VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

KONTEJNER PRO MIGRUJÍCÍ SOFTWAROVÉ KOMPONENTY BEŽÍCÍ NA OS ANDROID

A CONTAINER FOR MIGRATATING SOFTWARE COMPONENTS RUNNING ON ANDROID OS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. VLADIMÍR ŠČEŠŇÁK**
AUTHOR

VEDOUCÍ PRÁCE **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2018

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Kľúčové slová

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Citácia

ŠČEŠŇÁK, Vladimír. *Kontejner pro migrující softwarové komponenty bežící na OS Android*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Kontejner pro migrující softwarové komponenty bežící na OS Android

Prehlásenie

Prehasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením RNDr. Mareka Rychlého, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Vladimír Ščešňák
22. mája 2018

Poďakovanie

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

Obsah

Kapitola 1

Úvod

[TODO: Úvod, cieľ práce, teória, návrh...]

[TODO: kam by som mal dať reflexiu(?) - použitá technológia(?)]

Kapitola 2

Operačný systém Android

Táto kapitola sa venuje operačnému systému Android, ktorý v roku 2007 uverejnilo konzorcium Open Handset Alliance, pozostavajúce z technologických spoločností ako je Google. Android bol uvedený pod open-source licenciou `Apache/MIT` a spolu s ním uzrelo svetlo sveta aj prvá beta verzia Android Software Development Kit (v skratke SDK). Ako uvádza [8] v priebehu pár mesiacov si stiahlo prvú beta verziu zo stránky Google niekoľko miliónov ľudí a v roku 2008 bol v Spojených štátoch amerických predstavený prvý mobilný telefón bežiaci na tejto platforme.

Cieľom uvedenia tohto operačného systému bolo zjednodušenie vývoja a predávania softvérových aplikáci bežiacich na mobilných telefónoch a tiež zaviesť štandard podobný PC a Macintoshu, ktorý by bol primárne určený pre mobilné telefóny. V súčasnosti sa ale táto platforma rozšírla aj na mikropočítače, inteligentné televízory (známe ako Android TV), inteligentné hodinky (Android Wear), palubné systémy áut a ďalšie druhy elektroniky. V čase písania tejto práce je operačný systém Android dostupný vo verzii 8.0 Oreo. Informácie uvedené v tejto kapitole boli čerpané zo zdroja [1]

2.1 Vrstvy operačného systému Android

Ako môžeme vidieť na obrázku 2.1 samotný Android je založený na **linuxovom jadre**, čo je aj jeho prvá vrstva. To umožňuje využiť kľúčové funkcie zabezpečenia a tiež umožňuje výrobcom zariadení ľahko vyvinúť ovládače hardvéru.

Ďalšou vrstvou je **hardvérová abstrakčná vrstva**, ktorá pozostáva z viacerých modulov a poskytuje štandardné rozhrania, ktoré vystavujú schopnosti hardvéru zariadenia do vyššej úrovne aplikačného rámca Java API. Úlohou vrstvy, ktorá je označená ako **Android Runtime** je, že každá aplikácia beží vo svojom vlastnom procese a má vlastnú inštanciu **Android Runtime**. Tá vykonáva beh viacerých virtuálnych strojov s nízkou pamäťou pomocou `DEX` súborov, čo je formát bytekódu špeciálne navrhnutého pre Android.

Mnoho základných komponentov a služieb systému Android, ako už aj vyššie spomenuté, vyžadujú natívne knižnice napísané v jazyku `C` a `C++` a samotný Android poskytuje aplikačné rozhranie, pre jednoduchší prístup z vyvíjaných aplikácii.

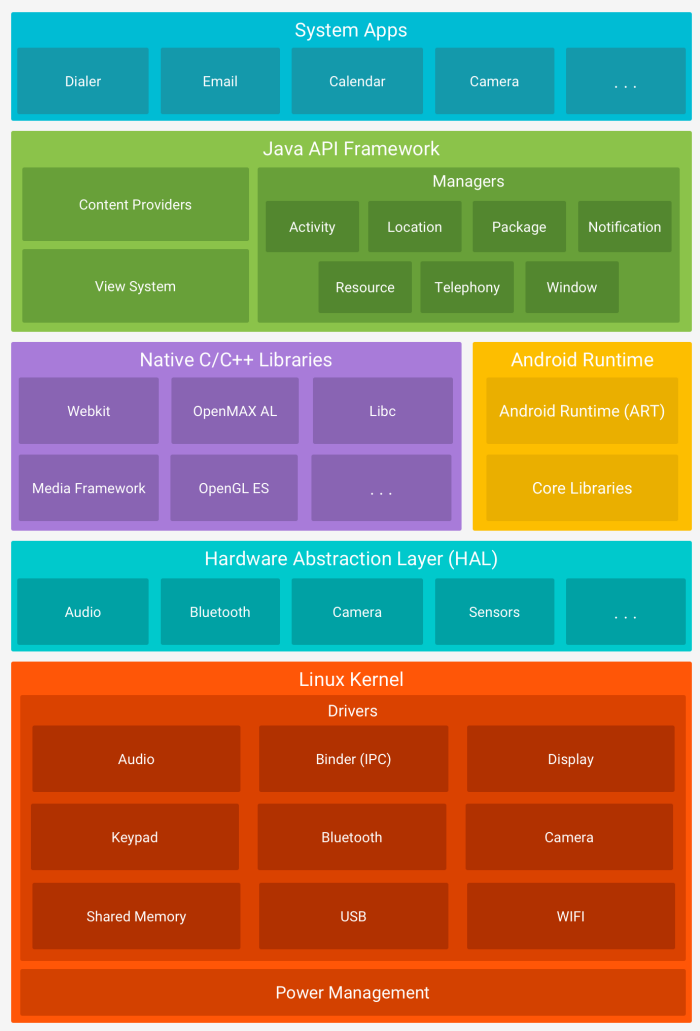
Java API Framework obsahuje základné stavebné bloky, písané v jazyku Java, ktoré potrebuje programátor pri vytváraní aplikácii. Existujú štyri rôzne typy komponentov aplikácie:

- Aktivity

- Služby
- Prijímače
- Poskytovatelia obsahu

Hlavne vďaka týmto blokom, môžeme ľahko vytvárať prívetivé používateľské rozhranie, pristupovať k zdrojom ako napríklad grafika, či iné súbory, alebo spravovať samotné aktivity. Túto vrstvu si popíšeme detailnejšie v ďalších častiach.

Poslednou vrstvou sú **Systémové aplikácie**, kde ide o množinu základných aplikácii, ktoré sú dodávané s každým Androidom. Ide hlavne o aplikácie ako telefonovanie, vytváranie textových správ, kamera, kalendár a podobne.



Obr. 2.1: Platforma Android

2.2 Aktivity a ich životný cyklus

Aktivity sú základným stavebným blokom aplikácii postavených na platforme Android. Sú akýmsi vstupným bodom medzi užívateľovom a aplikáciou. Samotná aktivita poskytuje

okno, kde aplikácia vykresľuje svoje užívateľské rozhranie, ktoré môže vyplniť celú obrazovku, môže byť menšie ako celá obrazovka, alebo sa dokonca môže objaviť v popredí iného okna.

Trieda `Activity` je na rozdiel od iných programovacích paradigiem, kde sú aplikácie spúšťané volaním metódy `main()`, iniciované systémom Android v inštancii `Activity` vyvolaním špecifických metód, ktoré odpovedajú konkrétnym fázam svojho životného cyklu. Pri interakcii užívateľa prechádzajú jednotlivé inštancie triedy `Activity` rôznymi stavmi. Týchto stavov je šesť a konkrétne sú to stavy *Created*, *Started*, *Resumed*, *Paused*, *Stopped*, *Destroyed*. Trieda `Activity` nám poskytuje niekoľko spätných volaní, ktoré dovoľujú aktivite vedieť, že sa nejako zmenil ich stav na základe ktorého, systém vytvára, zastavuje, obnovuje alebo ničí proces, v ktorom je daná aktivita umiestnená. Fázy životného cyklu a spätné volania, ktoré ich indikujú, si popíšeme nižšie a je ich možné tiež vyjadriť aj orientovaným grafom na obrázku 2.2.

2.2.1 Životný cyklus aktivity a jej spätné volania

- `onCreate()` - volanie, ktoré musí byť implementované. Toto volanie je spustené pri prvom vytvorení danej aktivity. V tomto bode vstupuje aktivita do stavu *Created*. V tejto metóde by mala byť implementovaná základná logika, ktorá sa vykoná len raz počas celej životnosti aktivity. Metóda obsahuje parameter `savedInstanceState` typu `Bundle`, obsahujúci predtým uložený stav aktivity. Ak je aktivita vytváraná prvý krát, hodnota tohto parametru je `null`. Aktivita ale hneď po spustení tejto metódy prejde do stavu *Started* a systém rýchlo za sebou zavolá metódy `onStart()` a `onResume()`.

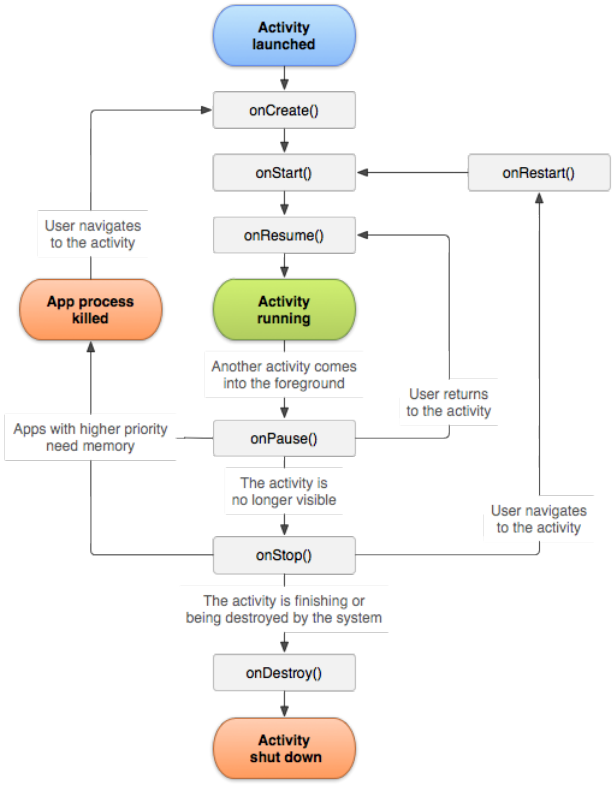
- `onStart()` - ak sa aktivita už nachádza v stave *Started*, systém vyvoláva toto spätné volanie. Metóda má na starosti to, že robí aktivitu viditeľnú pre používateľa a pripravuje ju na možnú interakciu. Na tomto mieste by sa mal inicializovať kód, ktorý udržuje používateľské rozhranie, alebo registrovať príjmač `BroadcastReceiver`, ktorý sleduje zmeny, na základe ktorých sa potom mení používateľské rozhranie. Po vykonaní tejto metódy aktivita prejde do ďalšieho stavu *Resumed* a volá metódu `onResume()`.

- `onResume()` - ak aktivita prejde do stavu *Resumed*, čo môže nastať buď prechodom zo stavu *Started*, alebo *Paused*, je vyvolaná táto metóda. V tomto stave aplikácia komunikuje s používateľom a reaguje na jeho interakciu, čo môže byť vybranie rôznych prvkov aplikácie, klepnutím, či iným povoleným gestom na displeji hardvéru, na ktorom tento operačný systém beží. V tejto metóde by sa mali inicializovať komponenty, ktoré neskôr v metóde `onPause()` uvoľníme. Aktivita a s ňou celá aplikácia ostáva v tomto stave, až kým nenastane udalosť podobná prejdeniu užívateľa do inej aktivity, prijatia hovoru, či vypnutie obrazovky zariadenia, kedy daná aktivita stráca pozornosť. Ak dôjde k tejto udalosti, aktivita prejde do stavu *Paused* a vyvolá sa metóda `onPause()`.

- `onPause()` - systém zavolá túto metódu ako prvú indikáciu toho, že používateľ opúšťa danú aktivitu. Nie vždy to znamená, že užívateľ aplikáciu zruší a musí dôjsť k volaniu metódy `onDestroy()`. V tomto momente sa aktivita nachádza v stave *Paused* a v tejto metóde by mali byť implementované veci ako zastavenie rôznych animácii, prehrávanie médií, alebo uvoľnenie rôznych komponent. Vykonanie tejto metódy však tiež neznačí prechod do iného stavu. V tomto stave aktivita ostáva, dokiaľ nedôjde

znovu k obnoveniu aktivity, v tomto prípade sa stav zmení na *Resumed* a systém vráti uloženú inštanciu danej aktivity. Ak je aktivita úplne neviditeľná, stav aktivity sa zmení na *Stopped* a systém zavolá metódu `onStop()`.

- `onStop()` - ak je už aktivita neviditeľná pre používateľa, vstupuje do stavu *Stopped*. To sa môže stať napríklad vtedy, ak je spustená nová aktivita, ktorá pokrýva celú obrazovku. Tu by sa mali začať uvoľňovať zdroje, ktoré používateľ ďalej nepotrebuje. Tiež by malo dôjsť k uvoľneniu príjmača `BroadcastReceiver`, ak bol zaregistrovaný v metóde `onStart()`. Aby sa predišlo pretečeniu pamäti je dôležité v tomto mieste uvoľniť všetky prostriedky, ktoré programátor použil, nakoľko je možné, že systém zabije proces bez toho, aby zavolal poslednú metódu, životného cyklu aktivity, metódu `onDestroy()`. Z tohto stavu môže byť aktivita vrátená k interakcii s používateľom a systém zavolá metódu `onRestart()` a prejde do stavu *Started*, alebo dôjde k ukončeniu aktivity a systém zavolá metódu `onDestroy()` a prejde do stavu *Destroyed*.
- `onDestroy()` - metóda, ktorá je volaná pred zničením aktivity. Je to posledná metóda, ktorú aktivita dostáva a v tomto momente ju systém vyvolá, pretože bola vyvolaná metóda `finish()`, alebo systém ničí proces obsahujúci túto aktivitu kvôli šetreniu zdrojov. Táto metóda môže byť volaná aj v prípade, že dôjde k zmene orientácie obrazovky no hneď na to, je volaná metóda `onCreate()`, aby sa obnovil proces a s ňou aj komponenty, ktoré obsahuje a došlo tak k prekresleniu obrazovky. Táto metóda uvoľňuje všetky zdroje, ktoré neboli uvoľnené skoršími volaniami ako je napríklad metóda `onStop()`.



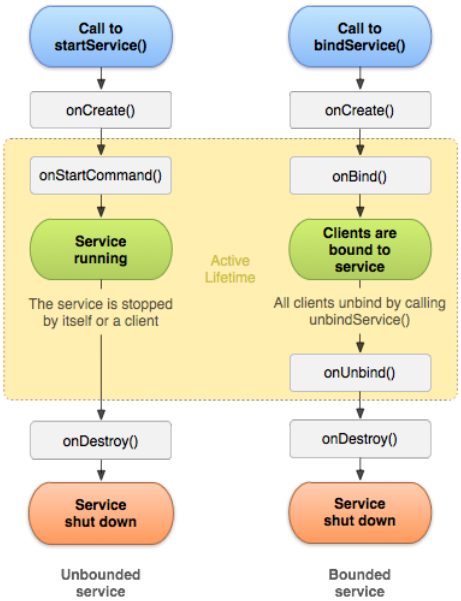
Obr. 2.2: Životný cyklus aktivity

2.3 Služby (dlhotrvajúce operácie) v systéme Android

Služba je komponenta aplikácie, ktorá má na starosti vykonávanie dlhodobých operácii na pozadí, bez užívateľského rozhrania. Službu môže spustiť iná komponenta, ako napríklad aplikácia, a táto služba beží na pozadí naďalej aj keď dôjde k prepnutiu na inú aplikáciu. Služba môže z pozadia spracovávať sieťové transakcie, prehrávať hudbu, spracovávať vstupy/výstupy súborov, alebo komunikovať s poskytovateľom obsahu. Existujú 3 rôzne typy služieb:

- **Foreground** - ide o službu, ktorá sa vykonáva na popredí a je teda viditeľná pre používateľa a je indikovaná zobrazením notifikácie v status bare. Ako príklad si môžeme uviesť prehrávanie hudby. Služby tohto typu pokračujú vo vykonávaní, aj keď používateľ nijako neinteraguje s aplikáciou.
- **Background** - služba, ktorá vykonáva operáciu na pozadí a používateľ ju tak priamo nezaznamenáva. Ide napríklad o získavanie aktuálnej polohy užívateľa.
- **Bound** - službu nazveme bound práve vtedy, ak sa k nej viaže nejaká komponenta (napríklad Aktivita), ktorá využíva volanie metódy `bindService()`. Táto služba ponúka rozhranie klient-server, ktorý umožňuje komponentom komunikovať so službou, posielať žiadosti, prijímať ich výsledky a komunikovať medzi procesmi. Služba beží len do doby, kým je k nej naviazaná iná komponenta, pri odviazaní sa služba zničí.

