TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Serverová komponenta pro aktualizaci modelu autonomního navigačního systému

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Jan Kozánek**
Vedoucí práce: Ing. Igor Kopetschke

Liberec 2017

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Server Component for Model Update of Autonomous Navigation System

Master thesis

Study programme: N2612 – Electrical Engineering and Informatics
Study branch: 1802T007 – Information Technology

Author: **Bc. Jan Kozánek**
Supervisor: Ing. Igor Kopetschke

Liberec 2017

Technická univerzita v Liberci
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2016/2017**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jan Kozánek**
Osobní číslo:	**M15000173**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Informační technologie**
Název tématu:	**Serverová komponenta pro aktualizaci modelu autonomního navigačního systému**
Zadávající katedra:	**Ústav nových technologií a aplikované informatiky**

Z á s a d y p r o v y p r a c o v á n í :

1. Seznamte se s komunikačním rozhraním autonomního navigačního systému experimentálních zařízení typu auto a dron.
2. Navrhněte model serverové aplikace, která bude dle ad hoc požadavků zařízení aktualizovat jeho 3D model okolí.
3. Serverovou aplikaci implementujte v jazyce Java a OpenGL za použití mapových podkladů z ArcGIS a OpenStreetMap.
4. Proveďte vyhodnocení v rámci testovacího provozu.

Rozsah grafických prací: **dle potřeby**

Rozsah pracovní zprávy: **40 - 60 stran**

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

Seznam odborné literatury:

[1] SHREINER, Dave. OpenGL: průvodce programátora. Vyd. 1. Přeložil Jiří FADRNÝ. Brno: Computer Press, 2006. DTP & grafika. ISBN 80-251-1275-6.
[2] RESTful Web Services. Sebastopol: O'Reilly Media, Inc, 2008. ISBN 9780596554606.
[3] GELETIČ, Jan. Úvod do ArcGIS 10. 1. vyd. Olomouc: Univerzita Palackého v Olomouci, 2013. Skripta. ISBN 978-80-244-3390-5.
[4] ArcGIS REST API. ArcGIS REST API [online]. Český úřad zeměměřický a katastrální, 2016 [cit. 2016-10-12].
Dostupné z: http://ags.cuzk.cz/arcgis/sdk/rest/index.html
[5] OpenStreetMap API. OpenStreetMap [online]. OpenStreetMap Wiki, 2016 [cit. 2016-10-12]. Dostupné z:https://wiki.openstreetmap.org/wiki/API

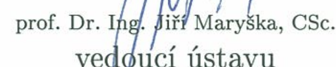
Vedoucí diplomové práce: **Ing. Igor Kopetschke**

Ústav nových technologií a aplikované informatiky

Datum zadání diplomové práce: **20. října 2016**

Termín odevzdání diplomové práce: **15. května 2017**

prof. Ing. Zdeněk Plíva, Ph.D.
děkan



prof. Dr. Ing. Jiří Maryška, CSc.
vedoucí ústavu

V Liberci dne 20. října 2016

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum: 15. 5. 2017

Podpis:

Poděkování

Rád bych poděkoval panu Ing. Igoru Kopetschkemu za pomoc při psaní diplomové práce a za jeho rady.

Abstrakt

Tato diplomová práce popisuje vývoj a ovládání webové služby, která poskytuje data autonomnímu navigačnímu systému pro zařízení typu dron a auto.

Nejprve je ve zprávě charakterizováno autonomní zařízení. Stručně jsou popsány hardwarové periferie a komunikační rozhraní experimentálního zařízení.

Další část se zabývá návrhem serverové aplikace, která dle ad hoc požadavků zařízení aktualizuje 3D mapu a okolí. Spolu s tím se dále zaobírá vytvořením schématu relační databáze pro ukládání této mapy.

Webová služba využívá veřejně dostupná REST API služeb ArcGIS a OpenStreetMap pro sběr příslušných mapových podkladů, ze kterých získá potřebné informace. Ze služby ArcGIS pomocí API získáme například nerovnost terénu, odpovídající texturu a nejvyšší a nejnižší body pro jednu dlaždici. Podobným způsobem ze služby OpenStreetMap získáme informace o objektech, kterým se zařízení má vyhnout. Pro objekty se spočítá půdorys a výška a z těchto informací následně dojde ke spojení a vytvoření vlastních mapových podkladů, o které může navigační systém požádat. Výsledky je možné vracet jak ve formátu JSON, tak i v XML.

Jelikož jsou tyto operace výpočetně velice náročné, po dokončení se jejich výsledky ukládají do databáze. Při budoucím dotazu na množinu dlaždic se již vypočtené dlaždice načtou z databáze a počítat se budou pouze nové neznámé.

Služba rovněž poskytuje data pro webovou aplikaci sloužící pro sledování dráhy letu dronu a pro desktopovou aplikaci sloužící pro výběr oblasti, ve které se navigační systém bude pohybovat.

Při programování byl použit programovací jazyk Java, aplikační server Tomcat 7.0, ArcGIS SDK 10.2.4 a databáze MariaDB.

Klíčová slova

ArcGIS, autonomní navigační systém, dron, Java, mapy, OpenStreetMap, serverová komponenta, webová služba

Abstract

This master thesis describes development and controlling of a web service which provides data to an autonomous navigation system used for drone and car devices.

First of all, we characterize an autonomous device. The hardware peripherals and communication interface of the experimental device are briefly described.

The next part deals with the design of a server application which updates the 3D map according to the ad hoc requests of the device. It also deals with the creation of a relational database schema for storing this map.

The web service uses the publicly available REST API of ArcGIS and OpenStreetMap to collect relevant map data to obtain the necessary information. From ArcGIS using API, we get terrain inequality, corresponding texture, and the highest and lowest points for a single tile. Similarly, from OpenStreetMaps we get information about the objects which a device should avoid. For objects, a floor plan and heights are calculated and this information will then be merged with information from ArcGIS and created for the maps itself, which the navigation system can request. The results can be returned both in JSON format and XML.

Since these operations are computationally demanding, the results are stored in a database after completion. For a future query on a set of tiles, the already calculated tiles will be loaded from database and only new unknown ones will be calculated.

The service also provides data for a drone flight tracking web application and a desktop application to select an area which the navigation system will move in.

Java programming language, Tomcat 7.0 application server, ArcGIS SDK 10.2.4 and MariaDB database were used for programming.

Keywords

ArcGIS, autonomous navigation system, drone, Java, maps, OpenStreetMap, server component, web service

