



**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 SOFTWAREOVÉ KOM-  
PONENTY BEŽÍCÍ NA OS ANDROID**

A CONTAINER FOR MIGRATING SOFTWARE COMPONENTS RUNNING ON ANDROID OS

**DIPLOMOVÁ PRÁCE**

MASTER'S THESIS

**AUTOR PRÁCE**

AUTHOR

**Bc. VLADIMÍR ŠČEŠŇÁK**

**VEDOUcí PRÁCE**

SUPERVISOR

**RNDr. MAREK RYCHLÝ, Ph.D.**

**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.

## Klíčové 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 běžící 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

21. mája 2018

## Podakovanie

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

# Obsah

<b>1</b>	<b>Úvod</b>	<b>2</b>
<b>2</b>	<b>Operačný systém Android</b>	<b>3</b>
2.1	Vrstvy operačného systému Android . . . . .	3
2.2	Aktivity a ich životný cyklus . . . . .	4
2.3	Služby (dlhotrvajúce operácie) v systéme Android. . . . .	6
2.4	Zabezpečenie . . . . .	8
2.5	Prístup k zdrojom . . . . .	10
<b>3</b>	<b>Použité technológie</b>	<b>13</b>
3.1	Serializácia . . . . .	13
3.2	Reflexia . . . . .	13
3.3	Django a REST API . . . . .	16
3.4	Firebase Cloud Messaging . . . . .	16
<b>4</b>	<b>Existujúce riešenia</b>	<b>17</b>
<b>5</b>	<b>Návrh aplikácie</b>	<b>18</b>
5.1	Návrh architektúry aplikácie . . . . .	18
5.2	Návrh serverovej časti aplikácie . . . . .	19
5.3	Návrh kontajneru . . . . .	21
5.4	Návrh prepojenia komponenty a kontajnera . . . . .	26
<b>6</b>	<b>Implementácia</b>	<b>28</b>
6.1	Implementácia serverovej časti aplikácie . . . . .	28
6.2	Implementácia kontajneru . . . . .	33
6.3	Implementácia ukázkovej komponenty . . . . .	33
6.4	Implementácia vzorovej Andorid aplikácie . . . . .	33
<b>7</b>	<b>Možnosti rozšírenia kontajneru</b>	<b>34</b>
<b>8</b>	<b>Záver</b>	<b>35</b>
	<b>Literatúra</b>	<b>36</b>

# 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, pozostávajúce 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 [5] v priebehu pár mesiacov si stiahlo prvú beta verziu zo stránky Googlu niekoľko miliónov ľudí a v roku 2008 bol v Spojených štátoch amerických predstavený prvý mobilný telefón bežiaci na tejto platforme.

Cielom 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 rozšírila 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 výrobcovi zariadení ľahko vyvinúť ovládače hardvéru.

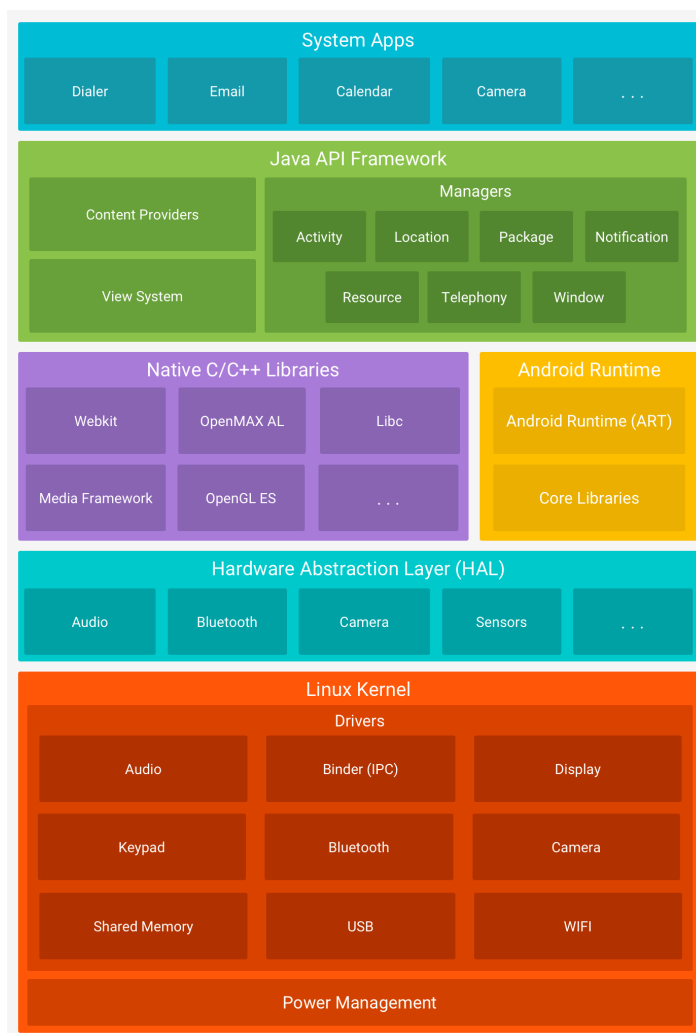
Ďalšou vrstvou je **hardvérová abstrakčná vrstva**, ktorá pozostáva z viacerých modulov a poskytuje štandardné rozhrania, ktoré vystavujú schopnosti hardvéru zariadenia do vyššej úrovne aplikačného rámca Java API. Úlohou vrstvy, ktorá je označená ako **Android Runtime** je, že každá aplikácia beží vo 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 aplikácií.