Ako môžeme vidieť na obrázku 2.3, životný cyklus služieb je jednoduchší ako vyššie uvedený životný cyklus aktivít.



Obr. 2.3: Životný cyklus služieb

Samotný životný cyklus služby, jeho vytvorenie až zničenie, môže vzniknúť z týchto dvoch možností

- **Spustenie služby** - pri tejto možnosti, je služba vytvorená, ak iná komponenta zavolá metódu `startService()`. Táto služba beží po dobu neurčitú a musí sa sama zasta-

viť vyvolaním metódy `stopSelf()`. Tiež ju môže zastaviť iná komponenta volaním metódy `stopService()`. V prípade, že je služba zastavená, systém ju zničí.

- **Viazaná služba** - druhá možnosť je tá, že služba je vytvorená keď iná komponenta (klient) volá metódu `bindService()`. Klient potom komunikuje so službou prostredníctvom rozhrania `IBinder`. Klient môže uzavrieť toto spojenie volaním metódy `unbindService()`. Na túto službu sa môžu viazať viacerí klienti, a ak sa všetci odpoja, je služba zničená systémom. Táto služba nemusí byť zastavovaná sama sebou.

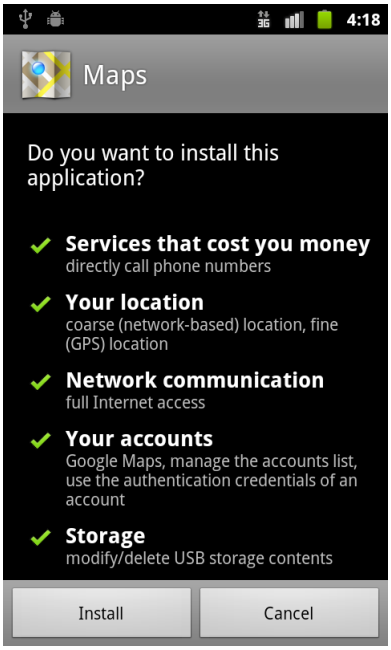
Tieto dve možnosti nie sú úplne oddelené a môžeme sa naviazať aj na službu, ktorá bola spustená vyvolaním metódy `startService()`. Implementovaním týchto metód môžeme potom sledovať dve vnorené cykly životného cyklu služby. Prvý cyklus, ktorý môžeme pozorovať na obrázku 2.3 vľavo, sa sústreďuje na čas, medzi volaniami metód `onCreate()`, kde môže služba podobne ako aktivita inicializovať rôzne komponenty, alebo vytvárať nové vlákna, a metódou `onDestroy()`, kde dôjde k uvoľneniu všetkých použitých zdrojov. Druhý cyklus môžeme vidieť na rovnakom obrázku vpravo, kde aktívna životnosť služby začína volaním metódy `onStartCommand()`, alebo `onBind()`. Každá metóda je potom spracovávaná v zmysle `Intentu`, ktorý bol predávaný ako parameter metódam `startService()`, alebo `bindService()`. Ak je služba spustená, jej aktívna životnosť sa končí súčasne s ukončením celej životnosti a ak je služba viazaná, tak aktívna životnosť končí návratom z metódy `onUnbind()`.

2.4 Zabezpečenie

Systém Android obsahuje prvotriedne bezpečnostné funkcie a spolupracuje s vývojármi a implementátormi zariadení, aby udržali platformu Android a celý ekosystém bezpečný. Robustný a bezpečnostný model je nevyhnutný na to, aby bol zabezpečený silný, energický ekosystém pre aplikácie a zariadenia postavené na tejto platforme. Výsledkom toho je aj to, že celý životný cyklus vývoja Androidu podlieha prísnemu bezpečnostnému programu. Systém Android je navrhnutý tak, aby chránil dôvernosť, integritu a dostupnosť používateľov, dát, aplikácii, zariadenia a siete.

Android bol navrhnutý s viacvrstvovou bezpečnosťou, ktorá je dostatočne flexibilná na to, aby podporovala otvorenú platformu, akou je Android a zároveň chránila všetkých používateľov. Android ponúka programátorom aplikácii pomoc pri riešení bezpečnostných otázok a to vydávaním stabilnej platformy. Tiež existuje tím, ktorý sa o bezpečnosť stará a kontroluje aplikácie a ich potencionálne zraniteľné miesta a navrhuje programátorom spôsoby, ako tieto problémy riešiť. Systém Android nezabúda ani na používateľov a dovoľuje im chrániť svoje súkromie tým, že im zobrazuje oprávnenia, ktoré môže daná aplikácia vykonávať (obrázok 2.4), a ak sa používateľovi nepáči, môže oprávnenie zakázať.

Android beží na širokej škále hardvérových konfigurácii a hoci je procesorovo-agnostický (beží na rôznych procesoroch rovnako), využíva výhody určitých hardvérových bezpečnostných funkcií, ako je napríklad `ARM eXecute-Never`[1] . Samotný operačný systém Android je postavený na jadre linuxu. Všetky prostriedky zariadenia, ako sú napríklad funkcie fotoaparátu, údaje GPS, telefónne funkcie, sú prístupné cez operačný systém. Aplikácie určené pre platformu Android sú najčastejšie písané v programovacom jazyku `Java` a bežia v Android runtime (ART). Aplikácie bežia v rámci bezpečnostného prostredia, ktorý je obsiahnutý



Obr. 2.4: Príklad oprávnení, ktoré aplikácia požaduje

v rámci aplikačného sandboxu. Aplikácie tak získavajú osobitnú časť súborového systému, v ktorom môžu zapisovať súkromné údaje, vrátane databáz a nespracovaných súborov.

2.4.1 Zabezpečenie na úrovni jadra

Základom pre mobilné počítačové prostredie je jadro linuxu s operačným systémom Android, ktorý obsahuje niekoľko kľúčových bezpečnostných funkcií, ktorými sú užívateľsky založený model oprávnení, izolovanie procesov, rozšírený mechanizmus pre bezpečnú medzi procesorovú komunikáciu a schopnosť odstrániť nepotrebné a potenciálne neisté časti jadra. Základným bezpečnostným cieľom jadra linuxu je navzájom izolovať medzi sebou zdroje používateľov. Linux tak zabraňuje používateľovi A čítať súbory používateľa B, zabezpečuje, aby používateľ A nevyčerpal pamäť používateľa B, zabezpečuje, aby používateľ nevyčerpal zdroje používateľa B a zabezpečuje, aby používateľ A nevyčerpal zariadenia používateľa B (napríklad GPS, Bluetooth).

2.4.2 Aplikačný sandbox

Systém Android priraďuje každej aplikácii jedinečné používateľské ID (UID) a spúšťa ho ako samostatný proces. Tento prístup sa líši od ostatných operačných systémov, kde sa používajú viaceré aplikácie s rovnakými oprávneniami používateľa. Jadro vynucuje bezpečnosť medzi aplikáciami a systémom na úrovni procesov prostredníctvom štandardných vlastností linuxu, ako sú identifikátory používateľov a skupín, ktoré sú priradené aplikáciam. Ak sa napríklad aplikácia A pokúsi urobiť niečo škodlivé, ako je čítanie údajov aplikácie B, operačný systém to chráni, pretože aplikácia A nemá príslušné používateľské privilégia. Sandbox je teda jednoduchý, kontrolovateľný a založený na oddeľovaní procesov a oprávnení v štýle `UNIX`-u.

[1] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360f/CACHFICI.html

Vďaka tomu že je aplikačný sandbox v jadre, sa tento bezpečnostný model rozširuje aj na natívny kód a aplikácie operačného systému. Všetok softvér, ktorý beží nad jadrom, ako sú knižnice operačného systému, alebo aplikácie, beží v aplikačnom sandboxe. V systéme Android neexistujú žiadne obmedzenia, ako písať aplikáciu, tak aby bola docielená potrebná bezpečnosť a v tomto ohľade je natívny kód rovnako bezpečný ako aj interpretovaný kód. Aby sa predošlo tomu, že dôjde k poškodeniu pamäte v jednej aplikácii a neohrozilo to ostatné aplikácie a bezpečnosť zariadenia, bežia aplikácie v aplikačnom sandboxe na úrovni operačného systému. Takáto chyba v pamäti umožní ľubovoľné spustenie kódu len v kontexte konkrétnej aplikácie s povoleniami, ktoré vytvoril operačný systém. Treba však podotknúť, že tak ako všetky bezpečnostné funkcie, ani aplikačný sandbox nie je nezlomný.

2.4.3 Kryptografia

Pre aplikácie Android tiež poskytuje sadu kryptografických API. Patria k nim implementácie štandardných a bežne používaných kryptografických primitív, ako sú `AES`, `RSA`, `DSA`, `SHA`. Rozhrania API sa však používajú aj na protokoly vyššej úrovne ako sú protokoly `SSL` a `HTTPS`. Vo verzii Android 4.0 bola predstavená trieda `KeyChain`, ktorá umožňuje aplikáciam používať úložisko systémových poverení pre súkromné kľúče a reťazce certifikátov.

2.4.4 Konfigurácia zabezpečenia siete

Táto funkcia umožňuje aplikáciam prispôsobiť nastavenia zabezpečenia siete v bezpečnom deklaratívnom konfiguračnom súbore bez úpravy kódu aplikácie. Tieto nastavenia je možné nakonfigurovať pre konkrétne domény a pre konkrétnu aplikáciu. Kľúčové funkcie:

- **Vlastné nastavenie dôvernosti** - dochádza k modifikácii certifikačných autorít, ktoré aplikácia považuje za dôveryhodné. Napríklad dôvera konkrétnym certifikátom s vlastným podpisom, alebo obmedzenie súboru verejných certifikačných autorít, ktorým aplikácia verí.
- **Prepisy len pri ladení** - pre ladiace účely, kedy sa neohrozuje základňa používateľov, ktorí aplikáciu už používajú.
- **Obmedzenie návštevnosti** - chránenie aplikácie pred pripojeniami pomocou nešifrovaného `HTTP` protokolu namiesto protokolu `HTTPS`
- **Pripojenie certifikátu** - obmedzenie, aby aplikácia dôverovala len k pripojeným certifikátom.

2.5 Prístup k zdrojom

Okrem vyššie spomenutého nám, operačný systém Android ponúka tiež možnosti pristupovať k zdrojom zariadenia. Týmito zdrojmi sú úložisko súborov, poloha zariadenia (známa ako GPS), kamera, senzory a rôzne ďalšie prvky konektivity, ktoré nám dovoľujú spájať sa s inými zariadeniami a patria medzi nich napríklad technológia Bluetooth, NFC, USB.

2.5.1 Úložisko súborov

Android ponúka niekoľko možností ako je možné ukladať údaje aplikácii. Ich použitie sa líši od veľkosti, či viditeľnosti pre ostatné aplikácie. Poďme si bližšie popísať aké možnosti nám systém ponúka:

- **Interné úložisko súborov** - v predvolenom nastavení sú súbory uložené do interného ukladacieho priestoru a k týmto súborom nemôžu pristupovať ostatné aplikácie. Pri odinštalovaní aplikácie sú súbory nachádzajúce sa v tomto úložisku odstránené. Android nám tiež ponúka možnosť uchovávať niektoré údaje len dočasne, k čomu slúži špeciálny adresár vyrovnávacej pamäte, ktorý má každá aplikácia, avšak pri zaplnení interného úložiska nám tieto súbory systém zmaže. Pri odstránení sa tieto súbory taktiež odstránia.

- **Externé úložisko súborov** - tento úložný priestor je nazvaný externým, pretože prístup k nemu nie je zaručený. Používatelia môžu tento priestor pripojiť k počítaču ako externé pamäťové zariadenie a môže byť aj fyzicky odstrániteľné (SD karta). Súbory uložené na externom úložisku sú čitateľné pre všetkých používateľov a môžu byť modifikované. Toto úložisko by malo slúžiť na ukladanie používateľských údajov, ktoré by mali byť prístupné iným aplikáciam a mali by ostať uložené aj v prípade, že užívateľ aplikáciu odinštaluje. Súbory je možné ukladať aj do adresáru špecifického pre danú aplikáciu, najmä ak ide o veľké dáta a tieto údaje sa pri odinštalovaní aplikácie odstránia.

- **Zdieľané preferencie** - toto riešenie je vhodné, ak nie je potrebné ukladať množstvo informácií a nevyžaduje sa ani žiadna štruktúra. Ide vlastne o ukladanie primitívnych dát vo formáte kľúč-hodnota. Pre zjednodušenie práce s týmito primitívami existuje rozhranie `SharedPreferences`, ktoré uľahčuje ukladanie týchto typov. Toto rozhranie ukladá páry kľúč-hodnota do súborov `XML`, ktoré pretrvávajú v reláciách používateľov, aj keď je aplikácia zabitá. Úložisko je vhodné pre ukladanie jednoduchých dátových typov a údajov, ako je napríklad uloženie najvyššieho skóre užívateľa.

- **Databázy** - Android poskytuje plnú podporu `SQLite` databázam. Pri vytváraní databázy, je databáza prístupná len aplikácii, ktorá ju vytvorila a je vhodná pre ukladanie štruktúrovaných údajov.

2.5.2 Prístup k polohe zariadenia

Jednou z jedinečných funkcií mobilných aplikácii je informovanosť o polohe. Používatelia mobilných zariadení si so sebou berú svoje zariadenia a pridávanie informácii o lokalite do aplikácie ponúka používateľom viac kontextuálny zážitok. Rozhrania pre poskytovanie lokality umožňujú do aplikácie pridávať informáciu o polohe pomocou automatického sledovania polohy, geofencingu a rozpoznávaniu činností.

2.5.3 Snímače

Väčšina zariadení so systémom Android má vstavané snímače, ktoré merajú pohyb, orientáciu a rôzne podmienky prostredia. Tieto snímače sú schopné poskytovať hrubé dáta s vysokou presnosťou. Sú užitočné pri polohovaní trojrozmerného zariadenia, alebo pri sledovaní zmien okolitého prostredia. Niektoré snímače sú hardvérové, iné zas softvérovo založené. Android podporuje tri široké kategórie senzorov:

- **Snímače pohybu** - snímače, ktoré merajú zrýchlenie a silu otáčania pozdĺž troch osí. Táto kategória zahŕňa senzory ako akcelometre, gravitačné senzory, gyroskopy a rotačné vektorové snímače.

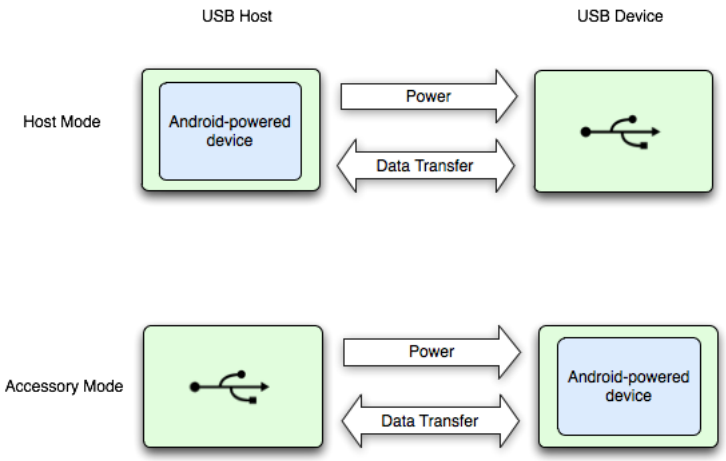
- **Snímače prostredia** - tieto snímače merajú rôzne environmentálne parametre, ako je teplota, tlak okolitého vzduchu, a vlhkosť. Do tejto kategórie spadajú barometre, fotometre a teplomery.
- **Snímače polohy** - merajú fyzickú polohu zariadenia. Tu patria senzory orientácie a magnometre.