Obsah

Prohlášení 4

Úvod 13

1. Popis experimentálních zařízení 14

1.1 Dron 14

1.2 Auto 14

1.3 Komunikace 15

2. Návrh modelu serverové aplikace 16

2.1 Výstupní formát 17

2.1.1 JSON 17

2.1.2 XML 18

3. Databáze 20

3.1 Základní požadavky 20

3.2 Databázový model 20

3.3 MariaDB 21

4. Serverová aplikace 22

4.1 Tomcat server a servlet 22

4.2 Specifikace serveru 25

5. Popis aplikace 26

5.1 Rozšířený model 26

5.2 Struktura aplikace 28

5.2.1 Web.xml 29

5.2.2 Context.xml 30

5.2.3 Adresář src 31

5.3 Převod souřadnic 32

5.4 ApplicationConfig 32

5.5 Použité externí knihovny 33

5.5.1 Gson ... 33

5.5.2 Jersey ... 34

5.6 Využití anotací a návratový typ metod ... 35

5.7 Seznam metod ... 36

5.7.1 Metody pro převod souřadnic ... 37

5.7.2 Metoda využívaná experimentálním zařízením ... 37

5.7.3 DBFactory ... 39

5.7.4 Získání DMR dat ... 41

5.7.5 Získání entit ... 42

5.7.6 Scéna ... 46

5.7.7 Metody pro vizualizaci map ... 47

6. Testování aplikace ... 49

7. Závěr ... 51

POUŽITÁ LITERATURA A ZDROJE ... 53

Seznam obrázků

Obrázek 1: Základní model aplikace ... 16

Obrázek 2: Relační model databáze ... 20

Obrázek 3: Princip servletu a webového serveru ... 22

Obrázek 4: Vícevláknový servlet ... 24

Obrázek 5: Rozšířený model aplikace ... 27

Obrázek 6: Základní adresářová struktura webové aplikace ... 29

Obrázek 7: Struktura balíčků ... 32

Obrázek 8: Úprava souřadnic ... 38

Seznam tabulek

Tabulka 1: Anotace pro mapování metod ... 34

Tabulka 2: Anotace pro získání parametrů metod ... 34

Tabulka 3: Seznam metod ... 36

Seznam zdrojových kódů

Zdrojový kód 1: Výstup ve formátu JSON ... 18

Zdrojový kód 2: Výstup ve formátu XML ... 19

Zdrojový kód 3: Nastavení servletu ... 30

Zdrojový kód 4: Namapování servletu ... 30

Zdrojový kód 5: Nastavení přístupu k prostředkům ... 30

Zdrojový kód 6: Údaje pro přístup k databázi ... 31

Zdrojový kód 7: Použití metod knihovny Gson ... 34

Zdrojový kód 8: Anotace metody ... 35

Zdrojový kód 9: Hledání dlaždice pro zadaný bod ... 39

Zdrojový kód 10: Tělo dotazu pro získání DMR ... 41

Zdrojový kód 11: Výstup ze služby getSamples 42

Zdrojový kód 12: Adresa volající službu OpenStreetMap 43

Zdrojový kód 13: Výstup služby OpenStreetMap 44

Zdrojový kód 14: Výstup DMR a DMR služby.. 46

Zdrojový kód 15: Struktura požadavku pro metodu getTilesMxNPost 47

Seznam zkratek a symbolů

DMR – Digitální model reliéfu České republiky

DMP – Digitální model povrchu České Republiky (obsahuje navíc stavby a rostlinný pokryv)

S-JTSK – Systém jednotné trigonometrické sítě katastrální. Souřadnicová síť používaná v geodézii na území České republiky a Slovenska, vycházející z Křovákova zobrazení

WGS-84 – World Geodetic System 1984. Světově uznávaný geodetický standard, který definuje souřadnicový systém.

JSON – Javascript Object Notation (způsob zápisu dat)

XML – Extensible Markup Language (obecný značkovací jazyk)

SQL – Structured Query Language (standardizovaný dotazovací jazyk používaný v databázích)

PK – Primary key (primární klíč)

FK – Foreign key (cizí klíč)

Úvod

Tato diplomová práce je součástí většího celku, který se skládá ze tří aplikací. Tyto aplikace bude využívat experimentální zařízení typu dron či auto, které má v sobě zabudovaný autonomní navigační systém. První částí je desktopová aplikace [1]. V ní si uživatel vybere cílovou oblast, ve které se zařízení bude pohybovat a zvolí způsob letu (náhodný pohyb nebo vybraná cesta). Po zvolení dané oblasti kontaktuje zařízení a předá mu informace o dané oblasti. Tyto informace získá ze serverové komponenty, která je předmětem této práce. Stejná data využívá i webová aplikace pro monitorování a tracking autonomních zařízení [2], která z nich vykreslí mapu a umožní sledování zařízení. Cílem diplomové práce tedy bylo vytvořit serverovou komponentu, která by vytvářela a následně poskytovala data těmto zdrojům.

Veškeré informace jsou získány z volně dostupných zdrojů. Těmi jsou poskytovatelé mapových podkladů ArcGIS [3] a OpenStreetMap [4]. Oba zdroje mají veřejně dostupná REST API, pomocí kterých data poskytují.

Služba OpenStreetMap je využívána ke zjištění informací o entitách. Ty v sobě nesou údaje o typu entity (budova, silnice, lesy a mnoho dalších) a její přesný půdorys.

Nicméně tento poskytovatel nemá údaje o výškách daných entit. Proto je potřeba využít ArcGIS, který tyto údaje má, a přiřadit výšku ke správným entitám. Stejně tak je tento zdroj využit pro zjištění informací o dlaždici, na které se entity nachází. Potřebujeme znát informace, jako je zakřivení terénu, nejvyšší a nejnižší výška dlaždice a její textura.

V momentě, kdy máme všechny informace pohromadě a správně seskupené, je potřeba tyto výsledky nějak zveřejnit. K tomu slouží aplikační server Tomcat, na kterém aplikace běží. Je zveřejněno několik metod poskytujících tato data buď ve formátu JSON nebo XML.

Toto téma jsem si zvolil, protože mě tématika webových služeb zajímá. Rovněž se mi líbila možnost vyzkoušet si spolupráci v týmu několika lidí na větším projektu.

1. Popis experimentálních zařízení

1.1 Dron

Tato aplikace je primárně určená pro zařízení typu dron. Škola disponuje dvěma typy dronů, které mohou tuto serverovou komponentu využívat. Prvním je DJI Phantom II a druhým DJI Tarot 690.

Oba modely je možné osadit kamerou Go Pro Hero 4, která disponuje rozlišením nastavitelným až na 4K. Přenos obrazu na ovládací jednotku operátora je uskutečněn pomocí jednotky FT956 na LCD panel s přijímačem pomocí 5.8GHz pásma. Pro ovládání kamer jsou k dispozici Arduino klony, jako jsou například Arduino NANO, Arduino 2560 CoreBoard nebo Raspberry Pi 2.

K řízení dronu se používá systém od DJI, konkrétně NAZA-M V2. Tento systém obsahuje komponenty jako například kompas, gyroskopy, akcelerometry a tlakové čidlo. K dispozici je také DJI DataLink LK-24BT, kterým lze dron na dálku ovládat a programově nastavovat pomocí bodů trasu, kterou má dron letět.

Samotný dron rovněž disponuje periferiemi, jako jsou akcelerometry a gyroskopy CJMCU-MPU, dále tlakové čidlo BMP180 a PWM čip PCA9685, který se používá pro řízení servomotorů a regulátorů. Všechny tyto periferie komunikují po sběrnici I^2C s Raspberry Pi 2.

Svoji polohu může dron zjišťovat několika způsoby. Buď pomocí GPS zařízení GT-730F nebo také díky GPS modulu NEO-6M, případně NEO-7M. Pro práci s těmito periferiemi se používá knihovna TinyGPS.

1.2 Auto

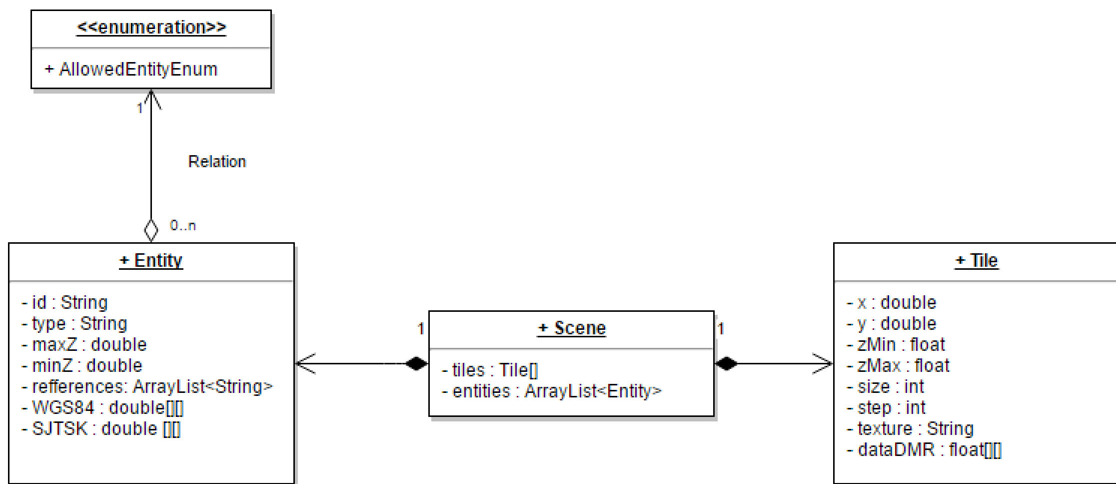
Jedná se o auto na dálkové ovládání, které je v podstatě odlehčenou verzí typu dron. Je možné osadit jej stejnými periferiemi. Na rozdíl od dronu nás však nebude zajímat výška. Komunikace probíhá stejným způsobem.

1.3 Komunikace

Jelikož je dron rovněž vybaven SIM kartou podporující LTE, má možnost využívat datovou síť. Webová aplikace je veřejně dostupná na školní síti, takže stačí pomocí metody GET odeslat na adresu služby dotaz s aktuální polohou zařízení a se směrem letu/jízdy a veškeré informace mu budou poskytnuty.