**Java API Framework** obsahuje základné stavebné bloky, písané v jazyku Java, ktoré potrebuje programátor pri vytváraní aplikácií. Existujú štyri rôzne typy komponentov aplikácie a to Aktivity, Služby, Prijímače a 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 aplikácií, ktoré sú dodávané s každým Androidom. Ide hlavne o aplikácie ako telefonovanie, vytváranie textových správ, kamera, kalendár apodobne.



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 užívateľom a aplikáciou. Samotná aktivita poskytuje okno, kde aplikácia vykresľuje svoje užívateľské rozhranie, ktoré môže vyplniť celú obrazovku, môže byť menšie ako celá obrazovka, alebo dokonca môže sa 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` volaním 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.

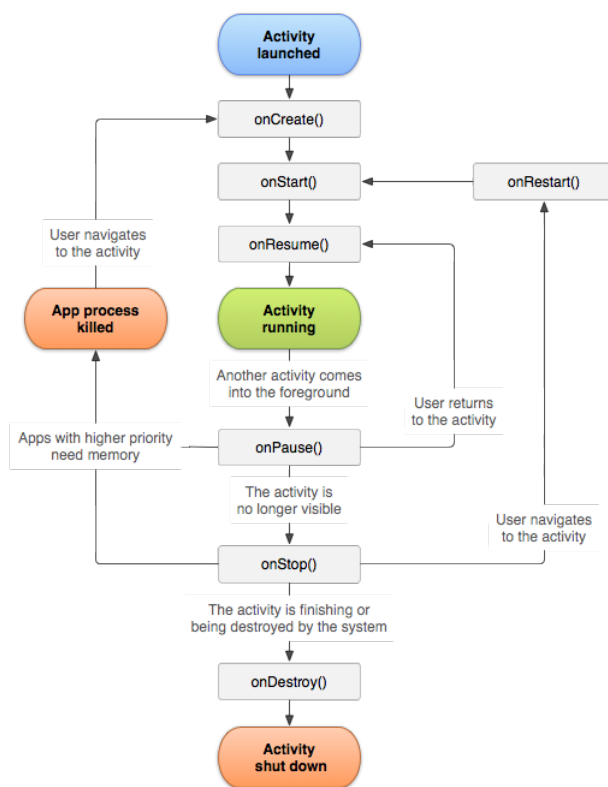
## Životný cyklus aktivity a jej spätné volania

- **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 prvýkrát, 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 udržiava 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 dispeleji 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 prejdenu 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 animácií, 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 aktivite, 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 zavolať 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

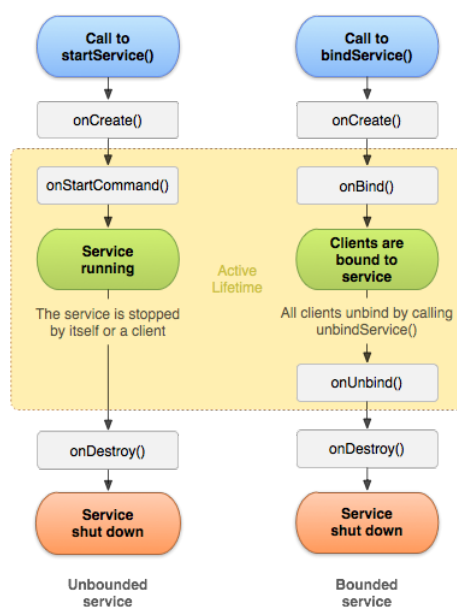
## 2.3 Služby (dlhotrvajúce operácie) v systéme Android.

Služba je komponenta aplikácie, ktorá má na starosti vykonávanie dlhodobých operácií 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 sú indikované 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 komponentám komunikovať so službou, posilať 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 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 zastaviť 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 rozhoranie `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 sa 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()`.

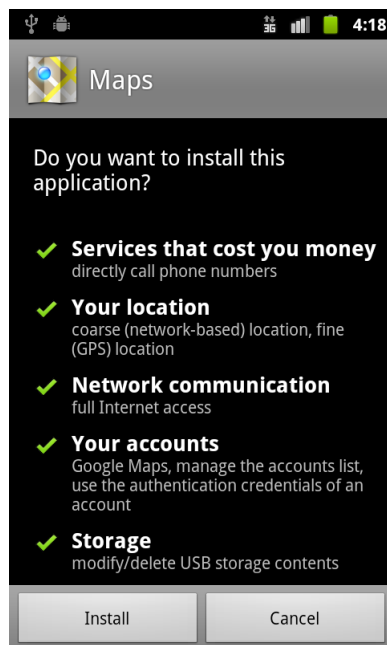
## 2.4 Zabezpečenie

Systém Android obsahuje prvotriedne bezpečnostné funkcie a spolupracuje s vývojarmi 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, aplikácií, 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 konfigurácií a hoci je procesorovo-agnostický (beží na rôznych procesoroch rovnako), využíva výhody určitých hardvérových bezpečnostných funkcií, ako je napríklad `ARM eXecute-Never`<sup>1</sup>. 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 v rámci aplikačného sandboxu. Aplikácie tak získavajú osobitnú časť súborového systému, v ktorom môžu zapisovať súkromné údaje, vrátane databáz a nespracovaných súborov.

<sup>1</sup><http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360f/CACHFICI.html>



Obr. 2.4: Príklad oprávnení, ktoré aplikácia požaduje

## Zabezpečenie na úrovni jadra

Základom pre mobilné počítačové prostredie je jadro linuxu s operačným systémom Android s niekoľkými kľúčovými bezpečnostnými funkciami, ktorými sú užívateľsky založený model oprávnení, izolovanie procesov, rozšírený mechanizmus pre bezpečnú medziprocesorovú komunikáciu a schopnosť odstrániť nepotrebné a potenciálne neisté časti jadra. Základným bezpečnostným cieľom jadra linuxu je navzájom izolovať medzi sebou zdroje používateľov. Linux tak zabráňuje 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).

