informace. Uživatele lze tak ověřit i na základě ID tokenu. Příchozí ID token je tak možno autorizovat přímo na severu kam byl odeslán, není tak potřeba odesílat další požadavek na vzdálený server. Google nabízí PHP knihovnu, která poskytuje požadovanou funkcionalitu [22].

```
$client = new \Google_Client(['client_id' => $this->clientId]);
if ($data = $client->verifyIdToken($accessToken)) {
      if (!$userId
            = $this->userRepository->getUserIdByLogin($data["email"], 2)){
                  $userId = $this->userRepository->addUser(
                        $data["given_name"],
                        $data["family_name"],
                        $data['email'],
                        "",
                        UserRepository::CLIENT_GOOGLE);
      }
}
```

Z kódu je patrná hlavní část celého procesu. Třída *Google_Client* se použije k dekódování ID tokenu, pokud je ID token validní, vrátí třída pole parametrů, email, jméno, příjmení apod. Server ověří, zda se v databázi nachází uživatel se stejným emailem, pokud ne, je vytvořen.

Takto je možné zajistit celý autorizační proces na Resource serveru. Uživatel nemusí svůj účet nikde registrovat, je vytvořen automaticky s prvním odeslaným validním ID tokenem.

6. Webová aplikace

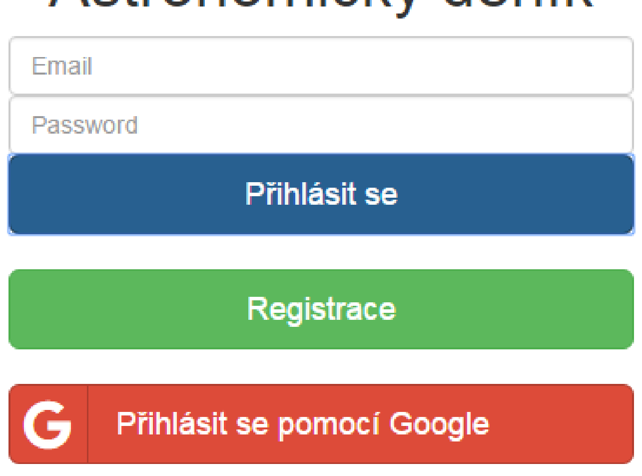
6.1. Popis a implementace

Webová aplikace pro přístup k datům je vytvořena v PHP pomocí frameworku Nette [23]. K formátování je použit framework Bootstrap [24], který zajistí aplikaci potřebnou responsibilitu.

Úkolem aplikace je umožnit uživateli vytvořit na serveru účet, pro přístup k AstroApi, na které je napojena aplikace pro Android. Uživatel má po přihlášení přístup k datům vytvořeným v mobilní aplikaci. Lze si prohlížet uložená data, upravovat je, ale také vytvářet nové záznamy. Všechny data jsou plně synchronizovaná a maximálně aktuální. Pro přihlášení a registraci jsou použity standartní nástroje frameworku Nette.

6.2. Přihlášení pomocí Googlu

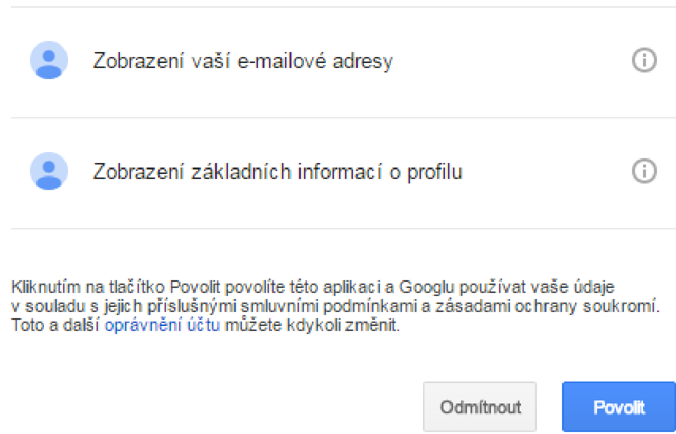
Aplikace pro Android umožňuje uživateli použít pro přihlášení svůj účet, pod kterým je zařízení registrováno. Stejným způsobem lze používat i webovou aplikaci. Autorizace bude probíhat tak jak je popsáno v Authorization Code Grant. Webová aplikace je v tomto případě v roli klienta.