2.5.4 Prvky konektivity

Podpora technológie **Bluetooth** nie je výnimkou ani u tohto operačného systému. Táto technológia umožňuje zariadeniam bezdrôtovo vymieňať si dáta s inými Bluetooth zariadeniami. Android ponúka aplikačný rámec pre prácu s Bluetooth funkciami. Prostredníctvom tohto rámca môžu aplikácie vykonávať skenovanie ďalších Bluetooth zariadení, pripojenie sa k iným zariadeniam prostredníctvom vyhľadávania služby, preniesť údaje do a z iných zariadení.

Ďalšou podporovanou službou je **Near Field Communication** skôr známa pod skratkou NFC. Služba umožňuje zdielať malé dáta medzi štítkami NFC a zariadením, ktoré túto technológiu podporuje, alebo medzi dvoma takýmito zariadeniami. Existujú jednoduché štítky, ktoré ponúkajú len sémantické čítanie a písanie a niekedy umožňujú len čítanie karty. Na druhej strane existujú aj zložité štítky, ktoré ponúkajú rôzne matematické operácie a obsahujú kryptografický hardvér, na overovanie prístupu.

Android podporuje aj rôzne periférie **USB** a to pomocou dvoch režimov a to USB príslušenstvo (USB accessory) a USB hostiteľ (USB host), ktoré sú znázornené aj na obrázku 2.5. V prvom menovanom funguje externý USB hardvér ako hostiteľ a ako príklad si môžeme uviesť dokovaciu stanicu. V hostiteľskom režime funguje zariadenie ako hostiteľ a medzi príklady patria napríklad digitálne kamery.



Obr. 2.5: USB režimy

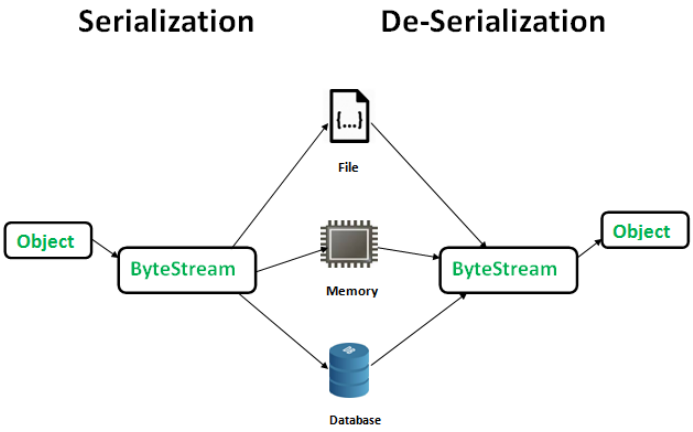
Kapitola 3

Použité technológie

V tejto kapitole si predstavíme technológie, ktoré boli pri vypracovávaní tejto práce použité. Najskôr si povieme, čo je to serializácia a ako funguje. Následne si predstavíme reflexiu, kde pôjdeme viac do hĺbky. Rozoberieme si, čo reflexia je, aké mechanizmy používa a predstavíme si aj niektoré metódy, ktoré budú v tomto projekte používané. Ďalej si povieme niečo o technológii Django a REST API a kapitolu ukončíme výkladom o Firebase Cloud Messagingu, čo je systém na posielanie správ do Android zariadenia.

3.1 Serializácia

Ako uvádza [5] serializácia objektov v jazyku Java dovoľuje previesť každú inštanciu triedy implementujúcu rozhranie `Serializable`, alebo `Externalizable` na sekvenciu bajtov (serializácia), ktorá môže byť neskôr použitá pre úplnú rekonštrukciu stavov pôvodného objektu (deserializácia). Objekty, ktoré sú prevádzané do tejto sekvencie často tvoria vzťah s inými objektmi, a tak ak je objekt uložený, uložia sa aj všetky objekty, ktoré sú prístupné, aby nedošlo k poškodeniu vzťahov medzi týmito objektmi. Serializované objekty je možné prenášať cez počítačovú sieť, alebo medzi zariadeniami. Pre lepšiu predstavu, ako serializácia funguje nám môže poslúžiť obrázok 3.1.



Obr. 3.1: Serializácia a deserializácia v jazyku Java

3.2 Reflexia

Ako hovorí [4] reflexia je schopnosť bežiaceho programu preskúmať sám seba jeho softvérové prostredie a zmeniť svoje správanie. Aby program mohol vykonať toto sebaskúmanie, musí poznať svoju reprezentáciu. Tieto informácie nazývame `metadáta`. V objektovo orientovanom svete sú `metadáta` organizované do objektov nazývaných `metaobjekty` a samotná kontrola `metaobjektov` počas behu sa nazýva `introspekcia`. Vo všeobecnosti existujú tri techniky reflexie, pomocou ktorých môžeme ľahko zmeniť správanie, a to priama modifikácia metaobjektu, operácie na používanie `metadát` (napríklad dynamické vyvolanie metód) a intercession, kedy ide o reflexnú schopnosť, ktorá mení správanie programu tým, že priamo ovládne toto správanie.

Podľa [6] sa jedná o pokročilú vlastnosť jazyka, ktorá umožňuje vykonávať operácie, ktoré by inak nebolo možné vykonávať, napríklad umožňuje obísť zapuzdrenie (sprístupňuje privátnych členov). To však znamená riziko narušenia správneho behu programu a aj vďaka tomu, by mala byť reflexia používaná a nemali by ju používať ľudia s nedostatočnými znalosťami. Veľkou výhodou, ktorú nám reflexia umožňuje, je tá, že aplikácie sa ľahšie prispôsobujú meniacim sa požiadavkám.

Samotnú reflexiu a introspekciu si môžeme predstaviť tak, že sa na seba pozeráme do zrkadla. Zrkadlo nám poskytuje reprezentáciu samého seba, náš odraz, ktorý môžeme skúmať. Skúmanie samého seba v zrkadle nám poskytuje dôležité informácie o tom, čo máme oblečené, či nie sme špinavý, a tiež nám zrkadlo povie niečo o našom správaní. Napríklad či úsmev vyzerá úprimne, alebo či nejaké gesto nevyzerá príliš prehnane. To nám môže pomôcť pre pochopenie toho, ako prispôsobovať svoje správanie aby sme spravili čo najlepší dojem na iných ľudí. Podobne musí pri introspekcii program poznať vlastnú reprezentáciu, ktorá je najdôležitejším štrukturálnym prvkom reflexného systému. Skúmaním vlastnej reprezentácie dokáže program získať správne informácie o svojej štruktúre a správne sa rozhodovať pri vykonávaní dôležitých rozhodnutí.

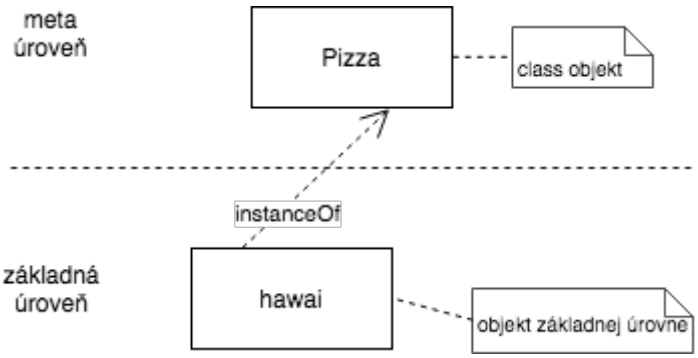
```
public static void setObjectColor( Object object, Color color ) {
   Class cls = object.getClass();
   Method method = cls.getMethod( "setColor", new Class[] {Color.class} );
   method.invoke( obj, new Object[] {color} );
}
```

Výpis 3.1: Použitie reflexie - pseudokód

Vo výpise 3.1 je znázornené využitie reflexie. Predstavme si nasledujúci problem, ktorý nám vznikol pri implementácii používateľského prostredia, kedy sú použité rôzne štandardné vizuálne komponenty jazyka Java (produkty tretej strany) a tieto komponenty sú integrované do aplikácie. Každá komponenta poskytuje metódu `setColor`, ktorá ako parameter očakáva typ `java.awt.Color` . Problém je v tom, že hierarchia je nastavená tak, že jediná spoločná základná trieda je `java.lang.Object` a tieto komponenty nemôžu odkazovať pomocou typu, ktorý podporuje túto metódu. Metóda `setObjectColor` pomocou reflexie vyžaduje triedu objektu, ktorý bol poslaný v parametroch metódy a následne v triede hľadá metódu `setColor`, ktorá ako parameter vyžaduje typ `Color`. Obe tieto volania sú istou formou introspekcie, ktoré umožňujú aby sa program preskúmal. Posledný riadok volá výslednú metódu objektu a predáva jej parameter typu `Color` - toto volanie možno označiť ako **dynamické vyvolanie**, čo je vlastnosť reflexie, ktorá umožňuje programu vyvolať metódu objektu v dobe behu, bez určenia metódy v čase kompilácie. Toto riešenie zistí, ktorá

metóda `setColor` je k dispozícii počas behu programu pomocou introspekcie a vyvolá metódu, ktorá sa práve vyžaduje. V tomto výpise sa tiež používajú inštancie `Class` a `Method` na vyhľadanie odpovedajúcej metódy, a tieto objekty sú súčasťou vlastnej reprezentácie Javy. Tieto objekty nazývame `metaobjekty` a uchovávajú informácie o programe.

Pre lepšie pochopenie `metaobjektov` uvažujme príklad 3.2. Na dosiahnutie hlavného účelu programu hovoríme o objektoch na **základnej úrovni**, čo je na obrázku objekt `hawai`, ktorý je inštanciou objektu `Pizza` a s týmito objektmi pracujeme pri vývoji. Objekt `Pizza` je zasa objekt (`metaobjekt`) na **meta úrovni**. `Metaobjekty` sú pri reflexii výhodou a poskytujú všetky informácie, ktoré sú potrebné, a často poskytujú aj spôsoby na zmenu štruktúry programu, jeho správania alebo údajov.



Obr. 3.2: Základná a meta úroveň v reflexii

3.2.1 Vyhľadávanie metódy počas behu programu

Mohli sme si všimnúť, že v metóde 3.1, ktorá je uvedená ako príklad, má v parametroch `object`, ktorý je typu `java.lang.Object` a túto triedu dedí každá trieda v Jave. Táto trieda obsahuje metódu `getClass`, ktorá je často používaná pri začatí reflexného programovania, nakoľko mnohé reflexívne úlohy vyžadujú objekty, ktoré reprezentujú triedy. Metóda vracia inštanciu `java.lang.Class` a tieto inštancie sú `metaobjektmi`, ktoré Java používa na reprezentáciu tried, ktoré tvoria program. Tieto objekty sú najdôležetejším druhom `metaobjektov`, pretože všetky Java programy sa skladajú z týchto tried.

`Class` objekty nám poskytujú programovacie `metadáta` o `fieldoch` triedy, konštruktoroch, metódach a vnorených triedach. Poskytujú aj informácie o hierarchii dedičnosti a poskytujú prístup k reflexívnym vlastnostiam.

Trieda `Class` nám poskytuje tieto metódy[1] na skúmanie metód:

- `Method getDeclaredMethod(String name, Class[] parameterTypes)` - vracia objekt `Method`, ktorý reflektuje zadanú deklarovanú metódu triedy, alebo rozhrania reprezentovaného týmto `Class` objektom.
- `Method[] getDeclaredMethods()` - vracia pole objektov `Method`, ktoré reflektuje všetky metódy deklarované triedou, alebo rozhranie reprezentované týmto `Class` objektom.
- `Method getMethod(String name, Class[] parameterTypes)` - vracia objekt `Method`, ktorý reflektuje zadanú metódu, ktorá je verejná, alebo rozhranie reprezentované týmto objektom `Class`.

- `Method[] getMethods()` - vracia pole objektov `Method`, ktoré reflektuje všetky verejné metódy, alebo rozhranie reprezentované týmto `Class` objektom, vrátané tých, ktoré deklaruje trieda, alebo rozhranie a tiež tie, ktoré boli zdedené z rodičovských tried, alebo rodičovských rozhraní.

3.2.2 Trieda `Method` a dynamické vyvolanie

Táto trieda sa nachádza v balíčku `java.lang.reflect` a je to trieda `metaobjektov`, ktorá reprezentuje metódy. Každý objekt tejto triedy poskytuje informácie o metóde, jej návratovom type, názvu, typy parametrov, typ výnimky prípadne anotáciu. Objekt `Method` nám taktiež umožňuje zavolať metódu, ktorú predstavuje. Metódy, ktoré okrem iných táto trieda obsahuje[2]:

- `Annotation[] getDeclaredAnnotations()` - vracia všetky anotácie, ktoré sú priamo prítomné v tomto prvku.
- `Class getDeclaringClass()` - vracia objekt `Classs` reprezentujúci triedu, alebo rozhranie, ktoré deklaruje metódu reprezentovanú týmto `Method` objektom.
- `Class[] getExceptionTypes()` - vracia pole objektov `Class`, ktoré reprezentujú typy výnimiek, ktoré boli deklarované na zahodenie.
- `int getModifiers()` - vracia modifikátory jazyku Java pre metódu reprezentovanú objektom `Method` v podobe celého čísla.
- `String getName()` - vracia názov metódy ako reťazec.
- `Class[] getParameterTypes()` - vráti pole objektov `Class`, ktoré reprezentujú formálne typy parametrov, v poradí ako sú deklarované.
- `Class getReturnType()` - vráti objekt `Class`, ktorý reprezentuje formálny návratový typ metódy.
- `Object invoke(Object obj, Object[] args)` - vyvolá metódu reprezentovanú objektom `Method` na zadanom objekte s určenými parametrami.

Dynamické vyvolanie umožňuje programu zavolať metódu na objekt počas behu programu, bez určenia metódy v čase kompilácie. Prvý parameter metódy `invoke` je cieľ volania metódy, alebo objekt, ktorý túto metódu vyvolá. Druhým parametrom na vyvolanie je pole typu `Object`, ktorý metóda `invoke` predá dynamicky vyvolanej metóde ako jej aktuálne parametre.

3.3 Django a REST API

Táto technológia bola zvolená nakoľko Django je vysoko kvalitný webový framework napísaný v jazyku Python, ktorý podporuje rýchly vývoj a čistý pragmatický dizajn. Podľa [2] je vytváraný skúsenými vývojármi a rieši množstvo problémov, ktoré sa objavujú pri písaní aplikácii. Tento framework je zdarma a je open source a podľa [**?**] obsahuje množstvo

[1] https://docs.oracle.com/javase/7/docs/api/java/lang/Class.html
[2] Zoznam všetkých metód: https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Method.html

nástrojom pre tvorbu webových aplikácii. Jeho hlavnou výhodou je jeho tzv. objektovo-relačný mapovač (ORM), ktorý je zproztredkovateľom medzi dátovým modelom a relačnou databázou. Dátove moduly sú napísané v jazyku Python, z čoho vyplýva výhoda, ktorou je eliminácia písana databázových príkazov v jazykoch ako je SQL. Webové aplikácie vyvíjané vo frameworku využívajú podľa [7] architektonický vzor model-view-controller (MVC).

Vývoj aplikácie pomocou Django sprevádza počiatočné nastavenie. Konkrétne je potrebné automaticky vygenerovať kód, ktorý vytvorí projekt Django, ktorý obsahuje súbor nastavení pre konkrétnu inštanciu Djanga, konfiguráciu databázy a ďalšie špecifické nastavenie. Po nainštalovaní Djanga na náš operačný systém sa o toto generovanie automaticky postará príkaz, ktorý je uvedený v 3.2.

```
django-admin startproject project
```

Výpis 3.2: Príkaz na vytvorenie Django projektu

kde `project` označuje názov nášho projektu. Tento príkaz nám vygeneruje následovné súbory:

- Adresár `project/` je len kontajner projekt a na jeho mene nezáleží, nakoľko ho môžeme premenovať na čokoľvek iné
- `manage.py` - ide o pomôcku pre príkazový riadok, ktorý umožňuje rôznymi spôsobmi komunikovať s týmto Django projektom.
- Vnorený adresár `project/` je balíkom zaobaľujúci projekt, v budúcnosti potrebné pre importovanie
- `project/__init__.py` - prázdny súbor, ktorý informuje Python, že tento adresár by mal byť považovaný za balík Pythonu.
- `project/settings.py` - nastavenia/konfigurácia pre konkrétny Django projekt.
- `project/urls.py` - súbor s deklaráciami URL pre tento projekt
- `project/wsgi.py` - vstupný bod pre webové servery kompatibilné so službou WSGI.