## Aplikačný sandbox

Systém Android priradzuje 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 aplikáciám. 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.

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, bežia v aplikačnom sandboxe. V systéme Android neexistujú žiadne obmedzenia, ako písať aplikáciu, tak aby bola docieľená 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. Takto 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.

## 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 aplikáciám používať úložisko systémových poverení pre súkromné kľúče a refazce certifikátov.

## Konfigurácia zabezpečenia siete

Táto funkcia umožňuje aplikáciám prispôbiť 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. Týmto 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.

### 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 odstraňajú.

- **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 aplikáciám 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 odstraňujú.
- **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 štrukturovaných údajov.

## Prístup k poloha zariadenia

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

## 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 enviromentálne parametre, ako je teplota, tlak okolitého vzduchu, osvetlenie a vlhkosť. Do tejto kategórie spadajú barometre, fotometre a teplomery.

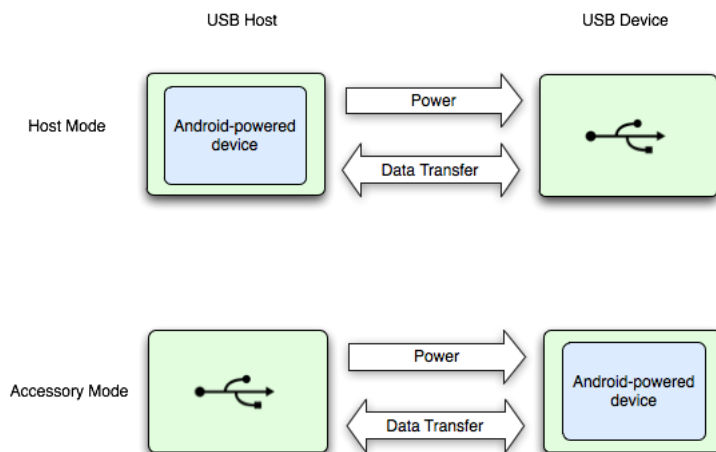
- **Snímače polohy** - merajú fyzickú polohu zariadenia. Tu patria senzory orientácie a magnetometre.

## Prvky konektivity

Podpora technológie **Bluetooth** nie je výnimkou ani u tohto operačného systému. Táto technológia umožňuje zariadeniam bezdrôtovo vymieňať 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.

Ďalšiou podporovanou je služba **Near Field Communication** skôr známa pod skratkou NFC. Služba umožňuje zdieľať malé dáta medzi štítkami NFC a zariadením, ktoré túto technológiu podporuje, alebo medzi dvoma takými to zariadeniami. Existujú jednoduché štítky, ktoré ponúkajú len sématické 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 menovaní 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 a 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 Wi-Fi Peer-to-Peer, čo je technológia, vďaka ktorej dokážu zariadenia medzi sebou komunikovať v prípade, že sú pripojené na rovnakú Wi-Fi sieť a táto technológia bude použitá pri posielaní si súborov medzi sebou.

### 3.1 Serializácia

Ako uvádza [3] 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 objektami, a tak ak je objekt uložený, uložia sa aj všetky objekty, ktoré sú z tohto objektu prístupné, aby nedošlo k poškodeniu vzťahov medzi týmito objektami. Serializované objekty je možné prenášať cez počítačovú sieť, alebo medzi zariadeniami.

*[TODO: príklad]*

### 3.2 Reflexia

Ako hovorí [2] 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, kde ide o reflexnú schopnosť, ktorá mení správanie programu tým, že priamo ovládne toto správanie.

Podľa [4] sa jedná o pokročilú vlastnosť jazyka, ktorá umožňuje vykonávať operácie, ktoré by inak nebolo možné vykonávať, napríklad umožňuje obísť zapúzdrenie (sprístupňuje



privátnych členov). To však znamená riziko narušenia správneho behu programu a aj vďaka tomu, by mala byť reflexia používaná čo najmenej a ľuďmi s dostatočnými znalosťami. Veľkou výhodou, ktorú nám reflexia umožňuje, je tá, že aplikácie sa ľahšie prispôbujú 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 prispôbovať 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.

---

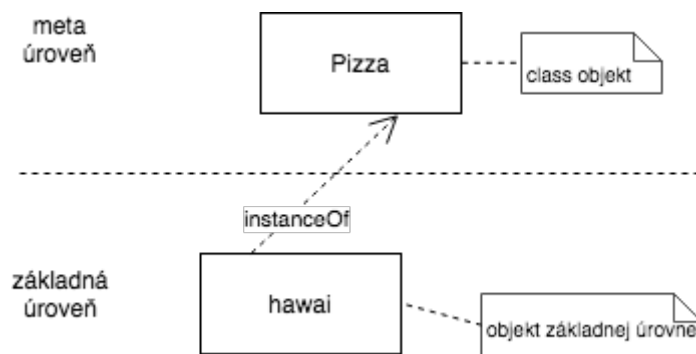
```
public static void setObjectColor( Object object, Color color ) {  
    Class cls = object.getClass();  
    Method method = cls.getMethod( "setColor", new Class[] {Color.class} );  
    method.invoke( obj, new Object[] {color} );  
}
```

---

Výpis 3.1: Použitie reflexie - pseudokód

Vo výpise 3.1 je znázornené využitie reflexie. Predstavme si nasledujúci problém, ktorý nám vznikol pri implementácii používateľského prostredia, kedy sú použité rôzne štandardné vizuálne komponenty jazyku 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 3.1. 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 objektami 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. 3.1: Základna a meta úroveň v reflexii

## 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 inštanície sú **metaobjektami**, ktoré Java používa na reprezentáciu tried, ktoré tvoria program. Tieto objekty sú najdôležitejším 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<sup>1</sup> 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, vrátane tých, ktoré deklaruje trieda, alebo rozhranie a tiež tie, ktoré boli zdedené z rodičovských tried, alebo rodičovských rozhraní.