Obrázek 17: Přihlašovací formulář, zdroj: vlastní

Pokud se chce uživatel přihlásit pomocí svého Google účtu, klikne na tlačítko a je přesměrován na *GooglesignPresenter*. Zde je automaticky přesměrován na stránku Googlu s výběrem jeho účtu. Při prvním přihlášení je nutné přidělit aplikaci oprávnění pro přístup k jeho informacím.

Aplikace AstroApi požaduje následující oprávnění:



Obrázek 18: Potvrzení oprávnění, zdroj: vlastní

Poté je přesměrován zpět na *GooglesignPresenter*, spolu s informacemi o uživateli. Zde jsou informace o uživateli ve formě *access_tokenu* předány do třídy *GoogleUserManager*, kde je uživatel korektně přihlášen do Nette aplikace.

Astronomický deník

Nový záznam

Jan Macháček (astro.test.2017@gmail.com) odhlásit

guid	Datum	Délka	Log	Počasí	Location
9dfaf190-486f-4571-aa2a-797097af1b78	28.03.2017 16:12	2h 02:00	Pozorování sluneční korony a slunečních skvrn. Přítomna pouze tři méně zřetelné skvrny.	Clear, clear sky, 20°C, humidity: 28%, pressure: 1020hPa	49.7984055 16.1915768
8e65f5b9-8fe6-4708-a4da-ff4f4718324a	23.03.2017 23:57	1h 01:00	Pozorování Měsíce v první čtvrti. Planeta Jupiter. Pozorování přechodů Galileových měsíců přeš kotouč Jupitera.	Mist, mist, 7°C, humidity: 87%, pressure: 1026hPa	50.0238371306 15.7206442739
65e32209-47e9-45f2-8fb0-f138c9abc0ee	18.03.2017 20:45	1h 01:00	Spirální galaxie v Andromedě M31, Galaxie M32 v trojúhelníku.	Clouds, scattered clouds, 11°C, humidity: 53%, pressure: 1028hPa	50.2047934 15.8297738

Obrázek 19: Výpis objektů ve webovém rozhraní, zdroj: vlastní

7. Vývoj a testování

Pro vývoj systému, který používá pro předávání zpráv rozhraní REST API je téměř nutné použít vhodné testovací nástroje.

Pro vytváření a odesílání HTTP požadavků je vhodné použít aplikaci Postman [25]. Zde lze vytvářet zprávy, odesílané na API. Při vývoji prostřednictvím tohoto nástroje byly vytvořeny všechny druhy zpráv, odesílané klientem, a bylo otestováno, zda odpovědi splňují očekávání. Velice to usnadní vývoj a testování OAuth i synchronizace.

Klienta lze vytvářet bez existence API. K tomu lze použít nástroj Apiary [26]. Ten umožní vytvořit kompletní REST API testovací rozhraní. Vytvoří se zde metody API, vracející testovací data. Klient se tak může na testovací API připojit a komunikovat tak, jak to bude prováděno se skutečným API.

Snad nejdůležitějším nástrojem pro vývoj je AndroidStudio a jeho AndroidMonitor. Pomocí logů umístěných v aplikaci lze zobrazovat v reálném čase zprávy informující o stavech aplikace. Tento nástroj je nezbytný pro vývoj takto složité aplikace.

Součástí aplikace je i napojení na nástroj Firebase [27]. Jde o knihovnu poskytovanou Googlem, která umožní sledovat a odesílat chyby vzniklé při běhu nainstalovaných aplikací.

Základní bezpečnost API a webové aplikace byla otestována nástrojem OWASP Zed Attack Proxy [28].

Algoritmus synchronizace byl otestován postupem popsaným v kapitole 3.5.43.5.4. K tomu byly použity dvě zařízení (telefon a tablet), připojená na jeden účet.