2. Návrh modelu serverové aplikace

Jelikož tato serverová komponenta funguje jako prostředník mezi více body, bylo potřeba pečlivě rozhodnout, jakou strukturu bude aplikace mít. Již na začátku bylo totiž potřeba znát přesně výstupní formát, aby ostatní členové týmu mohli z tohoto formátu vycházet a postavit na nich své aplikace. Základní vyobrazení modelu se nachází na obrázku níže a v této kapitole jej podrobně popíšu.



Obrázek 1: Základní model aplikace

Tabulka *Tile* reprezentuje třídu obstarávající výpočet informací o jednotlivých dlaždicích. Má atributy *x* a *y*, což jsou souřadnice jednoznačně identifikující danou dlaždici. Dále *zMin* a *zMax* reprezentující nejnižší a nejvyšší výšku dlaždice, díky kterým lze dlaždici vykreslit ve správné výšce. Dalšími vlastnostmi jsou *size* a *step*, které označují velikost dlaždice v metrech a krok, po kterém jsou získávána data. Tato data jsou uložena ve dvourozměrném poli *dataDMR* a konkrétně se jedná o digitální model reliéfu krajiny neboli zakřivení dlaždice.

Další důležitou tabulkou je *Entita*, která reprezentuje všechny objekty vyskytující se v tázané dlaždici. Každá tato entita má unikátní identifikátor. Poté svou nejvyšší výšku *maxZ* a také *minZ,* díky které lze objekt správně položit na dlaždici. Dále bylo rozhodnuto, že pro budoucí rozšiřitelnost se informace o půdorysu entity budou ukládat v několika formátech. Prvním z nich jsou reference, pod kterými jsou uloženy v **.osm* souborech. OSM soubor je výstup služby OpenStreetMap a obsahuje informace o všech entitách pro vybranou oblast. Je tedy možné zpětně dohledat entitu pomocí referencí

v těchto souborech. Dalším formátem je WGS-84. Jedná se o světově uznávaný geodetický standard, který definuje souřadnicový systém. Souřadnice WGS-84 vycházejí ze souřadnic zeměpisných, polohu tedy určíme pomocí zeměpisné délky a šířky (*longitude* a *latitude*). Posledním formátem je S-JTSK, ve kterém jsou uloženy souřadnice WGS-84 převedené do souřadnicové sítě používané v geodézii na území České republiky. Posledním atributem je typ určující, o jakou překážku se jedná. S tím souvisí výčtový typ *AllowedEntityEnum*. V něm se nachází konečná množina pojmenovaných hodnot. Tu lze v budoucnu rozšiřovat a přidávat další entity, které chceme zpracovávat. Momentálně se počítá se silnicemi, budovami, přírodním pokryvem a veřejnými zařízeními.

Na závěr se entity i dlaždice uloží do polí nacházejících se v tabulce *Scene*, kde se celý formát serializuje a odešle uživateli.

2.1 Výstupní formát

Pro všechny výše zmiňované atributy bylo potřeba připravit formát, ve kterém budou zasílány jako odpověď. Primárně je využíván formát JSON pro jeho jednoduchost a snadnou rozšiřitelnost, nicméně lze zvolit i formát XML.

2.1.1 JSON

Příklad formátu JSON se nachází níže. Za klíčem *„tiles“* jsou za sebou jako hodnoty uvedeny jednotlivé dlaždice. Ty jsou seřazené podle souřadnic (od levého horního rohu po pravý dolní). Po dlaždicích jsou obdobným způsobem vypsány všechny entity pro celou celkovou oblast. Jsou však rovněž seřazeny, aby odpovídaly pořadí dlaždic.

```
{
    "tiles": [{
            "x": -743808,
            "y": -1043584,
            "zMin": 186.205,
            "zMax": 189.49333,
            "size": 128,
            "step": 2,
            "texture": "iVBORw…",
            "dataDMR": [[186.77089, …], [186.63, …], …]
```

```
            },
            {další dlaždice…}],

    "entities": [{
            "id": "30169348",
            "type": "building",
            "maxZ": 202.72000122070312,
            "minZ": 187.9991455078125,
            "SJTSK": [[-743723, -1043673], …]},
             {další entity…},]
}
```

Zdrojový kód 1: Výstup ve formátu JSON

2.1.2 XML

XML formát má podobnou strukturu jako JSON, až na pár drobných odlišností. Jelikož se odesílá jako odpověď instance scény, je celý XML zabalen v kořenovém tagu *<scene>*. Další změnou je dvojrozměrné pole hodnot reliéfu. V XML není vypsáno jako pole polí, ale jednotlivá vnitřní pole jsou o úroveň výš seřazena za sebou a mají stejný název *dataDMR*. Jednotlivé hodnoty jsou uvnitř tagu *<item>*. Obdobně je to u entit a jejich půdorysů. Odlišnost je jen v názvu tagu, který je v tomto případě *<SJTSK>*. Ukázka výstupu je níže:

```
<scene>
      <tiles>
            <x>-743808.0</x>
            <y>-1043584.0</y>
            <zMin>186.205</zMin>
            <zMax>189.49333</zMax>
            <size>128</size>
            <step>2</step>
            <texture>iVBORw…</texture>
            </dataDMR>
                  <item>186.77089</item>
                        …
            </dataDMR>
            <dataDMR>
                  <item>186.63</item>
                        …
            </dataDMR>
                  …
      </tiles>

      <entities>
      <id>30169348</id>
      <type>building</type>
      <maxZ>202.72000122070312</maxZ>
      <minZ>187.9991455078125</minZ>
      <SJTSK>
            <item>-686717.0</item>
```

```
            <item>-973231.0</item>
        </SJTSK>
        <SJTSK>
            <item>-686757.0</item>
            <item>-973236.0</item>
        </SJTSK>
        …
        </entities>
        …
</scene>
```

Zdrojový kód 2: Výstup ve formátu XML

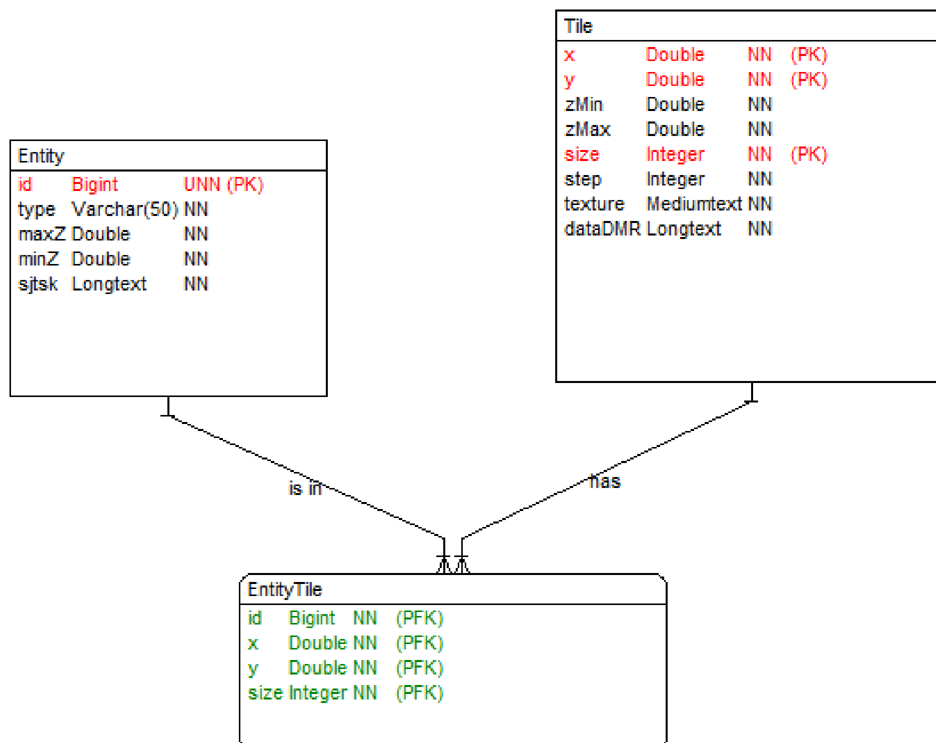
3. Databáze

3.1 Základní požadavky

Jelikož aplikace pro získání dat využívá REST API cizích poskytovatelů, je zpomalena o dobu, kterou trvá těmto serverům, než požadavek zpracují a odešlou výsledek. Bylo zjištěno, že výpočet jedné dlaždice trvá v průměru 15 sekund. Nejnáročnější je výpočet zakřivení terénu. Na získání reliéfu celé dlaždice se musí služba volat hned několikrát, jelikož všechna potřebná data se nevejdou do jednoho POST volání. A každé volání trvá několik sekund. Jelikož očekávaný počet dlaždic je minimálně devět, čekáme rázem na jeden požadavek 2 minuty a 15 sekund. Z tohoto důvodu bylo zavedeno *cachování* již vypočtených dlaždic do databáze. Dotaz na stejnou oblast, která je však již uložená v databázi, zkrátil dobu čekání na výsledek aplikace na 5 sekund.