## 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 návra-

<sup>1</sup><https://docs.oracle.com/javase/7/docs/api/java/lang/Class.html>

<sup>2</sup>Zoznam všetkých metód: <https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Method.html>

tovom type, názvu, typy parametrov, typ výnimky prípadne anotáciu. Objekt `Method` nám taktiež umožňuje zavolať metódu, ktorú predstavuje. Metódy, ktoré okrem iných táto trieda obsahuje<sup>2</sup>:

- `Annotation[] getDeclaredAnnotations()` - vracia všetky anotácie, ktoré sú priamo prítomné v tomto prvku.
- `Class getDeclaringClass()` - vracia objekt `Class` 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

*[TODO: Popísať Django, čo to je, prečo bol využitý, atď. + REST API k tomu.]*

### 3.4 Firebase Cloud Messaging

*[TODO: Čo je firebase, čo je cloud messaging, prečo je vhodné ho použiť]*

## Kapitola 4

# Existujúce riešenia

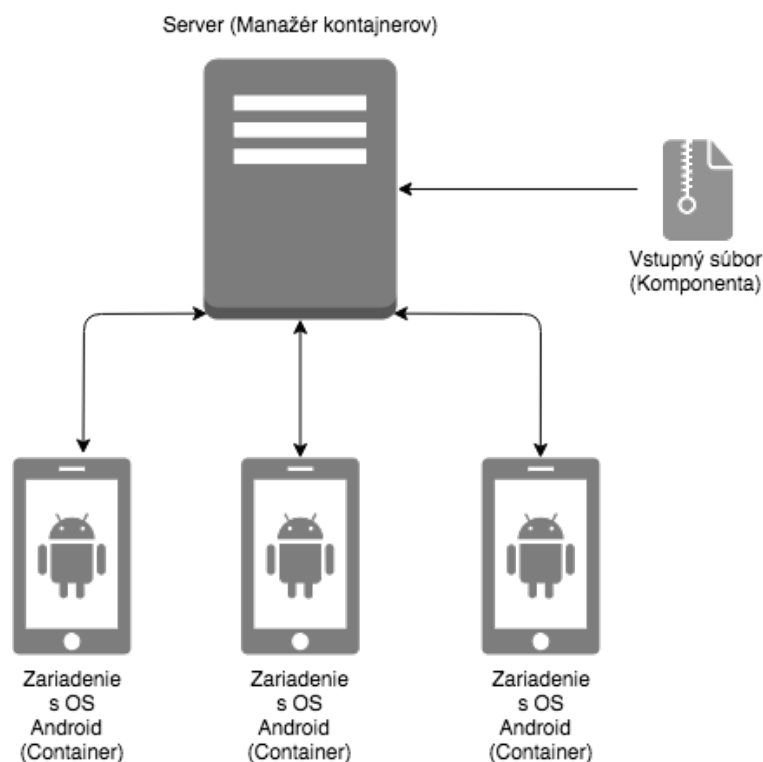
*[TODO: čo to je, existujúce riešenia]*

## Kapitola 5

# Návrh aplikácie

Cieľom tejto práce je navrhnuť kontajner pre komponenty bežiacie na operačnom systéme Android, to je rozhoranie 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. Ďalej 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