Aplikace byla poskytnuta několika uživatelům k testování. Oceňována byla zejména jednoduchost přihlašování pomocí Google účtu a možnost práce v offline režimu.

8. Závěr a doporučení

Tato práce přiblížila problematiku synchronizace dat mobilních zařízení. Podařilo se vytvořit systém, který splnil všechny stanovené cíle.

Jsou zde popsány komponenty, pomocí kterých lze zajistit synchronizaci dat mezi více připojenými klienty. Je použito minimální množství knihoven a hotových řešení třetích stran. Hlavní funkce byly naprogramovány pro potřeby této práce. To umožní dopodrobna pochopit fungování celého systému. Serverová část a mobilní aplikace mezi sebou komunikují prostřednictvím REST API. Rozhraní je tak snadno použitelné a testovatelné. Je vytvořena vlastí implementace OAuth, umožňující bezpečnou komunikaci přes REST API rozhraní. Součástí práce je i ověření uživatele pomocí účtu Google. Při zpracování bylo třeba nastudovat a pochopit systém, který Google pro ověřování uživatelů používá.

Aplikace pro Android umožní uživateli používat aplikaci i v místech s nedostupným internetovým připojením. Data jsou synchronizována se serverem a propagována na ostatní připojená zatřízení. Pro správu účtu, přihlášení a synchronizaci byly využity části Android frameworku k tomuto účelu určené a doporučené. To umožní maximálně úspornou a efektivní funkci aplikace. Záznamy vytvářené v telefonu jsou automaticky doplňovány informacemi o aktuální poloze s využitím senzorů přístroje. Pokud je dostupné internetové připojení, jsou doplněny i informace o aktuálním počasí pomocí API třetí strany.

Serverová část je vytvořena pomocí jazyka PHP a databáze MySql. API přijímá a odpovídá na požadavky klientů. Je schopná provádět autorizaci požadavků dle OAuth standartu. Zároveň umožní autorizovat požadavky klientů z účtů Google. Slouží jako primární úložiště dat vytvořených v aplikaci pro Android.

Webová aplikace běžící na serveru umožní uživateli registraci nového účtu i přihlášení pomocí jeho Google účtu. Slouží pro zobrazení a editaci dat vytvořených mobilní aplikací.

V práci jsou řešena zejména témata související se synchronizací dat. Jsou zde popsány hlavně procesy běžící na pozadí. Aplikace, která je výsledkem této práce splňuje všechny požadavky na ní kladené. Není však vhodná pro publikaci prostřednictvím Google Play. Slouží pouze jako referenční projekt, na kterém jsou demonstrovány postupy související se stanoveným cílem práce.

Aplikace, která by byla vhodná pro skutečný provoz, by v současné době měla být výsledkem týmové práce. Na vizuální stránce se musí podílet designér, je třeba stanovit vhodnou ergonomii, aplikaci je třeba lokalizovat atd. Z důvodu větší bezpečnosti by komunikace s REST API měla probíhat pomocí protokolu HTTPS. Dále je třeba zajistit fungování serveru na kterém je umístěno API.

Tento projekt využívá volně dostupnou doménu a testovací hosting. Testování hotové aplikace by mělo probíhat na pokud možno největším počtu zařízení různých výrobců. Zajištění těchto a dalších požadavků je ale nad rámec této práce.

Možnou modifikací aplikace je napojení na některou službu založenou na mBaaS. Tím by byla odstraněna nutnost provozovat server. Další možností je doplnění katalogu objektů o další moduly, například o katalog NGC apod. Uživatel by si tak mohl sám stanovit, které katalogy chce

mít v zařízení staženy. Mobilní telefon disponuje dostatečným výkonem pro výpočet efemerid – souřadnic na kterých se nacházejí planety a ostatní objekty sluneční soustavy, zobrazení grafu pozice Jupiterových měsíců atd. Aplikace by se tak stala komplexním nástrojem amatérského astronoma.