3.2 Databázový model

V programu CASE Studio 2 byl vytvořen relační model celé databáze. Ten popisuje základní požadavky na aplikaci kladené a zobrazuje vzájemné vztahy jednotlivých bloků. Každá entita (tabulka) obsahuje unikátní identifikátor.



Obrázek 2: Relační model databáze

Tabulka *Tile* obsahuje informace o příslušné dlaždici. Nacházejí se v ní koordináty *x* a *y*, nejnižší výška, nejvyšší výška, velikost dlaždice, její krokování, textura a data ohledně reliéfu. Všechny atributy používají jednoduché datové typy kromě textury a DMR dat. Textura je uložena ve formátu Base64 a pro její uložení je použit datový typ Mediumtext, který dokáže uložit až 16 MB. Pro rozsáhlá data reliéfu byl použit větší datový typ Longtext, který má kapacitu 4 GB. Primárním klíčem jsou souřadnice x, y a její velikost. Jedná se o složený klíč z toho důvodu, že v budoucnu bude možné uchovávat záznamy o mapě pro různé velikosti dlaždic. Prozatím se používá velikost 128 × 128 metrů.

V tabulce *Entity* uchováváme informace o objektech, které se mohou v dané dlaždici vyskytovat. Momentálně se počítá se čtyřmi typy objektů. *Highways*, neboli všechny druhy silnic, po které bude moci auto jezdit, dále *buildings*, tedy všechny budovy a zatím provizorně *amenity* a *landuse*, což jsou veřejná zařízení a přírodní překážky. V budoucnu se počítá s rozšířením možných typů. Pro každý z těchto objektů se uchovává jeho maximální a minimální výška. Minimální výška se shoduje s výškou terénu, na kterém objekt stojí, aby bylo možné ho na mapě zanořit do terénu. Posledním atributem je samozřejmě půdorys, který ohraničuje kde přesně se objekt nachází. Jelikož se jedná o dvojrozměrné pole souřadnic, byl na něj použit datový typ *Longtext*.

Poslední entitou v databázi je slabá entita *EntityTile*, která vznikla propojením dvou předchozích entit ve vazbě M:N. V této tabulce je uchována informace o tom, v jakých dlaždicích se entita nachází. Jedna entita totiž může stát na hranici mezi několika dlaždicemi a je potřeba tu samou entitu vrátit při dotazu na jakoukoliv z dlaždic, ve kterých se nachází. Primárním cizím klíčem je zde kombinace primárních klíčů ostatních tabulek, tedy id entity, koordináty dlaždice a její velikost.

3.3 MariaDB

V této práci byla použita databáze *MariaDB*. Jedná se o relační databázi, která je komunitou vyvíjenou nástupnickou větví (tzv. „forkem“) MySQL. Tato větev byla vytvořena z důvodu udržení licence svobodného softwaru GNU GPL. Po odkoupení MySQL společností Oracle byly totiž obavy o další osud směřování softwaru [5]. Na školní server byla ze stejného důvodu zvolena tato větev.

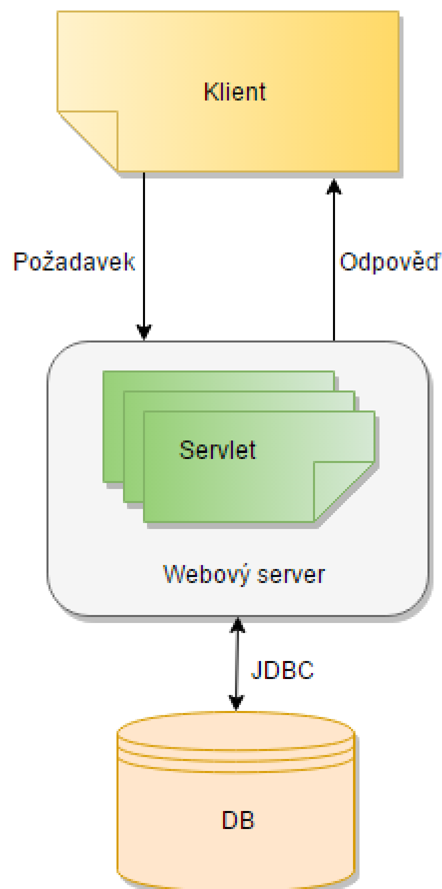
4. Serverová aplikace

Jelikož má být aplikace veřejně dostupná, bylo potřeba ji spustit na aplikačním serveru. Na školní síti byl proto nainstalován Apache Tomcat.

4.1 Tomcat server a servlet

Apache Tomcat je aplikační server a servlet kontejner. Je vyvíjen jako open source projekt a je založen na jazyce Java [6].

Servlet je zjednodušeně řečeno třída napsaná v Javě, která odpovídá na určitý typ síťových požadavků [7]. Nejčastěji na HTTP požadavky. Servlety jsou komponenty nezávislé na platformě a protokolu a jsou vykonávané na serveru. Podle specifikace jsou nevizuální, tedy nezobrazují se pomocí žádného grafického uživatelského rozhraní. Tyto servlety pak běží v servlet kontejneru, jako je například právě Tomcat. Jejich funkcionalita je znázorněna na obrázku níže:



Obrázek 3: Princip servletu a webového serveru

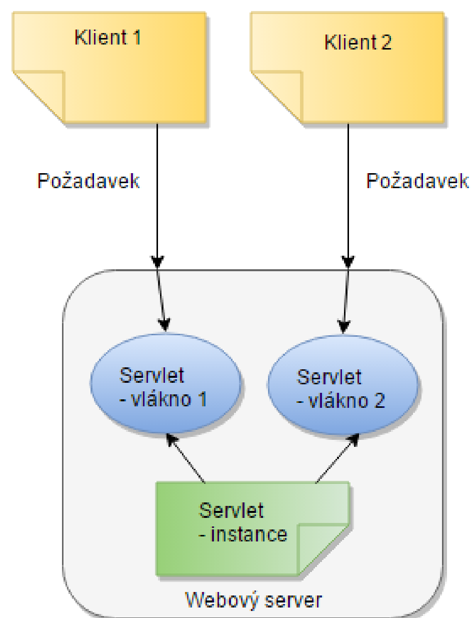
1. Klient odešle požadavek na server.
2. Server přepošle požadavek na konkrétní servlet.
3. Servlet vytvoří odpověď a předá jí zpět serveru. Odpověď je vytvořená dynamicky a její obsah závisí na konkrétním požadavku. Lze také používat externí zdroje, jako je například databáze připojená pomocí JDBC.
4. Odpověď je odeslána zpět klientovi.

Nespornou výhodou servletů je jejich přenositelnost a platformová nezávislost. Tuto vlastnost mají díky jazyku Java. Spolu s tím mají všechny výhody jazyka Java, jako je objektovost, modularita, bezpečnost apod.

Dalším přínosem je jejich výkonnost. Servlet je zkompilován a načten do paměti pouze jednou a z ní je volán při každém požadavku klienta. Díky tomu servlet méně vytěžuje systémové zdroje, databázové připojení apod.

Jak již bylo zmíněno, klient nekomunikuje přímo se servletem, ale s jeho nadřazeným serverem. Úlohou serveru je řídit spouštění a inicializaci servletu spolu s jeho ukončením a vymazáním z paměti. Záleží na nastavení serveru, kdy bude instance servletu uvolněna z paměti.

V případě, že na jeden servlet dorazí více požadavků najednou, musí server vytvořit pro každý požadavek vlákno. To znamená, že servlety jsou vícevláknové. Tato vlastnost je znázorněna na ilustraci níže:



Obrázek 4: Vícevláknový servlet

1. Na začátku je při prvním požadavku servlet načtený serverem. Všechny proměnné instance jsou inicializované. Servlet zůstává aktivní po celou dobu jeho životnosti.
2. Dva klienti zažádají o službu ten samý servlet. Server proto vytvoří dvě nezávislá vlákna, přičemž každé vlákno má přístup k instanci a jejím metodám a proměnným.
3. Každé vlákno obsahuje vlastní požadavky a odpovědi, které vrací příslušnému klientovi.