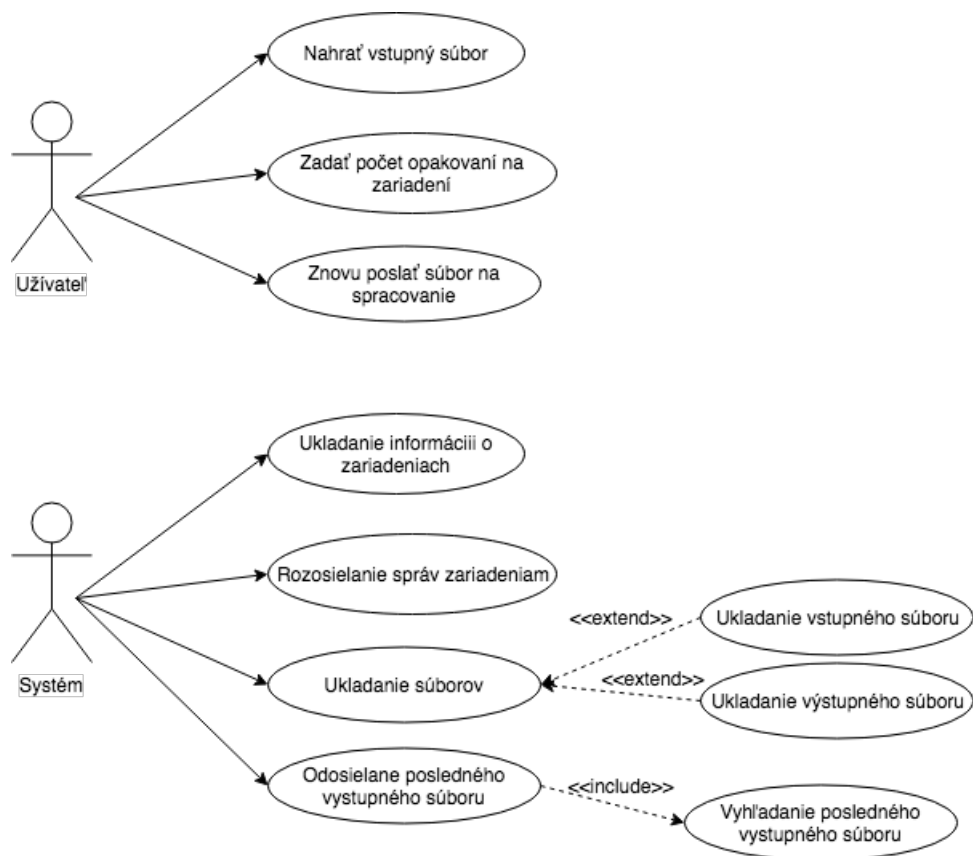


Obr. 5.1: Návrh architektúry aplikácie

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 rozdistribovaná na kontajnery. Zariadenie obsahujúce kontajner sa pri inštalácii zaregistruje na manažérovi, ktorý si túto informáciu uloží. Po nahratí komponenty, manažér odošle zaregistrovaným zariadeniam správu, s obsahom súboru, ktorý je potrebné spracovať. Kontajner po prijatí správy, správu spracuje a začne spustiť svoj životný cyklus.

## 5.2 Návrh serverovej časti aplikácie

Hlavnou úlohou server je ukladanie súborov a posielanie požiadaviek na ich spracovanie., no tiež bude server využívaný na registráciu zariadení, ktoré budú čakať na prijatie požiadavku. Ako je možné vidieť na obrázku 5.2, v tomto systéme sú zahrnuté dve role a to užívateľ a samotný systém. Užívateľ zaháji 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.



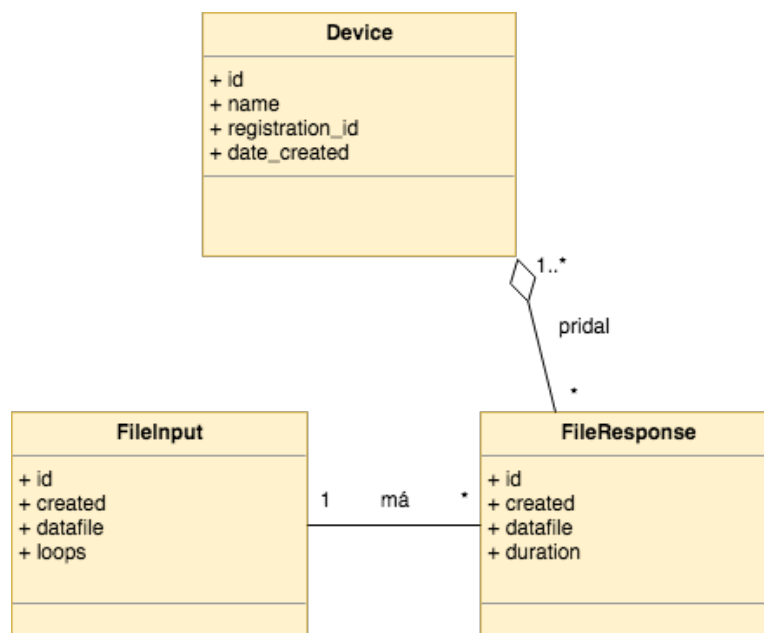
Obr. 5.2: Diagram prípadu užitia serverovej časti aplikácie

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

Vykonávanie všetkých týchto operácií 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 5.3.

## Doménový model

Trieda **Device** obsahuje atribúty, potrebné k identifikácii zariadenia, ktoré je pripojené. Týmito atribútmi sú jeho názov, unikátne registračné číslo a dátum registrácie. Táto trieda agreguje triedu **FileResponse**, ktorú pridáva do databázy a týchto súborov môže byť viacero. Trieda **FileResponse** obsahuje identifikačné číslo, dátum vytvorenia, súbor a čas, ktoré zariadenie vykonalo spracovávaním súboru. Trieda **FileInput** môže mať viacero výstupov vo forme **FileResponse** a obsahuje id, dátum vytvorenia, súbor a počet opakovaní.



Obr. 5.3: Doménový model serverovej časti aplikácie

## REST API

System 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 operácií, 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 nasledovne:

- GET /file-response - zobrazenie všetkých výstupných súborov
- POST /file-response - nahrávanie výstupného súboru, vo forme `formData`, kde sa očakáva súbor, identifikačné číslo zariadenia, názov vstupného 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 preskúmávanie vstupného súboru pomocou reflexie, načítavanie a ukladanie súborov. Komplexný výpis operácií je zobrazený v diagrame prípadov užitia 5.4. 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 budú 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 prísť 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.