9. Seznam použitých zdrojů

[1] *Mobile application platform, MBaaS* [online]. [cit. 2017-03-22]. Dostupné z: https://www.redhat.com/en/technologies/mobile/application-platform

[2] *ParsePlatform/parse-server: Parse-compatible API server module for Node/Express* [online]. [cit. 2017-03-22]. Dostupné z: https://github.com/ParsePlatform/parse-server

[3] *Firebase | App success made simple* [online]. [cit. 2017-03-22]. Dostupné z: https://firebase.google.com

[4] *AppSync.org: open-source patterns & code for data synchronization in mobile apps* [online]. [cit. 2017-03-19]. Dostupné z: http://confluence.tapcrowd.com/pages/viewpage.action?pageId=2262404

[5] *SOAP Version 1.2 Part 0: Primer (Second Edition)* [online]. [cit. 2017-03-27]. Dostupné z: https://www.w3.org/TR/soap12-part0/

[6] *Is REST losing its flair - REST API Alternatives* [online]. [cit. 2017-03-27]. Dostupné z: https://www.programmableweb.com/news/rest-losing-its-flair-rest-api-alternatives/analysis/2013/12/19

[7] RICHARDSON, Leonard a Michael AMUNDSEN. *RESTful Web APIs. O'Reilly Media*, 2013. ISBN 978-1-449-35806-8.

[8] *OAuth Core 1.0* [online]. [cit. 2017-03-27]. Dostupné z: https://oauth.net/core/1.0/

[9] *Understanding OAuth2* [online]. [cit. 2017-03-28]. Dostupné z: http://www.bubblecode.net/2016/01/22/understanding-oauth2/

[10] *Drahak/Restful: Drahak\Restful - Nette REST API bundle* [online]. [cit. 2017-03-28]. Dostupné z: https://github.com/drahak/Restful

[11] *RFC6749* [online]. [cit. 2016-08-19]. Dostupné z: https://tools.ietf.org/html/rfc6749

[12] *RFC3986* [online]. [cit. 2016-08-19]. Dostupné z: https://tools.ietf.org/html/rfc3986

[13] *HttpURLConnection | Android Developers* [online]. [cit. 2016-08-20]. Dostupné z: https://developer.android.com/reference/java/net/HttpURLConnection.html

[14] *Handler | Android Developers* [online]. [cit. 2017-04-19]. Dostupné z: https://developer.android.com/reference/android/os/Handler.html#postDelayed(java.lang.Runnable, long)

[15] *AccountAuthenticatorActivity* [online]. [cit. 2017-03-08]. Dostupné z: https://developer.android.com/reference/android/accounts/AccountAuthenticatorActivity.html

[16] LUCAS JORDAN, Pieter Greyling a Tony Hillerson. *TECHNICAL REVIEWER. Practical Android projects*. Norwood Mass: Books24x7.com, 2011. ISBN 978-143-0232-445.

[17] *RFC 6749 Client Credentials Grant* [online]. [cit. 2017-03-10]. Dostupné z: https://tools.ietf.org/html/rfc6749#section-4.4

[18] *Authenticating to OAuth2 Services* [online]. [cit. 2017-04-01]. Dostupné z: https://developer.android.com/training/id-auth/authenticate.html

[19] *AccountManager* [online]. [cit. 2017-02-14]. Dostupné z: https://developer.android.com/reference/android/accounts/AccountManager.html

[20] MEDNIEKS, Zigurd, G. BLAKE MEIKE, Laird DORNIN a Zane PAN. *Enterprise Android: programming Android database applications for the enterprise*. Indianapolis, IN: Wrox, a Wiley brand, 2014. ISBN 9781118240465.