Webové aplikace se na server nasazují (*deploy*) pomocí WAR souborů. WAR (Web application ARchive) má stejnou strukturu jako klasičtější JAR soubor, tedy jedná se o jeden soubor, který zapouzdřuje celou aplikaci.

Ve WAR souborech se nachází servlety, třídy v jazyce Java, XML soubory, JAR knihovny a další zdroje potřebné pro běh aplikace. Když je tento soubor nasazen na server, kontejner (v našem případě Tomcat) jej rozbalí a spustí. WAR soubory nelze upravovat za běhu, tudíž při jakékoliv změně v kódu je potřeba vytvořit nový WAR soubor a znovu jej nasadit na server. Před spuštěním však Tomcat ještě vyhledá popisovač nasazení ve formě souboru *web.xml*, což je konfigurační soubor obsahující informace, které server potřebuje znát pro nasazení. Obsahuje například název servletu nebo

jeho mapování. Může také obsahovat informace o zdrojích, ke kterým se aplikace bude připojovat. [8].

4.2 Specifikace serveru

Webová aplikace běží na školním serveru. Ten disponuje procesorem Intel® Xeon® CPU E5-2650 taktovaném na 2 GHz (maximální turbo frekvence 2,8 GHz). Procesor má 8 jader a 16 vláken. K dispozici jsou 4 GB RAM a na serveru běží operační systém CentOS ve verzi 6.9.

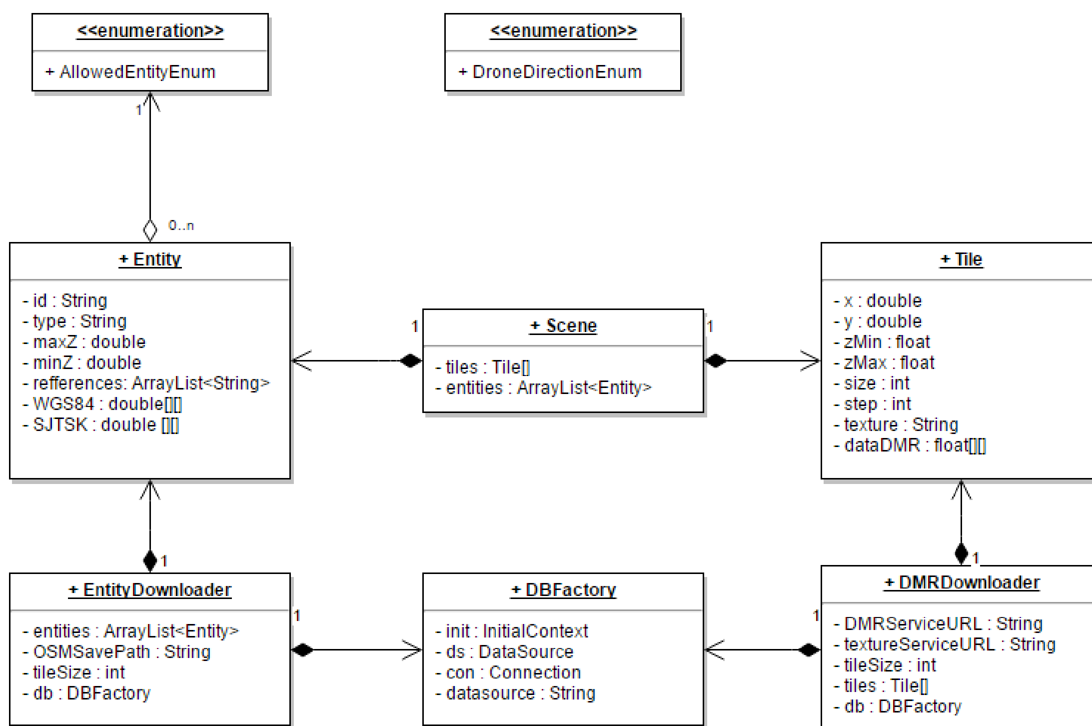
5. Popis aplikace

Aplikace byla vyvíjena ve vývojovém prostředí IntelliJ IDEA Ultimate, neboť tato edice na rozdíl od Community podporuje Java EE (Enterprise Edition), což je součást platformy Java, určená pro vývoj a provoz podnikových aplikací a informačních systémů. Platforma Java EE je potřebná pro vývoj webových aplikací v Javě [9].

Původní ideou bylo pomocí OpenGL počítat teselaci budov. Teselace je proces vyplnění roviny pomocí jednoho nebo více geometrických tvarů bez překrývání nebo mezer [14]. V našem případě šlo o triangulaci půdorysů budov. Při pokusu vykreslení střechy bez teselace docházelo u objektů obsahujících konkávní úhly ke špatným výsledkům. Střecha vybočovala z hranic objektu. Díky teselaci lze tomuto neduhu zabránit. V průběhu vedoucí práce rozhodl, že jelikož se jedná o problematiku vizualizace, neměla by se nacházet na serveru u výpočtu dat, ale přímo v aplikacích mých kolegů.

5.1 Rozšířený model

Rozšířený model aplikace vychází ze základního modelu, který jsem popisoval v kapitole 3. Pro jednotlivé třídy, reprezentující dlaždice a entity, bylo potřeba napsat třídy, které by obstarávaly získání požadovaných informací a jejich ukládání do databáze. Tento model lze vidět na ilustraci níže:



Obrázek 5: Rozšířený model aplikace

Tento class diagram již reprezentuje třídy v mé aplikaci. Nově se zde nacházejí *EntityDownloader, DMRDownloader, DBFactory* a *DroneDirectionEnum*. *DMRDownloader* jsem napsal ve spolupráci s panem Kubíčkem, neboť tato část práce se protíná s jeho prací. Do budoucna se totiž plánuje, že až bude vše dostatečně otestováno, tak se zmíněné downloadery a logika přesunou i do desktopové aplikace pana Kubíčka.

DMRDownloader slouží k získávání informací pro dlaždice. Kontaktuje tedy služby ArcGIS a naplní třídu *Tile*. Všechny dlaždice jsou nakonec uloženy v poli dlaždic. Tato třída dále umožňuje nastavit si velikost dlaždic, které nás zajímají. Ve výchozím stavu se zabýváme dlaždicemi o rozměru 128 × 128 metrů, ale do budoucna lze velikost libovolně měnit a na samotné logice se tak nic nezmění.

Obdobně funguje *EntityDownloader,* který kontaktuje službu OpenStreetMap a získá všechny entity a naplní tak třídu *Entity*. Všechny entity jsou ve výsledku ukládány do *ArrayListu*. Ten byl oproti poli zvolen z toho důvodu, že na rozdíl od počtu dlaždic nevíme předem, kolik každá dlaždic bude mít entit. Stejně jako v *DMRDownloaderu* si můžeme libovolně měnit velikost dlaždice. Na rozdíl od služby ArcGIS nevrací OpenStreetMap výsledky ve formátu JSON, ale nabídne ke stažení OSM soubor ve formátu

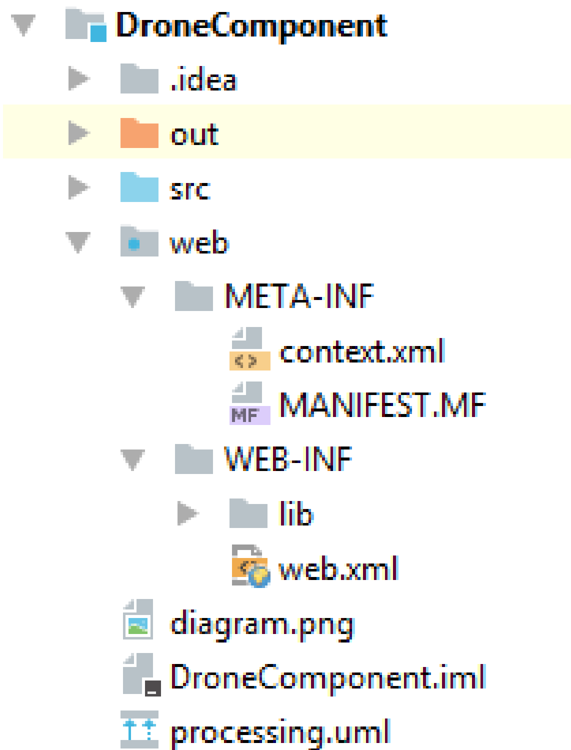
XML. Z toho důvodu je ve třídě atribut, umožňující zvolit si místo, kam se soubor stáhne.