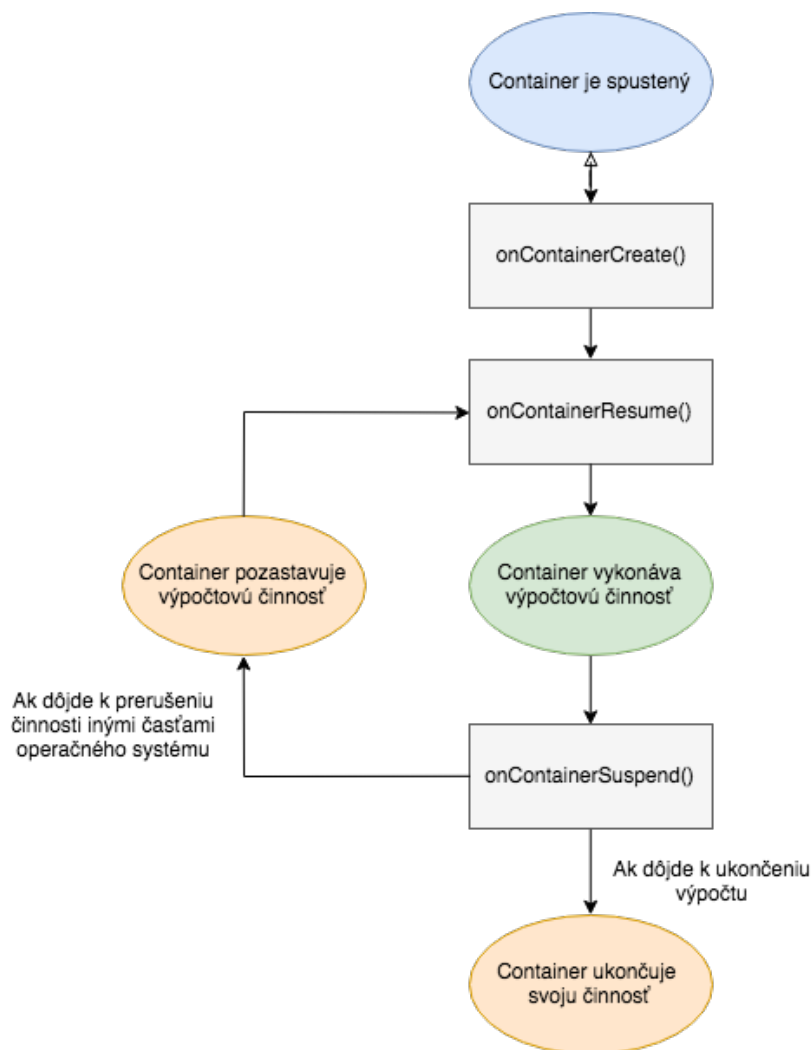
Obr. 5.4: Diagram prípadu užitia kontajneru

## 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 kontajnera 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 nezatažovať výpočetné prostriedky hostiteľa kontajneru, môže zavolať operáciu **onContainerSuspend()**, kedy dôjde k uloženiu momentálneho stavu objektu, nad ktorým prebieha výpočet. Ak dôjde k opätovnému uvoľneniu výpočetných 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 5.5.

## 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 5.6,



Obr. 5.5: Životný cyklus kontajneru

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**:

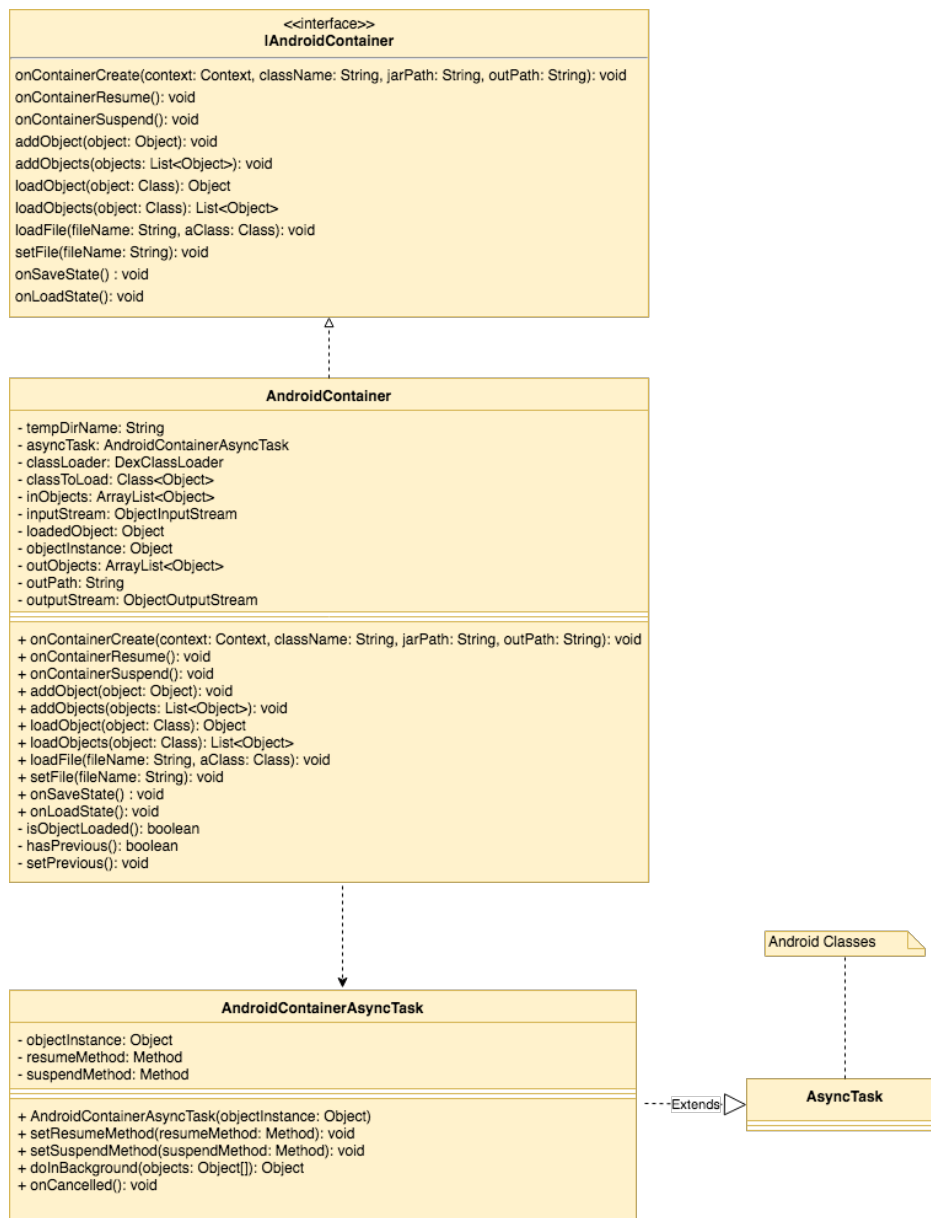
- `tempDir: String` - udáva názov dočasného optimalizovaného adresára, ktorý sa používa pri vytváraní objektu triedy `DexClassLoader` avšak od API verzie 26 je zastaralá a nepoužíva sa.
- `asyncTask: AndroidContainerAsyncTask` - atribút, ktorý drží referenciu na objekt triedy `AndroidContainerAsyncTask`, 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 preskúvanom vstupnom súbore
- `objectInstance: Object` - inštancia triedy, ktorá je preskúvaná 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 data 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 preskúvateľného vstupného súboru
- `hasPrevious(): void` - kontrola, či existuje výstupný súbor, obsahujúci serializovaný objekt z predchádzajúceho výpočtu
- `setPrevious(): void` - metóda pre nastavenie objektu, ktorý má byť preskúvaný pomocou reflexie