Pre spustenie serveru na ktorom náš projekt beží sa používa príkaz vo výpise 3.3.

```
python manage.py runserver
```

Výpis 3.3: Príkaz na spustenie Django serveru

Pred viac ako desať rokmi predstavil Roy Fielding, **REpresentational State Transfer** v článku [3] REST ako nový architektonický štýl. V priebehu rokov získal REST dynamiku vďaka svojej popularite pri budovaní webových služieb. RESTful architektúra musí spĺňať formálne obmedzenia. Dôležitým obmedzením je napríklad, Že aplikácia musí byť rozdelená na model **klient-server** a server musí ostať úplne bezstavový.

Kombinácia Djanga s nástrojom REST je bežnou praxou pri vytváraní webových stránok bohatých na údaje. V komunite Django existuje množstvo opakovane použiteľných aplikácii, ktoré nám pomôžu dodržať zásady REST pri vytváraní API. Jeden z najpopulárnejších je `django-rest-framework`. a jeho hlavnými výhodami sú:

- vylepšenie použiteľnosti pridaním webového prehľadávanie API
- pravidlá overovania vrátane balíkou pre `OAuth1` a `OAuth2`
- serializácia, ktorá podporuje objektovo-relačné mapovanie
- rozsiahla dokumentácia a skvelá podpora komunity

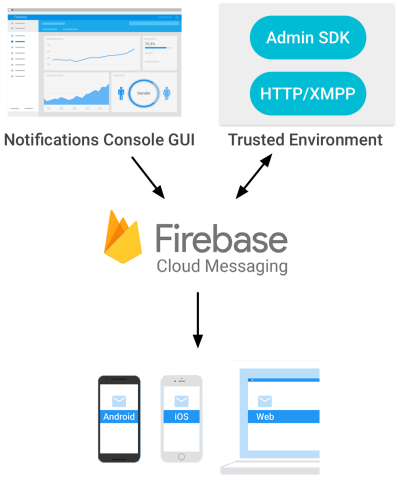
3.4 Firebase Cloud Messaging

Firebase Cloud Messaging (ďalej FCM) je riešenie pre zasielanie správ medzi rôznymi platformami, ktoré umožňujú spoľahlivé doručenie správ. Pomocou nástroja FCM môžme oznamovať klientskej aplikácii, že je k dispozícii nový e-mail, alebo iné údaje. Nástroj FCM nám dovoľuje posielať notifikačné správy, pre udržanie pozornosti užívateľa. Maximálna možná veľkosť správy, ktorá je prenášaná do aplikácie klienta je 4 kB. Medzi kľúčové možnosti tohoto nástroja patria

- odosielanie notifikačných, alebo dátových správ
- všestranné zacielenie na správy - správy sa dajú zacieliť podľa typu zariadenia, na skupiny zariadení a na zariadenia, ktoré sa prihlásili na odber nejakej témy.
- odosielanie správ z klientských aplikácii - odosielanie správy zo zariadení späť na server

Ako je zobrazene na obrázku 3.3. Implementácia FCM obsahuje dve hlavné komponenty na odosielanie a prijímanie:

- dôveryhodné prostredie, ako sú funkcie Firebase cloudu, alebo aplikačný server, na ktorých je možné vytvárať, zacieľovať a odosielať správy
- aplikácia klienta Android, iOS, alebo web, ktorá príjma správy



Obr. 3.3: Diagram architektúry Firebase Cloud Messaging

Firebase Cloud Messaging nám ponúka širokú škálu možností a funkcií pre zasielanie správ. Existujú dva typy správ, ktoré môžme prostredníctvom FCM posielať a to:

1. Notifikačné správy - správy, ktoré sú niekedy považované za "správy na displeji". Tieto správy sú automaticky spracovávané FCM SDK.
2. Dátové správy - správy, ktoré spracováva aplikácia klienta

Notifikačné správy obsahujú preddefinovanú sadu kľúčov viditeľných pre užívateľa. Naopak, dátové správy obsahujú iba užívateľsky definované páry kľúč-hodnota.

Kapitola 4

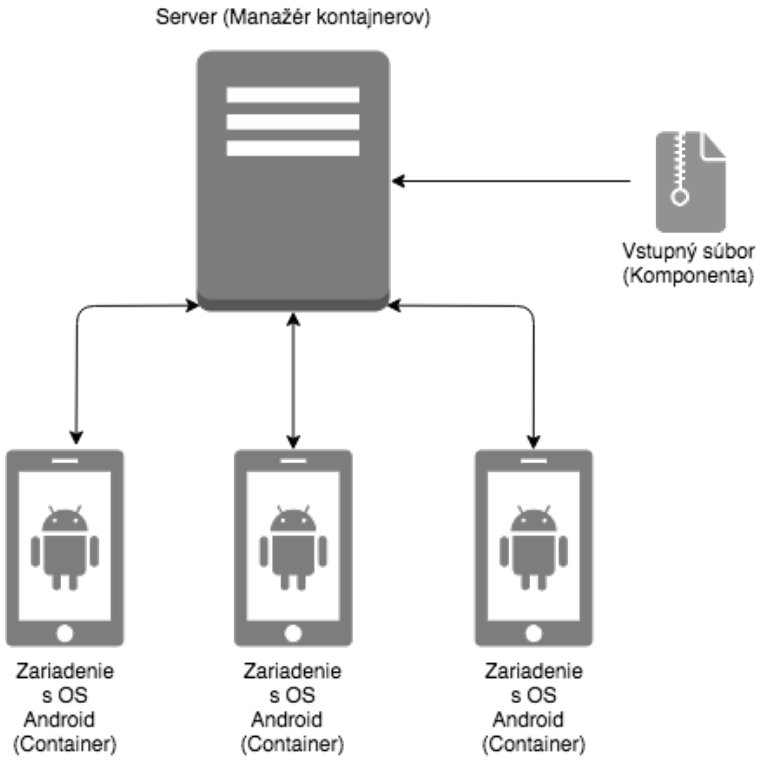
Existujúce riešenia

Kapitola 5

Návrh aplikácie

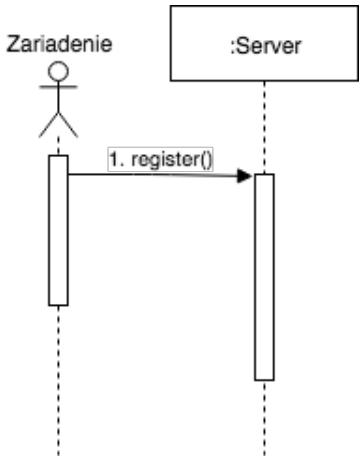
Cieľom tejto práce je navrhnúť kontajner pre komponenty bežiace na operačnom systéme Android, to je rozhranie komponenta-kontajner, spôsob prevádzky a distribúcie komponentov, ich životný cyklus a možnosť tvorby adaptérov pre iné typy komponentov. V tejto kapitole je uvedený návrh architektúry aplikácie, akým spôsobom komponenta komunikuje s kontajnermi, možnosti a spôsob distribúcie komponentov a ich životný cyklus. Následne si ukážeme návrh pre tvorbu iných komponentov, aby bol užívateľ schopný implementovať vlastnú komponentu a distribuovať ho do kontajneru, kde bude spracovávaný.

5.1 Návrh architektúry aplikácie



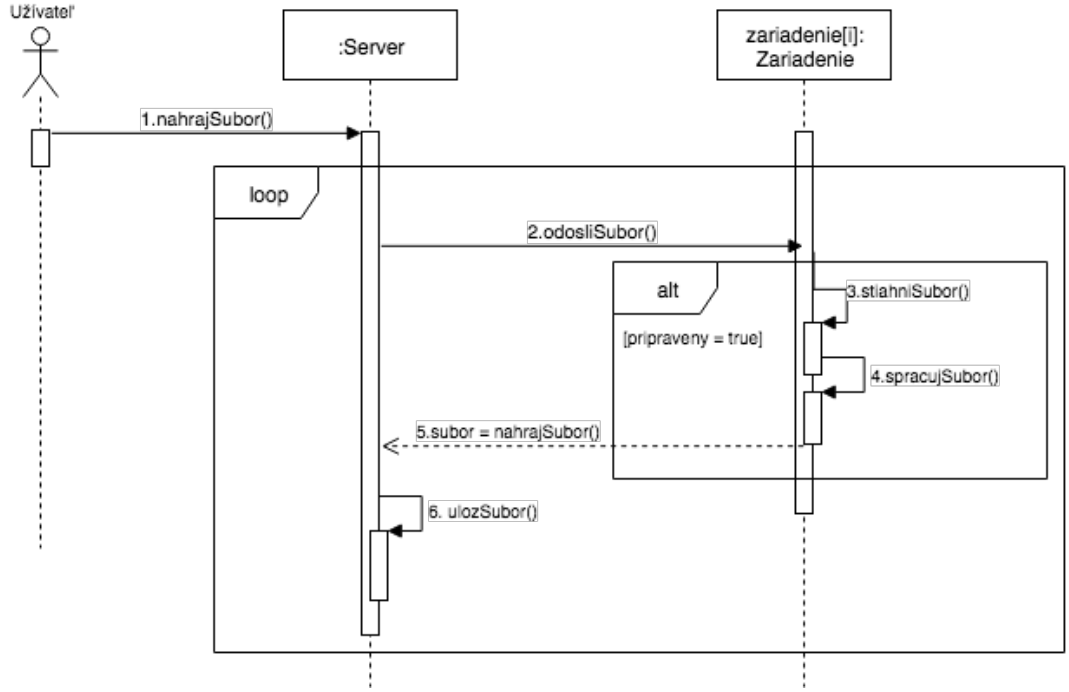
Obr. 5.1: Návrh architektúry aplikácie

Ako je zobrazené na obrázku 5.1 vstupom aplikácie je server, ktorý sa správa ako manažér kontajnerov. Na server je nahraná komponenta, ktorá má byť rozdistribuovaná na kontajnery. Zariadenie obsahujúce kontajner sa pri inštalácii zaregistruje na manažérovi, ktorý si túto informáciu uloží. Táto operáciu je vyjadrená na obrázku 5.2.



Obr. 5.2: Diagram sekvencie pre registráciu nového zariadenia

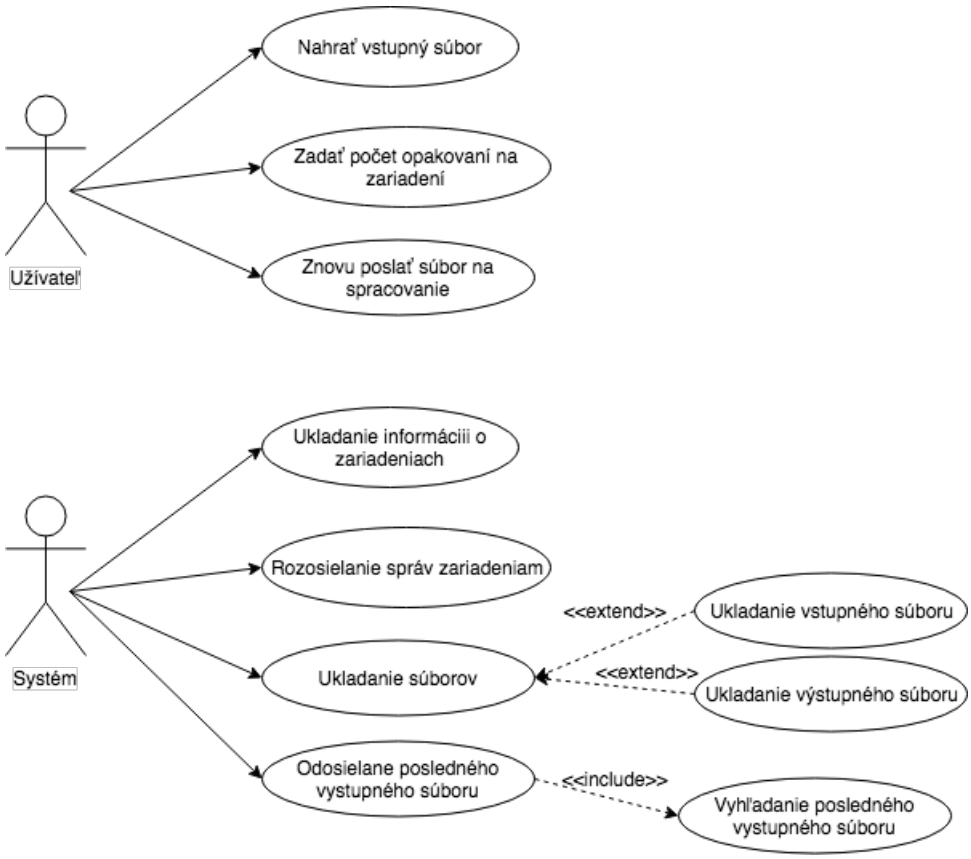
Po nahratí komponenty, manažér odošle zaregistrovaným zariadeniam správu, s obsahom súboru, ktorý je potrebné spracovať. Ak je zariadenie pripravené na spracovanie, súbor stiahne a spustí životný cyklus kontajneru. Po vykonaní operácii nad vstupným súborom je súčasný stav uložený do výstupného súboru, ktorý je odoslaný serveru. Server si následne tento súbor uloží. Priebeh týchto operácii je znázornený diagramom sekvencie na obrázku 5.3.



Obr. 5.3: Diagram sekvencie pre spracovanie vstupného súboru

5.2 Návrh serverovej časti aplikácie

Hlavnou úlohou serveru je ukladanie súborov a posielanie požiadaviek na ich spracovanie, no tiež bude server využívaný na registráciu zariadení, ktoré budú čakať na prijatie požiadavky. Ako je možné vidieť na obrázku 5.4, v tomto systéme sú zahrnuté dve role a to užívateľ a samotný systém. Užívateľ zahájí prácu serveru tým, že nahrá vstupný súbor, ktorý ma byť spracovávaný registrovanými zariadeniami. Následne môže zadať voliteľný počet opakovaní, ktoré majú byť vykonané na zariadení a to z toho dôvodu, že pri spracovávaní môže dôjsť k prerušeniu, čo sa počíta ako jedno opakovanie. Ak dôjde k registrácii nových zariadení, alebo dôjde k ukončeniu spracovávania na strane zariadení, a tým aj k nahraniu výstupného súboru, môže užívateľ odoslať súbor na spracovávanie znovu.



Obr. 5.4: Diagram prípadu užitia serverovej časti aplikácie

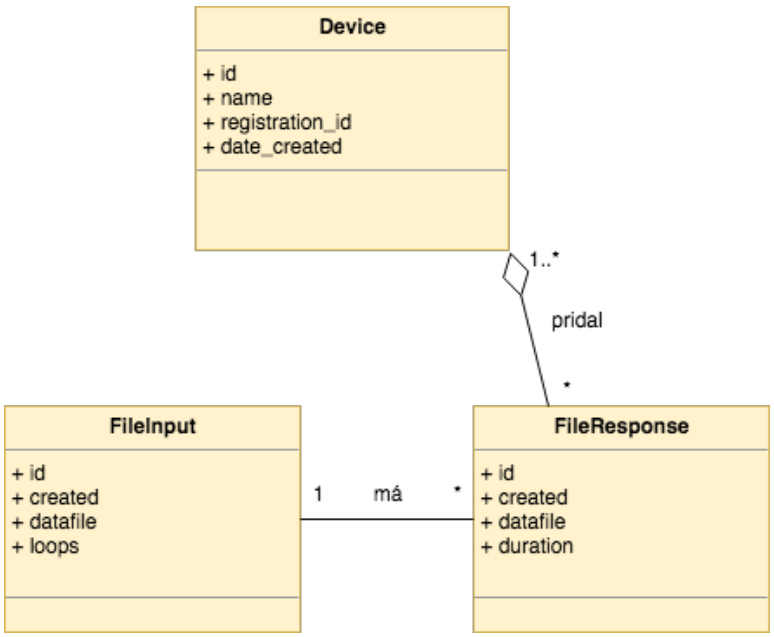
Samotný systém zabezpečuje ukladanie informácii o dostupných zariadeniach, dokáže im rozosielať správy na spracovávanie, ukladanie súborov, či už vstupných, alebo výstupných a odosielanie spracovaného výstupného súboru, ktorý bol na server nahraný ako posledný.