[21] *ID Token Authentication| API Client Library for PHP* [online]. [cit. 2017-03-30]. Dostupné z: https://developers.google.com/api-client-library/php/guide/aaa_idtoken

[22] *A PHP client library for accessing Google APIs* [online]. [cit. 2017-03-30]. Dostupné z: https://github.com/google/google-api-php-client

[23] *Rychlý a pohodlný vývoj webových aplikací v PHP | Nette Framework* [online]. [cit. 2017-03-30]. Dostupné z: https://nette.org/cs/

[24] *Bootstrap* [online]. [cit. 2017-03-30]. Dostupné z: http://getbootstrap.com/2.3.2/

[25] *Postman | Supercharge your API workflow* [online]. [cit. 2017-04-01]. Dostupné z: https://www.getpostman.com/

[26] *Apiary | Platform for API Design, Development & Documentation* [online]. [cit. 2017-04-01]. Dostupné z: https://apiary.io/

[27] *Firebase | App success made simple* [online]. [cit. 2017-04-01]. Dostupné z: https://firebase.google.com/

[28] *OWASP Zed Attack Proxy Project - OWASP* [online]. [cit. 2017-04-01]. Dostupné z: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[29] *RFC5988* [online]. [cit. 2016-08-19]. Dostupné z: https://tools.ietf.org/html/rfc5988

[30] *Running a Sync Adapter | Android Developers* [online]. [cit. 2017-03-15]. Dostupné z: https://developer.android.com/training/sync-adapters/running-sync-adapter.html

[31] *Weather API- OpenWeatherMap* [online]. [cit. 2017-03-29]. Dostupné z: http://openweathermap.org/api

[32] *Google APIs* [online]. [cit. 2017-04-01]. Dostupné z: https://console.developers.google.com

Příloha 1: Obsah přiloženého CD

Součástí práce je datové CD obsahující zdrojové kódy ukázkového projektu a zkompilovanou aplikaci, kterou lze přímo nainstalovat do přístroje.

CD obsahuje čtyři adresáře:

- Aplikace
- Android
- PHP
- SQL

Aplikace obsahuje zkompilovanou aplikaci AstroCatalog2. Aplikace je určena pro Android 7.1.2 Nougat, minimální verze operačního systému pro běh aplikace je Android 4.3 Jelly Bean.

Android adresář se zdrojovým kódem aplikace AstroCatalog2.

PHP zdrojový kód serveru a webové aplikace určený pro PHP 5.5.

SQL soubor pro vytvoření struktury tabulek a naplnění nutnými daty pro databázi MySQL.

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2016/2017

Studijní program: Aplikovaná informatika
Forma: Kombinovaná
Obor/komb.: Aplikovaná informatika (ai2-k)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Macháček Jan	Kolonie 448, Pardubice - Svítkov	I14296

TÉMA ČESKY:

Synchronizace dat aplikací pro mobilní zařízení

TÉMA ANGLICKY:

Data synchronization in mobile applications

VEDOUCÍ PRÁCE:

Ing. Pavel Kříž, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl práce: Nastudovat a popsat problematiku synchronizace dat aplikací pro mobilní zařízení. Demonstrovat dosažené znalosti na aplikaci pro OS Android, webovém klientu a serveru zajišťujícím synchrnonizaci. Implementovat přihlášení pomocí sociální sítě.

Osnova práce:
1. Úvod
2. Synchronizace dat mobilních zařízení
3. Návrh API a serveru
4. Implementace v prostředí Android
5. Webový klient
6. Přihlášení pomocí sociální sítě
7. Závěr a doporučení

SEZNAM DOPORUČENÉ LITERATURY:

1) https://developer.android.com/index.html
2) https://tools.ietf.org/html/rfc6749
2) Reto Meier: Professional Android 4th Edition, 2017, ISBN 978-1118949528
3) RICHARDSON, Leonard a Michael AMUNDSEN. RESTful Web APIs. O'Reilly Media, 2013. ISBN 978-1-449-35806-8.
4) LUCAS JORDAN, Pieter Greyling a Tony Hillerson. TECHNICAL REVIEWER. Practical Android projects. ISBN 978-143-0232-445
5) Android Application Development Cookbook - Second Edition, 2016, ISBN 1785886193
6) MEIKE, G. Blake. Android concurrency. Android deep dive series. ISBN 9780134177434

Podpis studenta: .... **Datum:** ....

Podpis vedoucího práce: .... **Datum:** ....

(c) IS/STAG , Portál - Podklad kvalifikační práce , I14296 , 13.04.2017 00:14