Obr. 5.6: 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. Týmto **metódami** sú:

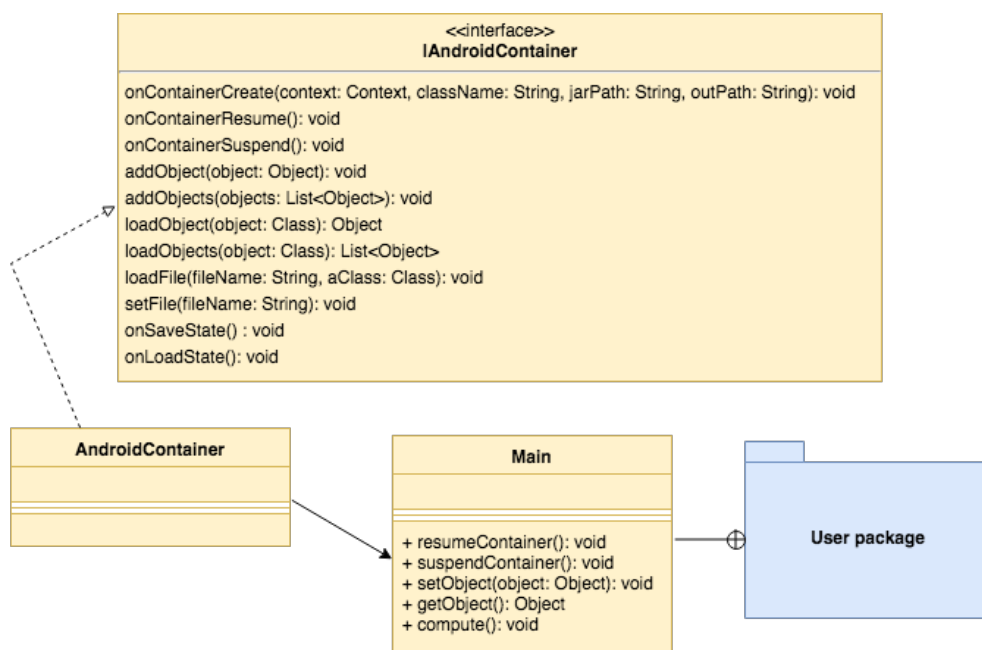
- **doInBackground(objects: List<Object>): Object** - metóda, ktorá ma za úlohu zahájenie výpočtovej činnosti kontajneru
- **onCancelled(): void** - metóda, ktorá ma za úlohu prerušiť výpočtovú činnosť kontajneru

## 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 zaobahuje objekty, ktoré sú používané k vykonaniu výpočtu. Túto situáciu si môžeme ukázať na grafe 5.7. 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 ma 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. 5.7: 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 Bubblesort.

### 6.1 Implementácia serverovej časti aplikácie

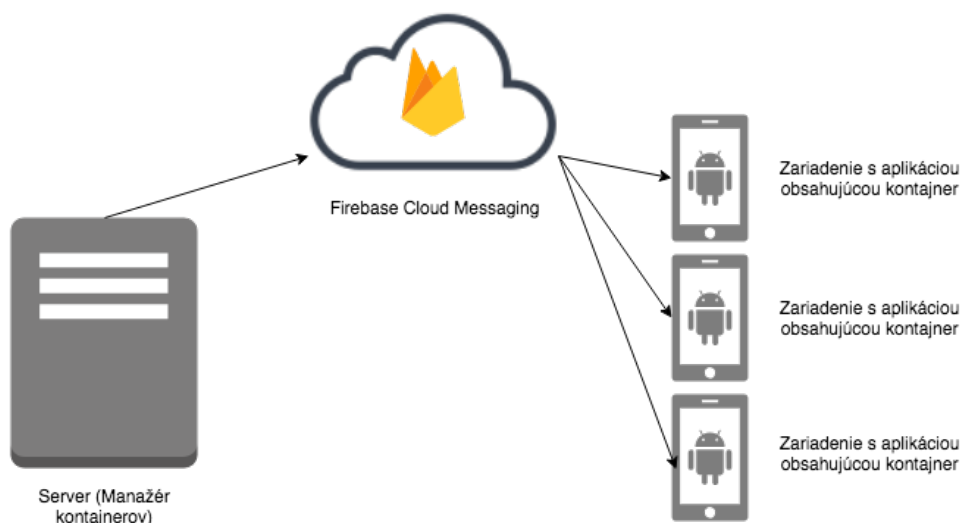
Ako už bolo spomenuté, úlohou servera je distribúcia komponent (vstupných súborov) na zariadenia, ktoré sú dostupné pre zahájenie výpočtu nad týmto súborom. Pre implementáciu primitívneho serveru, ktorý bude zvládať tieto jednoduché úlohy bol vybraný jazyk Python a framework Django s rozšírením Django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

Dôležitou časťou serveru, je obojstranná komunikácia so zariadeniami v reálnom čase. Keďže 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 preposielať správy zo serveru na mobilné zariadenie. Po vybraní vhodnej technológie - **Firestore Cloud Messaging**, ktorá je spomínaná v kapitole použité technológie, sme problém so zasielaním správ vyriešili.