Vykonávanie všetkých týchto operácii bude navrhnuté pomocou REST rozhrania, pomocou ktorého budú zariadenia so serverom komunikovať. Všetky dôležité vlastnosti budú uložené v databáze, ktoré sú znázornené doménovým modelom na obrázku 5.5.

5.2.1 Doménový model

Doménový model obsahuje tri triedy. Prvou z nich je trieda `Device`, ktorá reprezentuje zariadenie z reálneho sveta. Pre lepšie analýzu výsledných výstupov je potrebné uchová-

vať si názov zariadenia, podľa ktorého si môžeme vyhľadať jeho technickú špecifikáciu. Tento atribút je v návrhu označený ako `name`. Ďalší atribút, `registration_id`, slúži k určeniu zariadenia, nakoľko je potrebné uchovávať si unikátne registračné číslo (token), ktorý dokáže jednoznačne určiť, o ktoré zariadenie ide. Posledným atribútom je atribút `date_created`, ktorý zaznamenáva dátum a čas registrácie zariadenia. Trieda `Device` agreguje triedu `FileResponse`, ktorú pridáva do databázy a týchto súborov môže byť viacero. Trieda `FileResponse` má reprezentovať výstupný súbor, ktorý bol vytvorený nejakým zariadením. Táto trieda obsahuje atribúty, ktoré nám ho jedinečne určujú(`id`), ďalej atribút určujúcu dátum a čas vytvorenia (atribút `created`, názov súboru (`datafile`) a dĺžku výpočtu (`duration`). Poslednou triedou je trieda `FileInput`, ktorá ma reprezentovať vstupný súbor a táto trieda môže mať viacero výstupov vo forme `FileResponse`. Trieda `FileInput`obsahuje atribúty, `id` - identifikácia súboru, `created` - dátum vytvorenia, `datafile` - názov súboru a `loops` - počet opakovaní.



Obr. 5.5: Doménový model serverovej časti aplikácie

5.2.2 REST API

Systém poskytuje REST API pre jednoduchú správu a distribúciu súborov medzi zariadeniami, bez autentizácie užívateľa. Pre všetky operácie potrebné na získanie registrovaných zariadení, registráciu zariadenia, získanie detailu zariadenia, či vymazanie zariadenia definuje server tieto REST volania:

- `GET /device` - zobrazenie všetkých registrovaných zariadení
- `POST /device` - registrácia nového zariadenia, vo formáte `JSON` sa očakáva názov zariadenia a unikátne identifikačné číslo
- `GET /device/{registration_id}` - získanie detailu zariadenia, kde `{registration_id}` je unikátne identifikačné číslo

- `DELETE /device/{registration_id}` - vymazanie zariadenia, kde `{registration_id}` je unikátne identifikačné číslo
- `GET /send-message` - zaslanie správy zariadeniam

Ďalšou skupinou operácii, ktorých úlohou je práca so vstupným súborom, či už ide o získanie všetkých vstupných súborov, detail vstupného súboru, jeho vytvorenie a mazanie, odoslanie správy všetkým čakajúcim zariadeniam, a opätovné posielanie súboru na spracovanie. Pre tieto operácie sú definované tieto REST volania:

- `GET /file-input` - zobrazenie všetkých vstupných súborov
- `POST /file-input` - nahrávanie vstupného súboru, vo forme `formData`, kde sa očakáva súbor a číslo udávajúce počet opakovaní na jednom zariadení
- `GET /file-input/{id}` - získanie detailu vstupného súboru, kde `{id}` je identifikačné číslo súboru
- `DELETE /device/{id}` - vymazanie vstupného súboru, kde `{id}` je identifikačné číslo súboru
- `POST /send-all-device/{id}` - opätovné zaslanie vstupného súboru, kde `{id}` udáva identifikačné číslo súboru

Poslednou skupinou operácií slúži na prácu s výstupným súborom. Tieto operácie nám poskytujú zoznam všetkých nahraných výstupných súborov, jednotlivé detaily súborov, vytvorenie súboru, alebo jeho vymazávanie. Operácie sú definované následovne:

- `GET /file-response` - zobrazenie všetkých výstupných súborov
- `POST /file-response` - nahrávanie výstupného súboru, vo forme `formData`, kde sa očakáva súbor, identifikačné číslo zariadenia, názov vstupého súboru a nepovinný údaj, čas spracovávania na zariadení
- `GET /file-response/{id}` - získanie detailu výstupného súboru, kde `{id}` je identifikačné číslo súboru
- `DELETE /file-response/{id}` - vymazanie výstupného súboru, kde `{id}` je identifikačné číslo súboru
- `POST /get-last-result` - získanie posledného nahraného súboru, kde sa očakáva názov vstupného súboru vo formáte `JSON`

5.3 Návrh kontajneru

Navrhovaný kontajner je komplexnejší ako vyššie spomínaný server a má na starosti viacero úloh, medzi ktoré patria preskumávanie vstupného súboru pomocou reflexie, načítavanie a ukladanie súborov. Komplexný výpis operácii je zobrazený v diagrame prípadov užitia 5.6. Tento diagram obsahuje jednu rolu a to zariadenie, ktoré tieto operácie môže vykonávať.

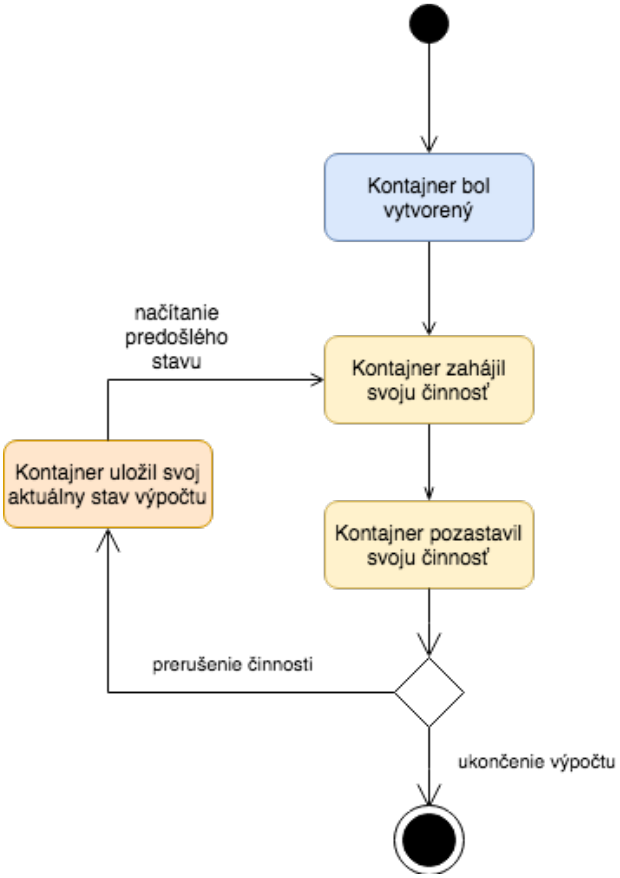
Vstupnou operáciou kontajneru bude načítanie súborov, kde kontajner bude vyhľadávať súbory, ktoré budu potrebné pri vykonávaní reflexie. Najprv sa pokúsi načítať vstupný súbor z vlastného súborového systému a následne otestuje, či existuje výstupný súbor, obsahujúci

už nejaké serializované objekty a v tomto prípade vyvolá obnovenie spracovávania vstupného súboru. Na ďalšiu prácu s reflexiou je potrebné pristúpiť k triede vo vstupnom súbore, čo zahrňuje aj samotné vyhľadávanie v tomto súbore. Následne môže dôjsť k vyvolaniu metód v tejto triede a to buď k zahájeniu výpočtu, alebo pozastaveniu výpočtu. Ďalej je potrebné aby kontajner dokázal získať zo vstupného súboru typ objektu, ktorý bude serializovaný a následne ho aj nastaviť, v prípade, že už existuje nejaký výstupný súbor.

Obr. 5.6: Diagram prípadu užitia kontajneru

5.3.1 Návrh životného cyklu kontajneru

Samotný životný cyklus kontajneru by nemal byť závislý od iných komponentov operačného systému Android, ako je napríklad `Activity`, alebo `Service`, ktoré majú vlastný životný cyklus. Programátor by mal mať možnosť ovplyvňovať stav kontajneru samostatne a implementovať ho podľa vlastného uváženia. Životný cyklus kontajneru začína v momente, kedy je vytvorená jeho nová inštancia. Ďalej musí programátor vyvolať operáciu `onContainerCreate()`, kedy dôjde k inicializácii potrebných objektov, s ktorými kontajner pracuje. Po počiatočnej inicializácii, môže programátor zahájiť výpočtovú činnosť kontajneru zavolaním operácie `onContainerResume()`. V prípade, že by programátor chcel reagovať na vstup užívateľa a zbytočne nezaťažovať výpočtové prostriedky hostiteľa kontajneru, môže zavolať operáciu `onContainerSuspend()`, kedy dôjde k uloženiu momentalného stavu objektu, nad ktorým prebieha výpočet. Ak dôjde k opätovnému uvoľneniu výpočtových prostriedkov, programátor môže znovu vyvolať operáciu `onContainerResume()`, ktorá zabezpečí načítanie stavu objektu, kedy došlo k prerušeniu. Ak dôjde k ukončeniu výpočtu, životný cyklus končí. Stavy, ktoré kontajner môže nadobudnúť sú uvedené na obrázku 5.7.



Obr. 5.7: Životný cyklus kontajneru

5.3.2 Diagram tried

Samostatný návrh kontajneru je možné najlepšie vyjadriť diagramom tried. Tento diagram nám zobrazuje všetky triedy, ktoré sú potrebné pre implementáciu kontajneru, ako aj ich atribúty, metódy a vzťahy medzi jednotlivými triedami. Diagram tried, na obrázku 5.8, obsahuje rozhranie `IAndroidContainer`, v ktorom je obsiahnuté správanie kontajneru. Toto správanie by mala trieda, ktorá ho implementuje, v našom prípade `AndroidContainer` vykonávať.

Trieda `AndroidContainer` obsahuje tieto **atribúty**:

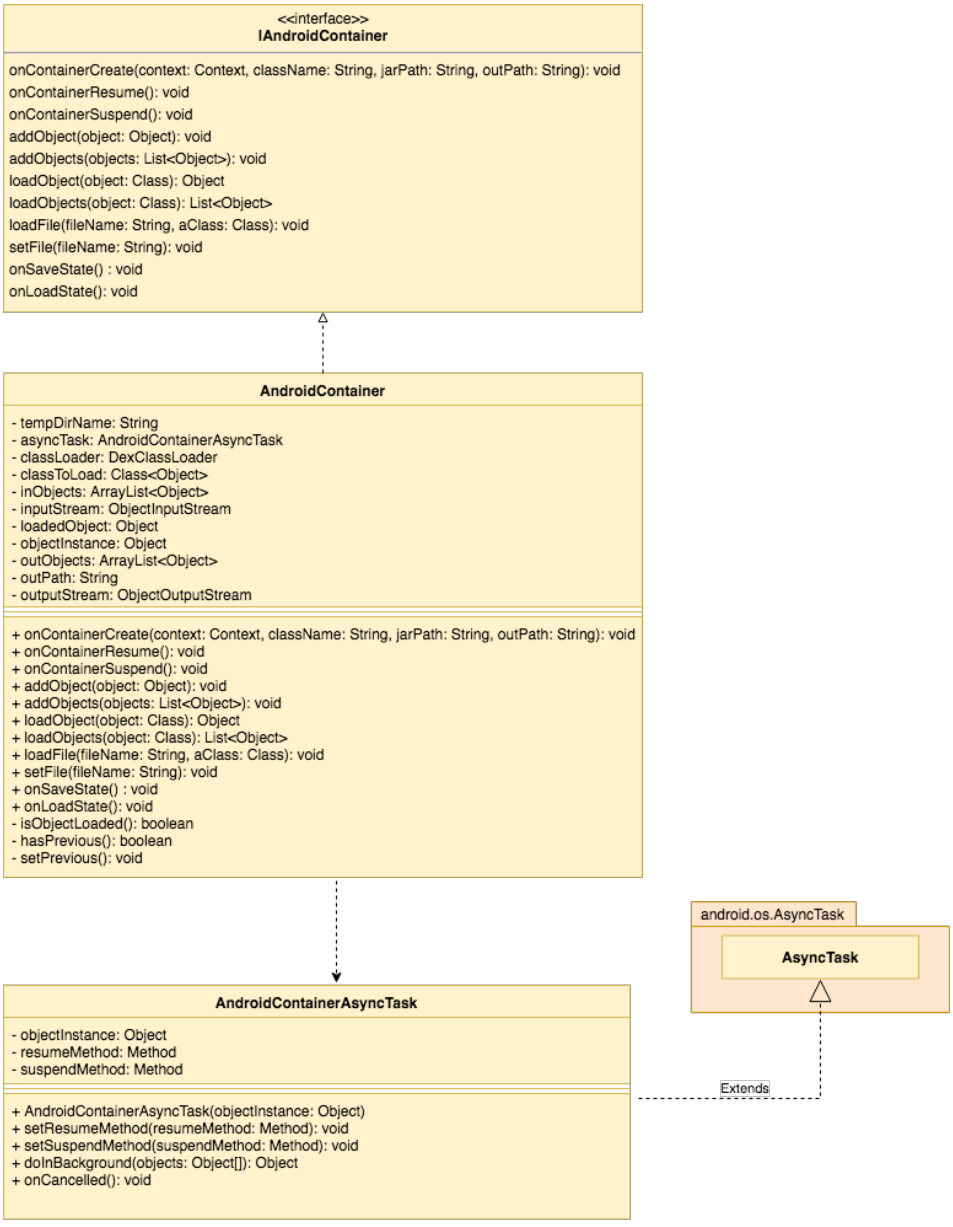
- `tempDir: String` - udáva názov dočasného optimalizovaného adresára, ktorý sa používa pri vytváraní objektu triedy `DexClassLoader` avšak od API verzie 26 je zastaralá a nepoužíva sa.
- `asyncTask: AndroidContainerAsyncTask` - atribút, ktorý drží referenciu na objekt triedy `AndroidContainerAsyncTask`, ktorá bude popísaná nižšie.
- `classLoader: DexClassLoader` - atribút potrebný pre prácu s reflexiou v systéme Android
- `classToLoad: Class<Object>` - trieda, ktorá ma byť pomocou reflexie načítaná

- `inObjects: ArrayList<Object>` - zoznam objektov, ktoré sú načítané z výstupného súboru
- `inputStream: ObjectInputStream` - vstupný tok, slúžiaci na čítanie objektov z výstupného súboru
- `loadedObject: Object` - serializovaný objekt, ktorý je definovaný v preskúmavanom vstupnom súbore
- `objectInstance: Object` - inštancia triedy, ktorá je preskúmavaná pomocou reflexie
- `outObjects: ArrayList<Object>` - zoznam objektov, ktoré budú zapísané do výstupného súboru
- `outPath: String` - cesta k výstupnému súboru
- `outputStream: ObjectOutputStream` - výstupný tok, slúžiaci na zápis do výstupného súboru

Ďalej obsahuje metódy, ktoré vykonávajú metódy definované v rozhraní `IAndroidContainer`. Týmito **metódami** sú:

- `onContainerCreate(context: Context, className: String, jarPath: String, outPath: String): void` - inicializácia kontajneru a potrebných atribútov
- `onContainerResume(): void` - metóda, ktorá ma za úlohu vypátrať, či už existuje nejaký výstupný súbor, kde sú serializované dáta pre daný výpočet, a samotný výpočet kontajneru, pomocou vyvolania objektu`AndroidAsyncTask`
- `onContainerSuspend(): void` - metóda na pozastavenie výpočtu objektu `AndroidAsyncTask` a uloženie objektov v serializovanej podobe do výstupného súboru
- `addObject(object: Object): void` - ukladanie objektu do atribútu obsahujúci zoznam objektov, ktoré budú zapísané do výstupného súboru
- `addObjects(objects: List<Object>): void` - metóda podobná metóde definovanej vyššie, avšak ukladá zoznam objektov
- `loadObject(object: Class): Object` - načítanie objektu špecifickej triedy, určenej parametrom
- `loadObjects(object: Class): List<Object>` - načítanie zoznamu objektov špecifickej triedy, určenej parametrom
- `loadFile(fileName: String, aClass: Class): void` - slúži na načítanie už existujúceho výstupného súboru
- `setFile(fileName: String): void` - nastavenie a inicializácia výstupného súboru a výstupného toku
- `onSaveState(): void` - metóda volaná pri zastavení výpočtu, kedy dochádza k zápisu objektov do výstupného súboru
- `onLoadState(): void` - metóda volaná pri obnovení výpočtu, kedy dochádza k načítaniu objektov z výstupného súboru

- `isObjectLoaded(): void` - kontrola, či bol objekt správne nahraný z preskúmavaného vstupného súboru
- `hasPrevious(): void` - kontrola, či existuje výstupný súbor, obsahujúci serializovaný objekt z predchádzajúceho výpočtu
- `setPrevious(): void` - metóda pre nastavenie objektu, ktorý má byť preskúmavaný pomocou reflexie



Obr. 5.8: Diagram tried kontajneru

Trieda `AndroidContainerAsyncTask`, ktorá dedí vlastnosti triedy `AsyncTask` definovanej v balíku, poskytovanom operačným systémom Android, ktorú využíva trieda zmienená vyššie, obsahuje tieto **atribúty**:

- `objectInstance: Object` - inštancia objektu, ktorý obsahuje metódy, ktoré budú vyvolané pomocou reflexie
- `resumeMethod: Method` - metóda, ktorá bude vyvolávaná pri zahájení výpočtovej činnosti kontajneru
- `suspendMethod: Method` - metóda, ktorá bude vyvolávaná pri zastavení výpočtovej činnosti kontajneru

Okrem konštruktoru, kde sa nastavuje atribút `objectInstance`, a operácii pre nastavenie ďalších atribútov obsahuje táto trieda metódy, ktorú prevažuje z nadradenej triedy. Týmito **metódami** sú:

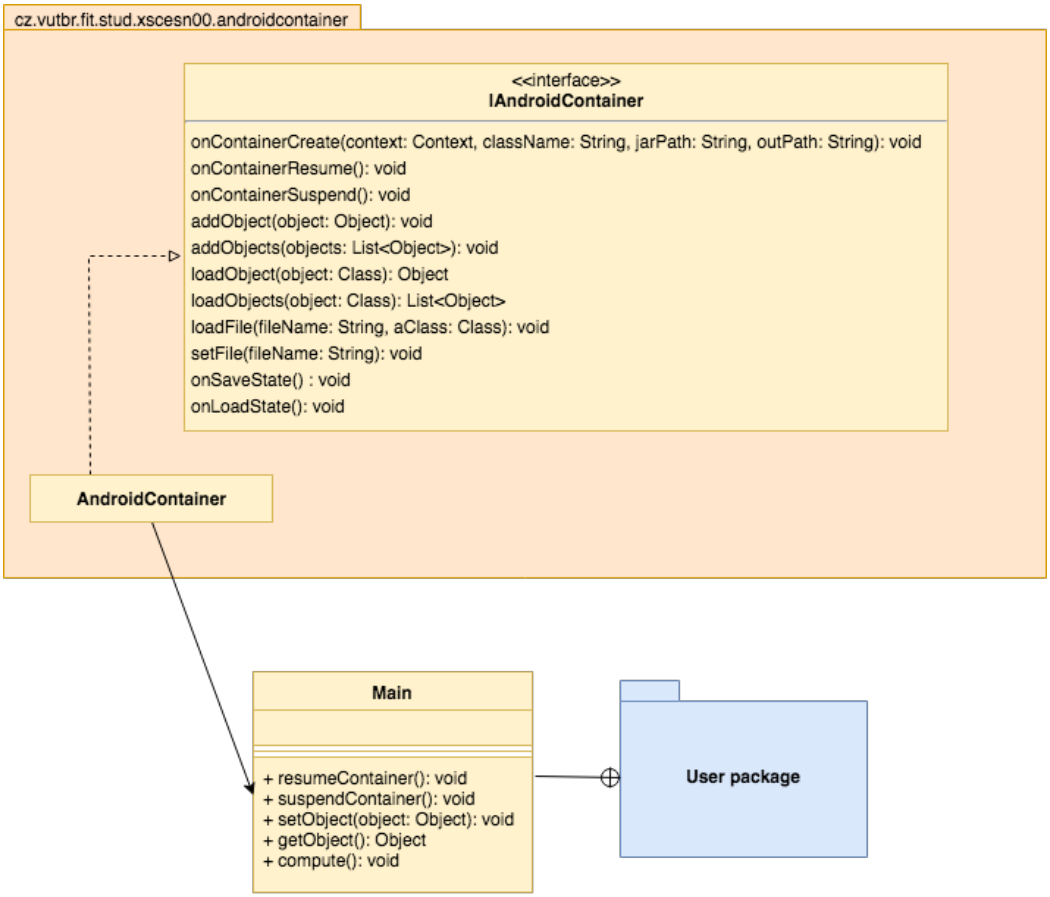
- `doInBackground(objects: List<Object>): Object` - metóda, ktorá ma za úlohu zahájenie výpočtovej činnosti kontajneru
- `onCancelled(): void` - metóda, ktorá ma za úlohu prerušiť výpočtovú činnosť kontajneru

5.4 Návrh prepojenia komponenty a kontajnera

Na rad prichádza otázka, ako dokáže programátor, ktorý chce využiť kontajner bežiaci na operačnom systéme Android, upraviť svoj kód tak, aby bol spustiteľný nami navrhovanom kontajneri. Aby bol schopný kontajner vykonať svoje operácie, ktoré pritom využívajú reflexiu, potrebuje, aby vyvolávaná trieda obsahovala metódy potrebné pre výpočet.

Programátor vo svojom balíčku, ktorý obsahuje vstupný program pre náš kontajner, má implementovanú triedu, ktorá je vstupom do tejto aplikácie, poprípade zaobaľuje objekty, ktoré sú používané k vykonaniu výpočtu. Túto situáciu si môžeme ukázať na grafe 5.9. V tomto momente je potrebné, aby programátor rozšíril svoju triedu o tieto **metódy**:

- `resumeContainer(): void` - v tejto metóde by mal programátor inicializovať objekt, v ktorom prebieha výpočet. Zároveň by sa v tejto metóde mala volať metóda `compute()` popísaná nižšie
- `suspendContainer(): void` - táto metóda sa volá pri prerušení, alebo zrušení výpočtu. Je vhodná na zrušenie závislostí, ktoré by mohli narušiť neukončený výpočet
- `setObject(object: Object): void` - metóda, kde je v parametroch predávaný objekt, ktorý by sa mal byť nastavený objektu, ktorý bol inicializovaný v metóde `resumeContainer()`. Táto metóda je dôležitá na nastavenie objektu do stavu, ktorý bol uložený pri prerušení výpočtu
- `getObject(): Object` - vrátenie objektu, v ktorom prebieha výpočet, tiež dôležitý na ukladanie stavu objektu
- `compute(): void` - metóda, ktorá je volaná pre zahájenie výpočtu programu



Obr. 5.9: UML diagram s potrebnými metódami

Kapitola 6

Implementácia

Táto kapitola je venovaná implementácii tejto práce, v súlade s návrhom riešenia. V úvodnej časti je opísaná implementácia a funkčnosť serveru, ktorý má spĺňať funkciu manažéra, ktorý distribuuje komponenty medzi zariadenia, ktoré obsahujú kontajner. V ďalšej časti je popísaná implementácia kontajneru, ktorý bol implementovaný ako knižnica, pre operačný systém Android. V poslednej časti je implementovaná ukážková komponenta, konkrétne triediaci algoritmus Bubble sort.

6.1 Implementácia serverovej časti aplikácie

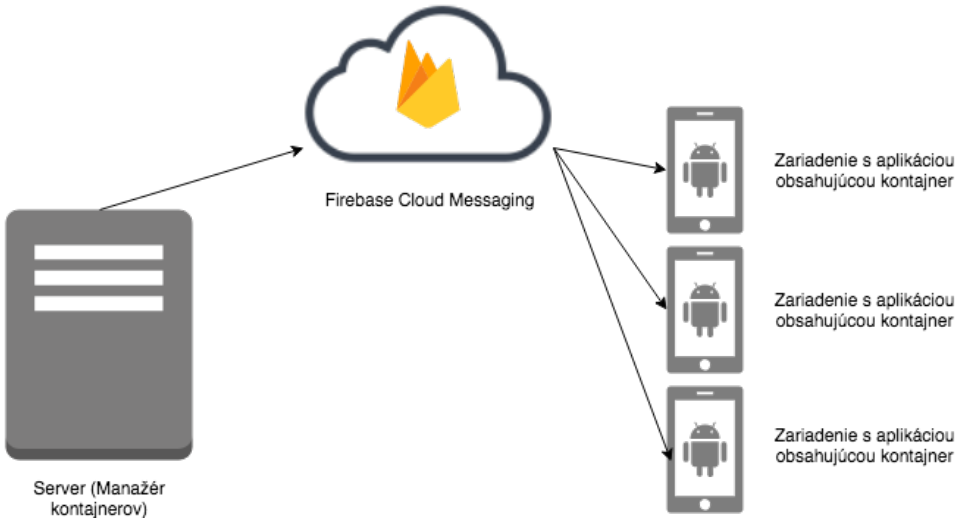
Ako už bolo spomenuté, úlohou servera je distribúcia komponent (vstupných súborov) na zariadenia, ktoré sú dostupné pre zahájenie výpočtu nad týmto súborom. Pre implementáciu primitívneho serveru, ktorý bude zvládať tieto jednoduché úlohy bol vybraný jazyk Python a framework Django s rozšírením Django REST framework, ktoré už boli spomenuté v predchádzajúcich kapitolách. Pri implementovaní serveru bol použitý architektonický vzor model-view-controller, známy aj ako MVC.

Dôležitou časťou serveru, je obojstranná komunikácia so zariadeniami v reálnom čase. Kedže sme limitovaný operačným systémom Android a je náročné dosiahnuť to, aby sa nami navrhnuté riešenie chovalo ako server a čakalo na pripojenie, bolo nutné zvoliť technológiu, ktorá bude rozposielať správy zo serveru na mobilné zariadenie. Po vybraní vhodnej technológie - **Firebase Cloud Messaging**, ktorá je spomínaná v kapitole použité technológie, sme problém so zasielaním správ vyriešili.

Celkové rozposielania správ je implementované tak, že po prijatí prvého súboru, odošle nami implementovaný server správu, do Firebase Cloud Messagingu, ktorý ďalej správu rozdistribuuje medzi zariadenia. Tento priebeh je ilustrovaný na obrázku 6.1.

6.1.1 Firebase Cloud Messaging Django

Pre implementáciu posielanie správ medzi serverom a Firebase Cloud Messaging, bol využitý balík `fcm-django` dostupný na [1]. Tento balík nám poskytuje zjednotenú platformu na odosielanie správ na mobilné zariadenia, či prehliadače. Obsahuje funkcionalitu, ktorá nám zabezpečí odosielanie správ, automatické odpájanie zariadení, ktoré nie sú aktívne, čo súvisí s tým, že na tieto zariadenia nie sú posielané žiadne správy. Tiež nám poskytuje modelovú triedu, ktorá nám rozširuje nami navrhnutý model, popisovaný v kapitole návrhu aplikácie. Tento systém dokáže odosielať **správy o upozornení** a **dátové správy**. My sme



Obr. 6.1: Princíp fungovania Firebase cloud messaging

sa rozhodli využiť dátové správy, v ktorých môžeme posielať rôzne dáta, v našom prípade ide o posielanie URL adresy vstupného súboru, a počtu opakovaní na zariadení.

Vyvolanie metódy na odosielanie správy je popísaný v **??**, kde je zobrazený kód (konkrétne vyvolanie signálu), ktorý sa po nahraní komponenty (vstupného súboru) spustí. Najprv pomocou balíku `fcm-django` dostaneme zoznam všetkých dostupných zariadení. Následne nad týmito zariadeniami vykoná metódu, kedy je odosielaná dátová správa, s odkazom na nahraný súbor a počtom opakovaní.

```
from fcm_django.models import FCMDevice

@receiver(post_save, sender=FileInput)
def file_input_post_save(sender, instance, **kwargs):
    devices = FCMDevice.objects.all()
    devices.send_message(data={"filename": instance.filename(), "loops":
        instance.loops})
```

Výpis 6.1: Odosielanie správ na dostupné zariadenia

6.1.2 Modelové triedy

Vďaka návrhu uvedeného v predchádzajúcej kapitole, vieme, že modelová trieda bude pozostávať z troch modelov, no za využitia balíčku spomenutého vyššie nám odpadá nutnosť implementovať triedu, ktorá sa stará o uchovávanie informácii o zariadeniach, no je potrebné počítať s ďalšími atribútmi, ako sú identifikačné číslo zariadenia, činnosť (aktívny/neaktívny) a typ operačného systému. Celkovo boli teda implementované len tieto dve modely:

- FileInput - model, ktorý bude obsahovať informácie o komponentoch (vstupných súboroch)

[1] https://github.com/xtrinch/fcm-django

- FileResponse - model, obsahujúci informácie o výstupných súboroch, ktoré uchovávajú serializované informácie o objektoch obsiahnutých v danej komponente

6.1.3 Databáza

Spolu s modelovými triedami úzko súvisí pojem databáza. Aby bolo možné pohodlne pracovať s informáciami uloženými na serveri, boli potrebné implementovať dátové úložisko. Existuje množstvo databáz, ktoré sa odlišujú v tom, ako je možné informácie ukladá. Pre jednoduchosť riešenia bol zvolený systém riadenia relačných databáz pomenovaný ako `SQLite`. Na rozdiel od mnohých iných systémov pre správu databáz, `SQLite` nie je databázový stroj klient-server, skôr býva implementovaný do koncového programu, ako je to aj v našom prípade. Je populárnou voľbou pre vstavaný databázový softvér pre ukladanie v aplikačnom softvéri, ako sú webové prehliadače, alebo vo vstavaných systémoch, ako sú napríklad mobilné telefóny.

Django ponúka možnosť šíriť zmeny v implementovaných modeloch do schémy databázy pomocou **migrácie**. Migrácia urobí z akejkoľvek zmeny modelov, alebo polí, novú modelovú položku, prípadne upraví stávajúcu položku, aby odpovedala návrhu v implementovanom modely. Je možné šíriť kód medzi viacerými verziami modelových tried, pretože migrácia nám zabezpečuje to, že modely budú stále v najnovšom stave a nedôjde k ich poškodeniu. Django sa pokúša napodobňovať `SQLite` vo:

- vytváraní novej tabuľky s novou schémou
- kopírovanie údajov naprieč databázou
- odstránenie starej tabuľky
- premenovanie novej tabuľky podľa originálneho názvu

Použitie tejto databázy nám zabezpečuje uloženie informácii o zariadeniach, vstupných súboroch a výstupných súboroch.