DBFactory obstarává připojení do databáze a jsou v něm k dispozici veškeré SQL dotazy, které budou pro jednotlivé downloadery potřeba. *DBFactory* se podle zadaného zdroje dat připojí k databázi a udržuje s ní spojení. Aby oba downloadery mohly využívat ukládání do databáze a aby bylo možné využívat databázových transakcí, je instance třídy *DBFactory* rovněž předávána jednotlivým downloaderům.

Poslední novou položkou je *DroneDirectionEnum*. Jedná se o výčtový typ, obsahující všechny směry, kterými dron může letět, tedy sever, severovýchod, východ, jihovýchod, jih, jihozápad, západ a severozápad. Jednotlivé směry se využívají k predikci, jakou dlaždici bude dron požadovat v momentě, kdy vletí do neznámé oblasti.

5.2 Struktura aplikace

Pro psaní webových komponent se dodržují jisté konvence, týkající se adresářové struktury. Zpravidla se v kořenovém adresáři nachází adresář *src,* obsahující zdrojové kódy dané aplikace. Dále se v kořenovém adresáři nalézá složka *web* a uvnitř *META-INF* a *WEB-INF*. V mé aplikaci je ještě složka *out*, ve které najdeme zarchivovanou výslednou aplikaci ve formátu *WAR*. Náhled struktury se nachází na obrázku níže:



Obrázek 6: Základní adresářová struktura webové aplikace

Dle oficiálních specifikací Servletu verze 2.4 od společnosti Oracle [10] obsahuje *WEB-INF* soubory, které jsou neveřejné. Kontejner nesmí žádný z těchto souborů poskytnout uživateli. V kódu samotného servletu jsou však pomocí metod *getResources* a *getResourceAsStream* k dispozici instance *ServletContext*. V případě této aplikace se zde nachází složka s externími knihovnami, využívanými v celé aplikaci a popisovač nasazení *web.xml*.

5.2.1 Web.xml

Jak již bylo zmíněno, jedná se o popisovač nasazení a definuje se v něm například, jaké třídy se mají načíst, jaké parametry se mají nastavit do kontextu nebo jak zpracovávat příchozí požadavky.

Soubor obsahuje kořenový tag *web-app*, ve kterém je definice jmenného prostoru. Dále v něm konfiguruji servlet. Ten je pojmenovaný *DroneComponentServlet* a definuje třídu, která jej obsluhuje.

```
<servlet>
  <servlet-name>DroneComponentServlet</servlet-name>
  <servlet-class>
      org.glassfish.jersey.servlet.ServletContainer
  </servlet-class>
</servlet>
```

Zdrojový kód 3: Nastavení servletu

Tento servlet je potřeba namapovat na určitou URL. Klient pak má možnost k němu pomocí této adresy přistupovat. Toho lze docílit pomocí *servlet-mapping* elementu. V našem případě, jelikož se jedná o jediný servlet, jsem ho namapoval na kořenovou adresu.

```
<servlet-mapping>
    <servlet-name>DroneComponentServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

Zdrojový kód 4: Namapování servletu

Dále definuji přístup k prostředkům, konkrétně k databázi. To se nachází v tagu *resource-ref*, který obsahuje stručný popis, jméno, podle kterého se dá tento zdroj dohledat, a že se jedná o typ *DataSource*.

```
 <resource-ref>
    <description>
            DB storing map tiles and entities
      </description>
    <res-ref-name>jdbc/kozanekDrony</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

Zdrojový kód 5: Nastavení přístupu k prostředkům

5.2.2 Context.xml

Tento soubor se nachází v adresáři *META-INF* a obsahuje již konkrétní údaje o přístupu k databázi. Obsahuje kořenový element *Context* a v něm *Resource*, jehož atributy jsou právě údaje potřebné k připojení do databáze. Stejně jako v souboru *web.xml* obsahuje jméno a typ. Dále uživatelské jméno a heslo pro připojení do databáze, URL adresu, na které je databáze dostupná, a driver, který se bude pro spojení používat. Já pro

komunikaci s databází využívám JDBC neboli Java Database Connectivity, což je API pro programátory v Javě definující jednotné rozhraní pro přístup k relačním databázím [11]. Tento ovladač byl stažen ze stránek MySQL, jmenuje se Connector/J 5.1.42 a byl nainstalován do složky *lib* v adresáři *WEB-INF*.

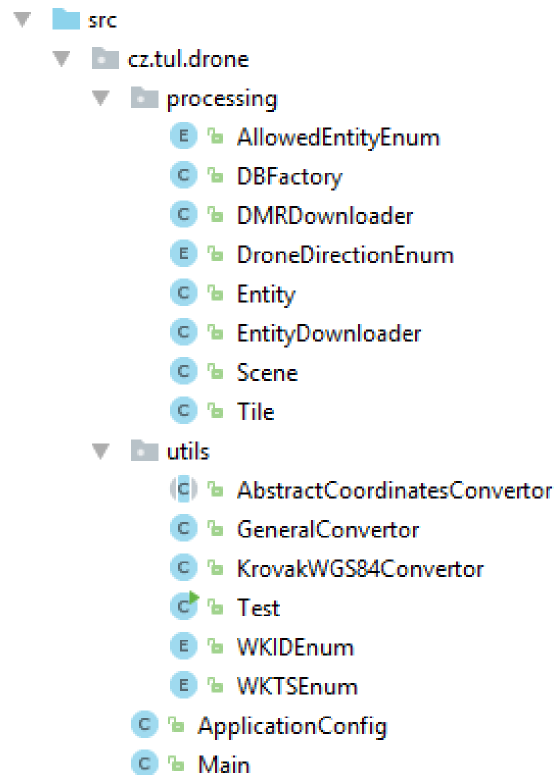
Jelikož prvotní výpočty dlaždic dlouho trvají a všechny výsledky ukládám do databáze pomocí transakce, přidal jsem do konfiguračního souboru validační dotaz, který ověří, že spojení je stále aktivní a nevyprší. Jedná se o jednoduchý dotaz, který vždy vrací hodnotu 1.

```
<Context>
    <Resource
         name="jdbc/kozanekDrony"
         auth="Container"
         type="javax.sql.DataSource"
         username="uname"
         password="********"
         driverClassName="com.mysql.jdbc.Driver"
         url="jdbc:mysql://drone.fm.tul.cz:4443/kozanek"
         maxIdle="3"
         validationQuery="select 1 as dbcp_connection_test"
         testOnBorrow="true"
    />
</Context>
```

Zdrojový kód 6: Údaje pro přístup k databázi

5.2.3 Adresář src

Ve složce *src* se již nacházejí klasické Java balíčky a v nich jednotlivé třídy. První je kořenový balíček *cz.tul.drone*, který obsahuje dvě třídy. *Main* a *ApplicationConfig*. Uvnitř tohoto balíčku jsou obsaženy další dva balíčky. Konkrétně *utils*, ve kterém se nachází knihovny potřebné pro převod souřadnic mezi soustavami S-JTSK a WGS-84, a dále balíček *processing,* obsahující hlavní logiku aplikace. Všechny tyto třídy v následujících kapitolách důkladně popíšu.



Obrázek 7: Struktura balíčků

5.3 Převod souřadnic

V balíčku *utils* se nacházejí třídy pro konverzi mezi soustavami S-JTSK a WGS-84. Převody jsou potřeba, protože služby ArcGIS pracují se soustavou S-JTSK, kdežto služby OpenStreetMap využívají WGS-84. Stejně tak i zařízení, pro která se tato aplikace píše, vracejí svoji polohu ve WGS-84. Je také potřeba mít možnost posouvat body po metrech do určitého směru, což by ve WGS-84 bylo velice složité.

Pro převod bylo potřeba využívat veřejně dostupných knihoven ArcGIS. Tento poskytovatel má na svých stránkách pro vývojáře dostupné ArcGIS SDK 10.2.4 pro platformu Java. Balíček pro převod souřadnic byl implementován a poskytnut vedoucím práce.

5.4 ApplicationConfig

Aby aplikace mohla fungovat, bylo potřeba přidat třídu se službami do množiny tříd. Tato třída rozšiřuje třídu *Application* a bylo potřeba překrýt metodu *getClasses()*. Ve třídě jsem vytvořil *HashSet*, do kterého jsem přidal třídu *Main.class* obsahující použí-

vané služby. Nad názvem třídy se nachází anotace @*ApplicationPath* s prázdným řetězcem uvnitř. Zde je možné definovat název služby, který by se zadával do URL při volání metod. Vzhledem k tomu, že máme pouze jednu službu pro drony, není potřeba ji pojmenovávat. URL díky tomu bude také kratší a přehlednější.