Celkové preposielanie správ je implementované tak, že po prijatí prvého súboru, odošle nami implementovaný server správu, do Firestore Cloud Messagingu, ktorý ďalej správu rozistribuje medzi zariadenia. Tento priebeh je ilustrovaný na obrázku 6.2.

#### Firestore Cloud Messaging Django

Pre implementáciu posielanie správ medzi serverom a Firestore Cloud Messaging, bol využitý balík `fcm-django` dostupný na <sup>1</sup>. Tento balík nám poskytuje zjednotenú platformu na odosielanie správ na mobilné zariadenia, či prehliadače. Obsahuje funkcionality, ktorá nam 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 nam 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 posilať rôzne dáta, v našom prípade ide o posielanie URL adresy vstupného súboru, a počtu opakovaní na zariadení.

Vyvolanie metódy na odosielanie správy je popísané 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 odosiadaná 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

## 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 potreba implementovať triedu, ktorá sa stará o uchovávanie informácií 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 komponentách (vstupných súboroch)

---

<sup>1</sup><https://github.com/xtrinch/fcm-django>



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

## 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 ukladať. 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 modeli. 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

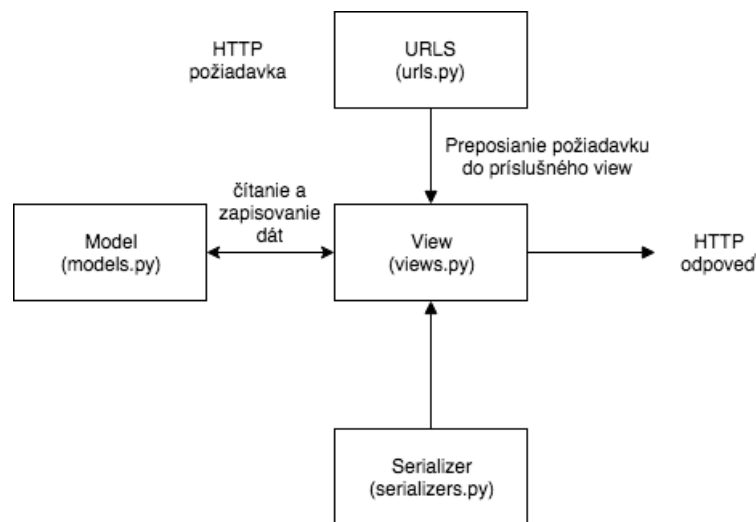
Použitie tejto databázy nám zabezpečuje uloženie informácií o zariadeniach, vstupných súboroch a výstupných súboroch.

## 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 ?? a jeho algoritmus pracuje nasledovne:

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 premennú **urlpatterns** (inštancia **django.conf.urls.url()**).
3. Následne Django prechádza všetkými vzormi adresy URL (zapísané pomocou regulárnych výrazov) v poradí akom sú napísané, a zastaví sa na prvom, ktorý zodpovedá požadovanej adrese URL.
4. Ak sa zhoduje jeden z výrazov definovaný regulárnym výrazom, Django importuje a vyvolá daný pohľad. Pohľad potom dostáva nasledujúce argumenty:

- inštancia objektu `HttpRequest`
  - ak porovnávať 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 `kwargs`
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 nasledovne a definuje spoločnú adresu, ktorá zaobahuje 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 požiadavky na získanie výstupného súboru. URL cesta ktorá definuje tieto volania je `file-response/` a registrácia tejto adresy bolo potrebné napísať nasledovné:

---

```

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 FCMDDeviceViewSet
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 FCMDDevice

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 prispôbiť ukladanie a nahrávanie modelovej triedy z databázy.

---

```

from django.db import models
from fcm_django.models import FCMDDevice

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(FCMDDevice)

    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 FCMDDevice

class FileResponseSerializer(serializers.HyperlinkedModelSerializer):
    device_id = serializers.SlugRelatedField(slug_field='registration_id',
        queryset=FCMDDevice.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

---

## 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

*[TODO: popísať, reflexia, metody, triedy...]*

## 6.3 Implementácia ukážkovej komponenty

*[TODO: popísať, implementácia....]*

## 6.4 Implementácia vzorovej Andorid aplikácie

## Kapitola 7

# Možnosti rozšírenia kontajneru

*[TODO: prediskutovať, čo by sa mohlo zlepšiť - ukončujúci stav, server, aplikácia]*

## Kapitola 8

## Záver

*[TODO: čo sa podarilo, čo nie, aký je výsledok....]*

# Literatúra

- [1] Developers, A.: *Android developer guide*. [Online; navštíveno 15.03.2018].  
URL <https://developer.android.com/guide/index.html>
- [2] Forman, I. R.; Forman, N.: *Java Reflection in Action (In Action Series)*. Greenwich, CT, USA: Manning Publications Co., 2004, ISBN 1932394184.
- [3] 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>
- [4] 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>
- [5] 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.