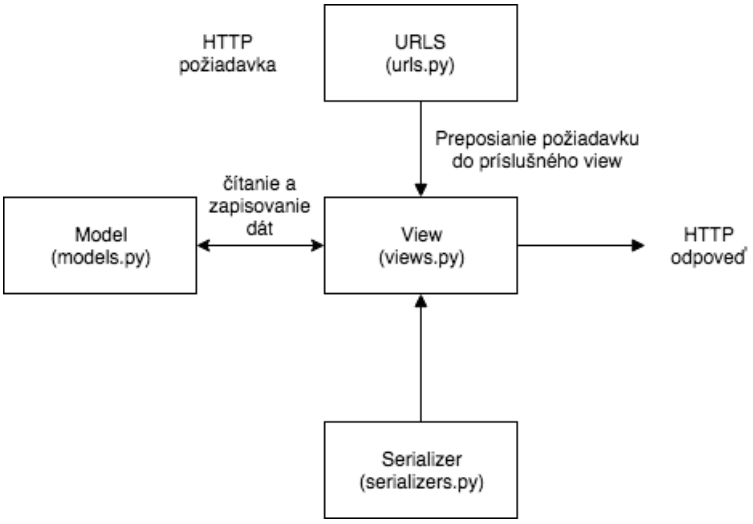
6.1.4 Spracovávanie požiadaviek

Aby systém vedel spracovávať požiadavky, ktoré na server prichádzajú, Django nám umožňuje navrhovať URL adresy, ktoré na server prichádzajú. Akým spôsobom Django postupuje pri prijatí požiadavky je zobrazený na obrázku 6.2 a jeho algoritmus pracuje následovne:

1. Django určí modul koreňového `URLconf`, ktorý sa má použiť. Táto hodnota je zvyčajne súčasťou nastavenia `settings.py` ako hodnota `ROOT_URLCONF`. Môže však dôjsť aj k situácii, že požiadavka obsahuje atribút `urlconf` a v tomto prípade sa použije táto hodnota.
2. Django načíta odpovedajúci modul a hľadá premennnú `urlpatterns` (inštancia `django.conf.urls.url()`.
3. Následne Django prechádza všetkými vzormi adresy URL (zapísané pomocou regulárných výrazov) v poradí akom sú napísané, a zastaví sa na prvom, ktorý zodpovedá požadovanej adrese URL.
4. Ak sa zhoduje jeden z výrazov definovaný regulárným výrazom, Django importue a vyvolá daný pohľad. Pohľad potom dostáva nasledujúce argumenty:

- inštancia objektu `HttpRequest`
- ak porovnávaný regulárny výraz nevráti žiadne pomenované skupiny, sú zhody z regulárneho výrazu poskytnuté ako argument udávajúci pozície
- argumenty kľúčových slov, ktoré sa skladajú z ľubovoľných menovaných skupín, ktoré zodpovedajú regulárnemu výrazu, a ktoré sú prepísané akýmikoľvek argumentami uvedenými vo voliteľnom argumente `kwarg`

5. Ak sa nenájde žiadny regulárny výraz odpovedajúci požiadavke, alebo dôjde k nejakej výnimke, Django zobrazí pohľad so zobrazenou hláškou.



Obr. 6.2: Spracovávanie požiadavky systémom Django

Pre implementáciu navrhovaných volaní, bol vytvorený súbor `urls.py`, ktorý vyzerá následovne a definuje spoločnú adresu, ktorá zaobaľuje všetky volania a to `api/`.

```
from django.conf.urls import url, include
from django.contrib import admin
from fcm_django.api.rest_framework import FCMDeviceViewSet
from rest_framework.routers import DefaultRouter
from django.conf import settings
from django.conf.urls.static import static

router = DefaultRouter()

urlpatterns = router.urls + [
    url(r'api/', include('container_manager.urls_api', namespace='api')),
]  + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Výpis 6.2: Príklad zavádzania URL endpointu

Kvôli stručnosti si ukážeme ako vyzerá prijatie požiadavku na získanie výstupného súboru. URL cesta ktorá definuje tieto volania je `file-response/` a registrácia tejto adresy bolo potrebné napísať následovné:

```
from django.conf.urls import url, include
```

```
from django.contrib import admin
from rest_framework.routers import DefaultRouter
from fcm_django.api.rest_framework import FCMDeviceViewSet
from file import views as file_v

router = DefaultRouter()
router.register(r'file-response', file_v.FileResponseViewSet,
    base_name='file-response')
```

Výpis 6.3: Získanie odpovedajúceho pohľadu

Ak príde teda požiadavok na adresu `file-input/` je vyvolaný pohľad `FileResponseViewSet` zo súboru `views.py`. V tomto súbore je trieda `FileResponseViewSet` implementovaná takto:

```
from django.shortcuts import render
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.parsers import FormParser, MultiPartParser
from rest_framework.viewsets import ModelViewSet
from file.models import FileResponse
from file.serializers import FileResponseSerializer
from rest_framework.generics import GenericAPIView, ListAPIView
from rest_framework import serializers, status
from fcm_django.models import FCMDevice

class FileResponseViewSet(ModelViewSet):

    queryset = FileResponse.objects.all()
    serializer_class = FileResponseSerializer
    parser_classes = (MultiPartParser, FormParser)

    def perform_create(self, serializer):
        serializer.save(datafile=self.request.data.get('datafile'))
```

Výpis 6.4: Príklad zavádzania URL endpointu

Tento pohľad pracuje s dátami uloženými v databáze, ktoré sú definované modelom `FileResponse` 6.5 a serializérom tohoto modelu `FileResponseSerializer` 6.6, ktorý má na starosti serializáciu, vďaka ktorej vieme prispôsobiť ukladanie a nahrávanie modelovej triedy z databázy.

```
from django.db import models
from fcm_django.models import FCMDevice

class FileResponse(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    datafile = models.FileField(upload_to=directory_response_path)
    duration = models.BigIntegerField(blank=True, null=True)
    time = models.DateTimeField(auto_now_add=True)
    file_input = models.ForeignKey(FileInput)
    device_id = models.ForeignKey(FCMDevice)

    def __str__(self):
        return self.datafile.name
```

Výpis 6.5: Modelová trieda `FileResponse`

```
from rest_framework import serializers
from file.models import FileInput, FileResponse
from fcm_django.models import FCMDevice

class FileResponseSerializer(serializers.HyperlinkedModelSerializer):
    device_id = serializers.SlugRelatedField(slug_field='registration_id',
        queryset=FCMDevice.objects.all())
    file_input = serializers.SlugRelatedField(slug_field='datafile',
        write_only=True, queryset=FileInput.objects.all())
    file_input_detail = serializers.HyperlinkedRelatedField(source='file_input',
        view_name='api:file-input-detail', read_only=True)
    file_input_detail_name = serializers.CharField(source='file_input.datafile',
        read_only=True)

    class Meta:
        model = FileResponse
        fields = ('id', 'created', 'datafile', 'duration', 'time', 'device_id',
            'file_input', 'file_input_detail', 'file_input_detail_name')
```

Výpis 6.6: Serializér triedy `FileResponse`

6.1.5 Ukladanie súborov

Medzi primárnu funkciu navrhovaného serveru, bola práca so súbormi. Django ponúka možnosť ukladania súborov do databázy, alebo do súborového systému. Pri implementácii nášho serveru sme sa rozhodli súbory ukladať do súborového systému a to konkrétne do priečinku `media/files/`.

6.2 Implementácia kontajneru

Hlavnou úlohou kontajneru, je preskúmavanie komponenty (vstupného súboru) pomocou reflexie, no popri tom nesmieme zabúdať na prácu so serializovanými objektmi, ktoré sú uložené vo výstupnom súbore. Na implementáciu bol vybraný jazyk Java a kontajner bol implementovaný ako open-source knižnica, pre voľnú distribúciu a použitie. Knižnicu je možné spúšťať na zariadeniach s verziou operačného systému Android 4.4 (API 19) a vyššie, čo je v čase písania tejto práce asi 90,1%[1]1 funkčných zariadení. Knižnica nepoužíva žiadne ďalšie knižnice tretej strany a je implementovaná využívajúc základné stavebné prvky systému Android.

6.2.1 Serializácia objektov

Kedže podstata celého kontajneru je v pokračovaní a prerušení výpočtu, je dôležité, aby dokázal pracovať so súbormi, kde sú uložené serializované objekty. Pre ukladanie samotných serializovaných objektov sa využíva trieda `ObjectOutputStream`, ktorý je súčasťou balíka `java.io.ObjectOutputStream` obsiahnutého v operačnom systéme Android. `ObjectOutputStream`

[1] https://developer.android.com/about/dashboards/

sa používa na zápis primitívnych dátových typov a objektov, ktoré podporujú rozhranie `java.io.Serializable`. Trieda každého serializovateľného objektu je zakódovaná vrátane názvu a signatúry triedy, jeho hodnôt polí a políčok objektov a taktiež pre akékoľvek iné objekty, ktoré odkazujú na iné. Nezapisuje však políčka označené ako `transient` (prechodné) a `static` (statické).

Pre ukladanie serializovaných objektov do súboru bola implementovaná metóda `setFile(String filename)` 6.7, kde je na zápis do súboru udávaný parametrom využitý `FileOutputStream`.

```
@Override
  public void setFile(String file) {
    try {
      outputStream = new ObjectOutputStream(new FileOutputStream(file));
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

Výpis 6.7: Metóda pre vytvorenie ObjectOutputStream

Naopak pre obnovenie serializovaných objektov sa používa objekt `ObjectInputStream` (súčasť balíka `java.io.ObjectInputStream`, ktorý deserializuje primitívne dáta, alebo objekty, ktoré podporujú vyššie spomínané rozhranie `java.io.Serializable`. Predvolený mechanizmus deserializácie objektov obnovuje obsah každého políčka na hodnotu a typ, ktorý mal pri zápise. Políčka deklarované ako `transient` (prechodné) a `static` (statické) sú ignorované. Referencie na iné objekty spôsobujú, že sa tieto objekty predčítavajú zo súboru podľa potreby. Je dôležité spomenúť, že pri deserializácii sa vždy prideľujú nové objekty, čo zabraňuje prepísaniu existujúcich objektov. Trieda `ObjectInputStream` zabezpečuje, že typy všetkých načítavaných objektov sa zhodujú s triedami prítomnými v Java Virtual Machine a triedy sa načítajú podľa potreby a to pomocou štandardných mechanizmov. Tu sme ale dostali do štádia, kedy sme museli implementovať vlastný mechanizmus načítavania triedy využivajúci reflexiu. Metóda, ktorá implementuje tento mechanizmus je popísaný vo výpise **??** a je implementovaná v triede `CustomObjectInputStream`, ktorá dedi triedu `ObjectInputStream` a prepisuje jej metódu `resolveClass()`.

```
class CustomObjectInputStream extends ObjectInputStream {

    @Override
    protected Class<?> resolveClass(ObjectStreamClass desc)
      throws IOException, ClassNotFoundException {
      if (desc.getName().equals(className.getName())) {
        return classLoader.loadClass(className.getName());
      }
      return super.resolveClass(desc);
    }
  }
```

Výpis 6.8: Mechanizmus na načítavania triedy

Implementácia načítavania serializovaných dát a ich deserializácia zo súboru je implementovaná podobne ako v prípade `ObjectOutputStream`.

6.2.2 Životný cyklus

Podľa návrhu ma mať životný cyklus kontajneru tri stavy. Tieto stavy sú implementované v troch metódach, kde každá metóda značí, kde sa momentálne kontajner nachádza.

Metóda `onContainerCreate()`, označuje stav, kedy dochádza k vytvoreniu kontajneru. V tomto stave sú inicializované potrebné objekty, ktoré sú potrebné pre prehľadávanie vstupného súboru pomocou reflexie. Objekt `DexClassLoader`, ktorý je súčasťou balíčka `dalvik.system.DexClassLoader`, načíta triedy zo súborov `.jar` a `.apk`, ktoré obsahujú položku `classes.dex`. Tieto triedy nie sú súčasťou nainštalovanej aplikácie a budú využité na vykonanie kódu. Pomocou `DexClassLoader` sa pomocou reflexie vyhľadáva trieda, ktorá má byť použitá pri výpočte a ukladá sa aj jej nová inštancia. Ďalej sa v tomto stave ukladajú cesty k súborom, s ktorými sa bude pracovať a tiež sa inicializuje objekt `AndroidContainerAsyncTask`, ktorý bude popísaný nižšie.

Ďalšou metódou, ktorá označuje stav, zahájenia činnosti, je metóda `onContainerResume()`. V tejto metóde dochádza k vyhľadávaniu už existujúcich serializovaných objektov vo výstupnom súbore a ich načítaniu. Ak sa tam tento súbor, alebo objekty nenachádzajú, kontajner pracuje ďalej s objektmi, ktoré su nanovo inicializované. V tomto stave sa tiež získava metóda, ktorá musí byť implementovaná v komponente (vstupnom súbore) a to metóda `resumeContainer()`. Táto metóda je ďalej predávana parametrom `AndroidContainerAsyncTask` a zahajuje výpočtovú činnosť kontajneru.

Poslednou metódou označujúca stav kontajneru je metóda `onContainerSuspend()`. V tejto metóde dochádza k získaniu metódy `suspendContainer()`, ktorá je implementovaná v komponente a môže obsahovať rôzne operácie pre ukončenie výpočtu. Táto metóda je tiež predávana `AndroidContainerAsyncTask` a po vyvolaní tejto metódy, je činnosť kontajneru pozastavená.

6.2.3 Výpočet

Zahájenie výpočtu kontajneru má na starosti trieda `AndroidContainerAsyncTask`, ktorá dedí chovanie triedy `AsyncTask`, ktorá je súčasťou systému Android a nachádza sa v balíku `android.os.AsyncTask`. Táto trieda nám umožňuje vykonávať operácie na pozadí, bez toho, aby užívateľovi zabraňovala prevzatia vlákna, kde beží používateľské rozhranie. Asynchronná úloha je definovaná výpočtom, ktorý beží na pozadí, a ktorého výsledok je publikovaný na vlákne používateľského rozhrania.
Pri implementácii triedy `AndroidContainerAsyncTask`, nebola implementovaná možnosť publikovať výsledok do použivateľskeho rozhrania, a preto boli preťažené a prepísané metódy `doInBackground()` a `onCancelled()`.

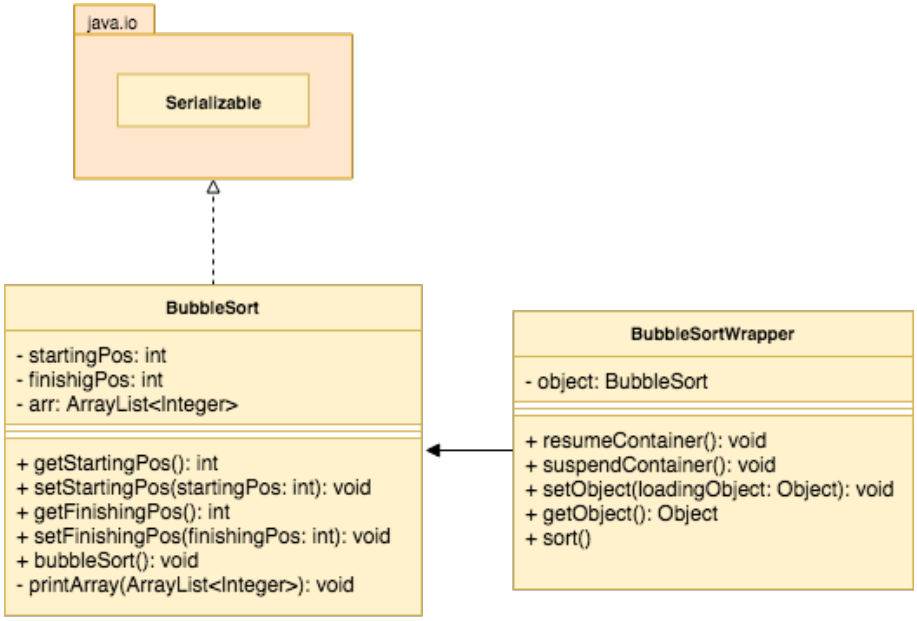
Metóda `doInBackground()` za užitia reflexie cyklicky vyvoláva metódu, ktorá bola tejto triede predaná, do doby, keď príde signál, ktorý túto prácu preruší. Pre demonštratívne účely bola do tejto metódy implementovaná aj možnosť uspať vlákno a to na dobu jednej sekundy.

`onCancelled()` je metóda, ktorá preruší činnosť tejto triedy a ignoruje jej výsledok. V tejto metóde sa podobne ako v predchádzajúcom prípade vyvoláva metóda predaná do tejto triedy pomocou reflexie.

6.3 Implementácia ukážkovej komponenty

Súčasťou zadania bola aj implementácia ukážkovej komponenty. Za ukážkovú komponentu bol zvolený triediaci algoritmus, a to algoritmus Bubble sort. Zložitosť tohto algoritmu je $\mathcal{O}(n^2)$, kde `n` je počet prvkov, ktoré majú byť zoradené.