5.5 Použité externí knihovny

5.5.1 Gson

V aplikaci často pracuji s formátem JSON. Ten jsem potřeboval nějakým způsobem zanalyzovat a jednotlivé hodnoty přiřadit proměnným. Proto jsem zvolil knihovnu od společnosti Google jménem Gson ve verzi 2.8.0 [12].

Tuto knihovnu jsem zvolil kvůli její jednoduchosti a veliké rozšířenosti mezi programátory. Poskytuje dvě velice jednoduché metody, které se postarají o většinu práce. Jsou jimi *toJson()* a *fromJson()*.

Metoda *fromJson()* se používá na převod řetězce ve formátu JSON do požadované datové struktury. Pokud například tělo POST metody obsahuje dvojrozměrné pole, lze do něj velice jednoduše tento řetězec převést. První parametr této metody přebírá proměnnou a druhý třídu, podle které má Gson JSON převést. V případě dvojrozměrného pole floatů by stačilo zavolat *fromJson(json, float[][].class)*. Obecně tedy lze říci, že JSON převede do jakéhokoliv objektu, včetně těch, které si uživatel napsal sám *(MyType target = gson.fromJson(json, MyType.class))*.

Metoda *toJson()* přebírá jediný parametr, kterým je proměnná, jíž chceme převést do formátu JSON. Opět lze převádět jak jednoduché datové typy, tak i složité objekty. Gson zachová přesnou strukturu, jakou objekt měl. V případě, že objekt obsahuje atributy, které nechceme, aby zařadil do serializace, stačí tyto atributy označit klíčovým slovem *transient*. Takto označené proměnné se chovají naprosto stejně jako jakékoliv jiné, jen jsou při serializaci ignorovány.

Alternativou pro metodu *fromJson()* je použití třídy *JsonParser* rovněž z balíku Gson. Ten při parsování vytvoří strom třídy *JsonElement*, který se pak velice snadno prochází a dotazuje se na něj. *JsonElement* lze dle libosti přetypovat na jakoukoliv proměnnou. Stačí si vyhledat prvek pomocí metody *get()*, do které se jako parametr napíše název klíče ve formátu JSON, a následně zavolat jeho metodu *getAs[Type]*, kde za

[Type] můžeme dosadit libovolný datový typ. Lze tedy například získat float voláním *getAsFloat()* nebo celé číslo voláním *getAsInt()*. Tato volání lze v kombinaci s dalšími *get()* voláními řetězit a postupovat hlouběji do stromu. Níže lze vidět ukázku takového kódu:

```
JsonParser jp = new JsonParser();
JsonElement root = jp.parse(json);
double x = root.getAsJsonObject().get("x").getAsDouble();
int width = root.getAsJsonObject().get("width").getAsInt();
```

Zdrojový kód 7: Použití metod knihovny Gson

5.5.2 Jersey

Další potřebnou knihovnou byla knihovna Jersey. Jedná se o otevřený framework RESTful webových služeb implementující specifikaci JAX-RS neboli Java API for RESTful Web Services. Tato knihovna usnadňuje vývoj webových služeb v Javě.

JAX-RS používá anotace, které byly přidány do Javy SE verze 5. Obsahuje anotace, díky kterým se mapují jednotlivé metody na URL a dále anotace, pomocí kterých lze přiřadit proměnné v URI do proměnných v Javě. Níže se nachází jejich zjednodušený seznam:

Tabulka 1: Anotace pro mapování metod

Anotace	Popis
@Path	Specifikuje relativní cestu k metodě
@GET, @PUT, @POST, @DELETE	Specifikuje typ HTTP požadavku
@Produces	Specifikuje typ odpovědi
@Consumes	Specifikuje typ žádosti

Tabulka 2: Anotace pro získání parametrů metod

Anotace	Popis
@PathParam	Získá hodnotu z proměnné v adrese
@QueryParam	Získá hodnotu z dotazu v adrese
@FormParam	Získá hodnotu z formuláře

5.6 Využití anotací a návratový typ metod

Ve třídě *Main* se nacházejí všechny metody, které jsou veřejně dostupné. Třída obsahuje anotaci *@Path(„/service“)*, kterou jsem nastavil jako řetězec „service“, aby bylo jasné, že se jedná o webovou službu. Ten je nyní součástí celé URL adresy.

Každá metoda obsahuje několik anotací, které popisují, na jaký typ požadavku bude metoda reagovat a jaké datové typy bude vracet. Lze využít jakýkoliv ze čtyř požadavků REST služby (GET, POST, UPDATE, DELETE), nicméně ve své práci využívám pouze GET a POST.

Dále anotace @*Path* nastavuje mapování na danou metodu. Do těchto mapování lze vložit také proměnné. Stačí je vložit do složených závorek (*{}*). Do URL adresy pak stačí zadat název této metody doplněný o proměnné. Tyto proměnné se pak musí zachytit v parametru metody pomocí anotací @*PathParam*, do které se zadá název proměnné a za anotaci se napíše již klasická deklarace proměnné v Javě. Pro dotazy typu POST se nepoužívají proměnné v URL, protože veškeré informace se odesílají v těle požadavku. V tomto případě má metoda jako parametr String. Ten je potřeba analyzovat a přiřadit data do správných proměnných.

Poslední anotace @*Produces* říká, jaký typ odpovědi metoda vytváří. Ve své aplikaci používám dva různé typy odpovědí. Pro JSON stačí zadat *MediaType.APPLICATION_JSON* a pro XML *MediaType.APPLICATION_XML*. Obdobně se používá u POST požadavků i anotace @*Consumes*, do které se rovněž napíše, jaký formát má metoda přijímat. Opět je na výběr buď JSON nebo XML.

Každá metoda má jako návratový typ *Response*, ve kterém lze nastavit návratový kód (200, 404, 500 atd.) a v případě kódu 200 ještě nastavit v hlavičce odpovědi, jestli se jedná o JSON nebo XML. Spolu s touto odpovědí lze odeslat entitu, která uživatele zajímá. V mém případě je to vždy proměnná s výsledkem výpočtu. Příklad takto anotované metody lze vidět v kódu níže:

```
@GET
@Path("/SJTSKToGps/{x}/{y}")
@Produces(MediaType.APPLICATION_JSON)
Public Response SJTSKToGps(@PathParam("x") double x, @PathParam("y")
                              double y) {…}
```

Zdrojový kód 8: Anotace metody

5.7 Seznam metod

V tabulce níže se nachází kompletní seznam všech dostupných metod. Lze v ní vidět názvy metod, na jakou adresu jsou namapovány a jaké parametry přebírají. Všechny tyto metody jdou detailně rozebrány v následujících kapitolách.

Tabulka 3: Seznam metod

Název metody	URI	Dotazovací metoda	Datové typy
SJTSKToGps	SJTSKToGps/x/y	GET	x – INT y – INT
GpsToSJTSK	GpsToSJTSK/x/y	GET	x – INT y – INT
multipleGPSToSJTSK	multipleGPSToSJTSK	POST	x – INT y – INT
getTilesAround	getTilesAround/x/y/direction	GET	x – INT y – INT direction – String
getTilesAroundMap	getTilesAroundMap/x/y/num	GET	x – INT y – INT num – INT
getTilesMxN	getTilesMxN/x/y/width/height	GET	x – INT y – INT width – INT height – INT
getTilesMxNPost	getTilesMxNPost	POST	x – INT y – INT width – INT height – INT movementMode – String movementPoints – String

První tři metody jsem napsal, aby ulehčily práci s převody ostatním, kteří by chtěli službu využívat. Jelikož převodník souřadnic je napsán v Javě a například webová aplikace pro sledování dronů je v Javascriptu a Pythonu, zveřejnil jsem metody pro jednoduchý převod jednoho a více bodů.

5.7.1 Metody pro převod souřadnic

Metoda *SJTSKToGps* reaguje na požadavek typu GET a je namapovaná na URI „*SJTSKToGps*/{x}/{y}“. Za *x* a *y* se dosadí kartografické body S-JTSK. V metodě vytvářím instanci převodníku z Křovákova zobrazení do WGS-84. Vytvořím si nový bod, který budu vracet, a zavolám na něj metodu *convert(),* která přijímá jako parametry body *x* a *y*. Body v soustavě S-JTSK ještě zaokrouhluji, protože by jinak způsobovaly nepřesnosti. Metoda vrací výsledek ve formátu JSON.