Táto komponenta je rozdelená do dvoch tried a to trieda `BubbleSortWrapper` a `BubbleSort`, ktorá implementuje rozhranie `java.io.Serializable`. Kompletný zoznam ich atribútov a metód môžeme nájsť na diagrame tried, ktorý je na obrázku 6.3.



Obr. 6.3: Diagram tried ukážkovej komponenty

Trieda `BubbleSort` je trieda, ktorá obsahuje neusporiadaný zoznam položiek typu `int`, ktoré majú byť po vykonaní algoritmu usporiadané vzostupne od najmenšieho po najväčší. Aby bolo možné tento algoritmus spúšťať a pozastavovať, bolo nutné upraviť algoritmus pre zoradenie týchto prvkov. Tento algoritmus je poupravený tak, že za každou iteráciou, kedy kontajner na zariadení riadi výpočet tejto komponenty, je hodnota poľa zväčšovaná o dva. Táto úprava nám zaručí, že simulácia tohto algoritmu sa môže v určitom čase pozastaviť, uložiť momentálny stav a prípadne pokračovať vo výpočte ďalej, až kým nebudú všetky prvky zoradené. Na túto činnosť nám slúžia atribúty `startingPos` a `finishingPos` určujúci počiatočnú, respektíve konečnú pozíciu vo vykonávanom cykle.

Zaobalenie tejto triedy a umožnenie prístupu k výpočtu tejto triedy kontajneru, ktorý je implementovaný v zariadení, má na starosti trieda `BubbleSortWrapper`. Táto trieda obsahuje referenciu na objekt typu `BubbleSort`, a tiež metódy, ktoré kontajner potrebuje pre svoju činnosť. V metóde `resumeContainer()` dochádza k inicializácii objektu, a to len v prípade, že objekt nie je inicializovaný a zahájenie triediaceho algoritmu. Metóda `suspendContainer()` je v ukážkovej komponente prázdna, nakoľko nevyužívame žiadne zdroje, ktoré by sme potrebovali počas prerušenia, či zastavenia výpočtu uvoľniť. Metódy `setObject()` a `getObject()` sú volané kontajnerom pre ukladanie, respektíve nastavenie serializovaného objektu. Metóda `sort()` má na starosti úpravu tohto algoritmu spomínanú vyššie, vďaka ktorej môžeme algoritmus prerušiť a obnoviť.

Výstupom tejto komponenty je `.jar` súbor, ktorý však na preskumávanie pomocou reflexie musíme previesť do formátu `.dex`. Ide o binárny formát súbor, generovaný z Java `.class` súborov `Dex kompilátorom`. Tento kompilátor prekladá Java virtual machine bytecode na Dalvik virtual machine bytecode a dá všetky súbory triedy do jedného súboru `.dex`.

Na prevedenie do odpovedajúceho formátu sme využili nástroj, ktorý je dostupný po nainštalovaní Android SDK do nášho operačného systému. Tento nástroj sa väčšinou nachádza v `/Android/sdk/build-tools/{verzia}/`. Príkaz, ktorý spustí konverziu je uvedený vo výpise 6.9, a očakáva argument súboru, ktorý ma byť konvertovaný na `.dex`

```
./dx --dex --output="/BubbleSort/BubbleSort.dex"
           "/BubbleSort/out/artifacts/BubbleSort/BubbleSort.jar"
```

Výpis 6.9: Príkaz na konvertovanie súboru do dex formátu

6.4 Implementácia vzorovej Android aplikácie

Vzorová aplikácia bola pomenovaná ako **Container Demo** a má prezentovať príklad, ako je možné pracovať s vyššie implementovaným serverom, kontajnerom a ukážkovou komponentou. Túto aplikáciu je podobne ako aj knižnicu obsahujúci kontajner spúšťať na mobilných zariadeniach s verziou operačného systému Android 4.4 (API 19) a vyššie. Aplikácia obsahuje len jednu obrazovku, ktorá je akýmsi pomyselným vstupom do aplikácie, no cieľom tejto práce nebola implementácia aplikácie s používateľským rozhraním. Táto aplikácia vykonáva výpočet riadený kontajnerom na pozadí a neprodukuje žiadne výstupy do užívateľského prostredia. Namiesto toho sa využíva dlhotrvajúca operácia v službe Android, ktorá je aktívna na pozadí. Detailnejšiu štruktúru aplikácie je možné vidieť v diagrame tried na obrázku 6.4.

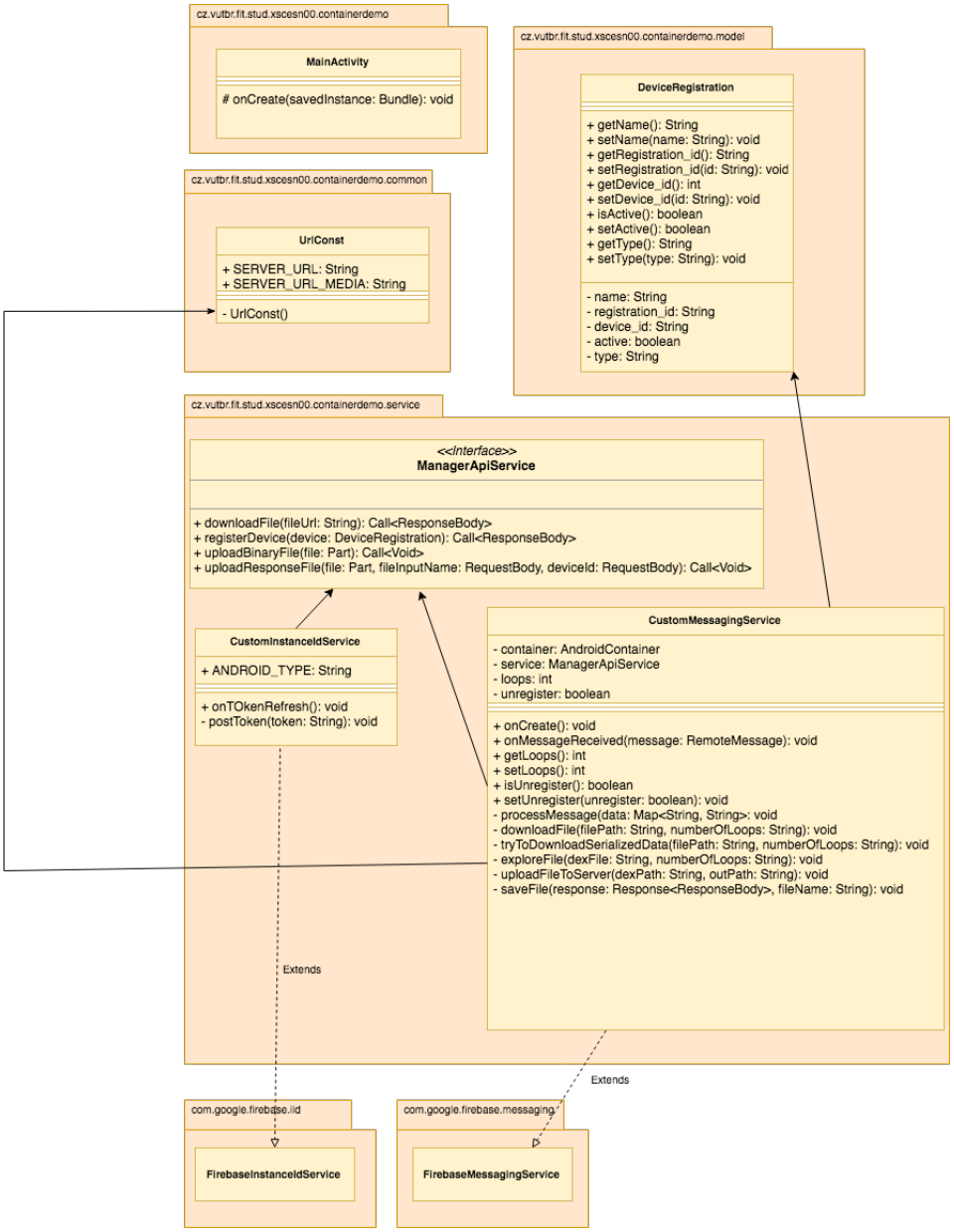
6.4.1 Registrácia zariadenia na server

Po prvom spustení aplikácie, sa vďaka využívaniu knižníc pre prácu s **Cloud Messaging** vygeneruje unikátne identifikačné číslo zariadenia (token), ktorý sa odošle na server, ktorého adresa je uvedená v triede `UrlConst`. K zmene tokenu môže dôjsť v týchto situáciach:

- aplikácia vymaže inštanciu ID
- aplikácia je obnovená na novom zariadení
- používateľ odinštaluje / znova nainštaluje aplikáciu
- používateľ vymaže údaje aplikácie.

Na posielanie požiadavkov na server sa využíva knižnica tretej strany - **Retrofit**[1]. Retrofit je knižnica, ktorá poskytuje klienta REST pre Android. To nám umožňuje relatívne ľahko načítať a nahrávať dáta vo formáte JSON (alebo iné štruktúrované dáta) cez webovú službu založenú na REST. Retrofit používa `OkHttp` v balíku `com.android.okhttp` pre HTTP požiadavky.

[1] http://square.github.io/retrofit/



Obr. 6.4: Diagram tried ukážkovej aplikácie

6.4.2 FirebaseMessagingService

Ide o základnú triedu pre príjem správ z **Firebase Cloud Messaging**, ktorá dedí z triedy `Service`. Táto trieda nám poskytuje funkcie na automatické zobrazovanie upozornení, no my upozornenia zobrazovať nechceme, namiesto toho užívateľ nebude vedieť o tom, že na jeho zariadení prebieha nejaký výpočet.

Vo vzorovej aplikácii, tvorí rozšírenie tejto triedy základný stavebný blok pre spustenie a prácu kontajneru. Pomocou zavedenia tejto služby na pozadí, čaká aplikácia na prijatie správy od serveru na zahájenie výpočtu. Po obdržaní dátovej správy, ktorá obsahuje URL adresu komponenty a číslom, ktoré udáva počet opakovaní sa začne sťahovanie súborov.

V prípade, že od serveru nepríde žiadna správa, táto služba čaká na správu a nenarušuje užívateľovi chod zariadenia.

6.4.3 Sťahovanie a nahrávanie súborov

Na sťahovanie a nahrávanie súborov sa taktiež používa vyššie zmienená knižnica `Retrofit`. Po tom čo aplikácia obdrží URL, pošle aplikácia na túto URL žiadosť na stiahnutie komponenty. Ak prebehne sťahovanie v poriadku, pošle aplikácia požiadavku na stiahnutie výstupného súboru, ktorý obsahuje serializované dáta. Ak je odpoveď kladná, súbor sa stiahne a objekt s ktorým sa pracuje sa nastaví do stavu, v ktorom je uložený. Inak zaháji kontajner svoju prácu bez nastavovania stavu objektu.

Ďalším krokom je implementácia `BroadcastReceiver`, ktorý bude zabezpečuje chod kontajneru.

6.4.4 BroadcastReceiver

Aplikácie v systéme Android môžu odosielať, alebo príjmať správy vysielané zo systému Android a iných aplikácii určených pre tento systém, podobne ako návrhový vzor **publish-subscribe**. Správy sa vysielajú v prípade, že nastane udalosť ktorá nás zaujíma. Napríklad od systému Android môžeme vyžadovať poslanie správy, v prípade výskytu rôznych systémových udalostí, napríklad keď sa zariadenie začne nabíjať. Tiež je možné, aby aplikácie produkovali vlastné správy, napríklad na upozornenie ostatných aplikácii, na niečo čo by ich mohlo zaujímať.

Aplikácie sa môžu zaregistrovať, pre prijímanie konkrétnych vysielaní a po odoslaní systém automaticky smeruje vysielanie do aplikácii. Vzorová aplikácia sa registruje na prijatie správy v momente, ak dôjde k vypnutiu, respektíve zapnutiu obrazovky. Registrácia na túto udalosť je zobrazená vo výpise 6.10, kde sa typ správ, ktoré chceme dostávať nastavuje na riadku dva a v tomto prípade, chceme byť oboznámený zo situáciou, kedy dôjde k vypnutiu obrazovky.

```
IntentFilter filter = new IntentFilter();
    filter.addAction(Intent.ACTION_SCREEN_OFF);
    registerReceiver(new BroadcastReceiver() {
      @Override
      public void onReceive(Context context, Intent intent) {

      }
    }, filter);
```

Výpis 6.10: Registrácia na udalosť vypnutia obrazovky

Táto skutočnosť bola využitá na demonštráciu spustenia, výpočtu a prerušenia kontajneru. V prípade, že sa obrazovka vypne, kontajner zaháji výpočet a prebieha až do doby, kým sa obrazovka znovu nezapne.

Kapitola 7

Záver

Cieľom tejto diplomovej práce bolo vytvorenie kontajneru pre komponenty bežiace na operačnom systéme Android, teda rozhranie komponenta-kontajner, spôsob prevádzky a distribúcie komponentov, životný cyklus kontajneru a možnosť tvorby adaptérov pre iné typy komponentov.

Ako prvé bolo potrebné naštudovať si operačný systém Android, jeho základné stavebné bloky a spôsoby implementácie aplikácii bežiacich na tejto platforme. Následne bolo potrebné štúdium reflexie a serializácie objektov na systéme Java virtual machine a možné rozdiely na systéme Dalvik virtual machine.

Štúdium existujúcich riešení nám poskytlo slušný základ k navrhnutiu spôsobu distribúcie komponentov a komunikácie medzi zariadeniami, čoho výsledkom je kapitola venovaná riešeniu tohoto problému.

Zo získaných znalostí a návrhu, ktorý bol viackrát upravovaný do konečnej podoby, je výsledkom knižnica Android Container, ktorú je možno importovať do aplikácii písaných v systéme Android. Spolu s knižnicou bol implementovaný aj primitívny server, ktorý ma za úlohu delegovať prácu medzi zariadeniami. Pre demonštráciu spolupráce tejto knižnice a tohto serveru, bola implementovaná vzorová aplikácia, ktorá simuluje výpočet ukážkovej komponenty, ktorou je triediaci algoritmus Bubble sort.

Táto knižnica poskytuje slušný základ na vytváranie aplikácii, ktoré budú pracovať s komponentami bežiacimi na systéme Android, no do budúcnosti by bolo dobré navrhnúť a implementovať mechanizmus na určenie ukončenia výpočtu, čo v tejto práci nebolo zrealizované. Taktiež by bolo vhodné implementovať mechanizmus na zapisovanie záznamov a ich zobrazenie užívateľovi. V prípade serveru, ktorý je zodpovedný za spravovanie jednotlivých komponentov by bolo vhodné pre koncového užívateľa rozšíriť funkcionalitu o zobrazenie viacerých štatistík pre budúcu analýzu.

Literatúra

[1] *Android developer guide*. [Online; navštíveno 15.03.2018].
URL https://developer.android.com/guide/index.html

[2] Django Framework. [Online; navštíveno 22.5.2018].
URL https://www.djangoproject.com/

[3] Fielding, R. T.; Taylor, R. N.: Principled Design of the Modern Web Architecture. *ACM Trans. Internet Technol.*, ročník 2, 2002: s. 115–150, ISSN 1533-5399.

[4] Forman, I. R.; Forman, N.: *Java Reflection in Action (In Action Series)*. Greenwich, CT, USA: Manning Publications Co., 2004, ISBN 1932394184.

[5] Genčúr, M.: *Rámec pro dynamickou aktualizaci aplikací v jazyce Java*. [Online; navštíveno 15.03.2018].
URL http://www.fit.vutbr.cz/study/DP/DP.php?id=6955&file=t

[6] Kozák, D.: *Srovnání výkonu a vlastností objektově orientovaných databází*. [Online; navštíveno 23.04.2018].
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=14162&file=t

[7] Mokrá, R.: *Návrh a implementace nástroje pro plánovaní projektů a času s využitím principů z oblasti plánovaní úloh reálného času*. [Online; navštíveno 21.05.2018].
URL http://www.fit.vutbr.cz/study/DP/DP.php?id=14175&file=t

[8] Rogers, R.; Lombardo, J.; Mednieks, Z.; aj.: *Android Application Development: Programming with the Google SDK*. O'Reilly Media, Inc., první vydání, 2009, ISBN 0596521472, 9780596521479.