Prakticky stejně funguje metoda *GpsToSJTSK*, která ale převádí GPS souřadnice do S-JTSK. Jediným rozdílem je to, že se volá metoda *convertBack()*.

Obě tyto metody převádějí jeden jediný bod. Pro převod více bodů najednou lze použít metodu *multipleGpsToSJTSK*. Ta reaguje na požadavek typu POST a v těle této metody se nachází pole bodů ve tvaru $[[x_1, y_1], [x_2, y_2], ...]$. Toto pole se pomocí knihovny Gson převede na dvojrozměrné pole reálných čísel s pohyblivou řádovou čárkou a následně se celé projde v cyklu a postupně převede do souřadnic S-JTSK. Výsledek je vrácen ve formátu JSON ve stejném tvaru, v jakém byl příchozí požadavek.

5.7.2 Metoda využívaná experimentálním zařízením

Zbylé čtyři metody sdílí stejnou logiku. Liší se pouze tím, kolik dlaždic počítají, případně do jakého směru se budou počítat. Experimentální zařízení využívají metodu *getTilesAround*.

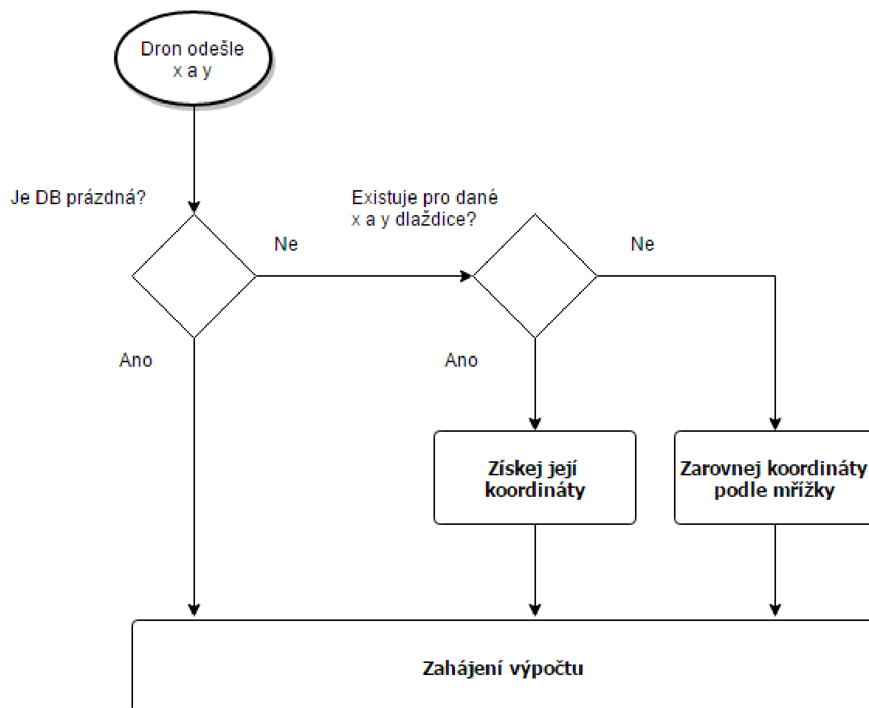
Serverová komponenta bude kontaktovat mou webovou službu v momentě, kdy bude vědět, že mu brzy dojdou jeho mapové podklady. V tu chvíli odešle souřadnice určující jeho aktuální polohu a podle vektoru pohybu také světovou stranu, jejímž směrem se pohybuje.

Souřadnice nejprve převádím do S-JTSK a zaokrouhlím na celá čísla, aby se s nimi lépe počítalo. Jelikož se dron v momentě volání bude nacházet v již vypočtené dlaždici, musím podle směru letu dopočítat, která dlaždice bude první, kterou bude potřeba do-

počítávat. K tomu využívám metody ve výčtovém typu *DroneDirectionEnum*. Ten obsahuje metodu pro každý možný směr letu. Jelikož jsou souřadnice převedené do S-JTSK, mohu v každé metodě podle příslušného směru přičítat nebo odečítat velikost dlaždice. Tím se posunu o jednu dlaždici ve směru letu/jízdy zařízení a budu počítat dlaždice v jejím okolí.

V budoucnu se počítá s přidáním dalšího parametru, který by zařízení posílalo se směrem jízdy/letu, a tím je rychlost. Momentálně se dá počet dlaždic v okolí nastavit ručně, ale počítá se s tím, že v budoucnu by se v závislosti na rychlosti zařízení tento počet dynamicky měnil.

Předtím, než na nově získané souřadnice začneme získávat všechny informace, musí proběhnout několik ověření. Je potřeba zjistit, jestli se pro hledané koordináty v databázi již nenachází záznam. Pokud ano, použijí se souřadnice této dlaždice. Pokud ne, musí se souřadnice zarovnat do mřížky. Kdybych totiž uložil data pro současné souřadnice, mohly by se nově vypočtené dlaždice překrývat s již existujícími a v databázi by nastala redundance dat. Tento postup je znázorněn na schématu níže:



Obrázek 8: Úprava souřadnic

Informaci o tom, zdali se daná souřadnice nachází v databázi, zjišťuji pomocí jednoduchého SQL dotazu. Hledám takovou dlaždici, jejíž souřadnice *x* se nachází v intervalu ohraničeném velikostí dlaždice.

```
SELECT X, Y FROM Tile WHERE
(X <= ? AND X + 128 > ?) AND (Y >= ? AND Y - 128 < ?)
```

Zdrojový kód 9: Hledání dlaždice pro zadaný bod

Za otazníky v dotazu se dosadí právě zkoumané proměnné. Pokud taková dlaždice existuje, budou její souřadnice *x* a *y* dále použity pro výpočet. V opačném případě dojde k zarovnání bodu do mřížky k levému hornímu okraji.

Zarovnání je docíleno tak, že se z databáze zjistí nejnižší a nejvyšší souřadnice *x* a *y*. Tyto hodnoty jsou krajními hodnotami v celé již existující mapě. Následně se zjišťuje, jestli se náš hledaný bod nachází pod nejmenší částí nebo nad nejvyšší částí. Zjišťuje se to z toho důvodu, abychom vždy vybrali nejkratší cestu k hledanému bodu. Když víme, kudy je to k bodu nejblíže, začne se k němu postupně přičítat, případně odečítat, velikost dlaždice do té doby, než se dostaneme k našemu hledanému bodu. Jakmile jej překročíme, tak víme, že se nacházíme ve správné dlaždici a můžeme použít její levý horní roh. Může samozřejmě nastat i případ, kdy se bod nachází mezi již několika vytvořenými dlaždicemi a není tedy ani nad maximem, ani pod minimem. V tomto případě zůstává princip stejný, jen se najde opět nejbližší dlaždice z dané strany a vybere se nejkratší cesta. Tento proces se vykonává jak pro bod *x*, tak pro bod *y*.

V případě, že v databázi nebude žádná dlaždice, lze využít dvou možností. Buď neproběhne žádná korekce souřadnic a první dlaždice vytvoří pomyslnou mřížku, podle které se ostatní dlaždice budou zarovnávat, nebo se na začátku definuje oblast, podle které vznikne mřížka. Pro druhý případ jsem napsal ještě metodu, ve které je zadaná hranice České Republiky a od této hranice se vždy bude mřížka přepočítávat. Tím se docílí toho, že i kdyby z nějakého důvodu došlo ke smazání databáze, tak se vytvoří identická mřížka.

5.7.3 DBFactory

Před provedením výpočtu je potřeba připojit se do databáze. Bude totiž potřeba kontrolovat, které dlaždice a entity jsou již vypočtené a ty které nejsou, abychom je mohli

uložit. Připojení je uskutečněno ve třídě *DBFactory* v konstruktoru. Jelikož všechny údaje pro připojení se nacházejí v konfiguračních XML souborech popsaných dříve, stačí vytvořit počáteční kontext pomocí třídy *InitialContext* a zavolat na jeho instanci metodu *lookup()* s parametrem *„java:comp/env/jdbc/kozanekDrony".* Tento název jsem taktéž definoval v konfiguračních souborech, proto jej stačí zavolat a JDBC se postará o získání potřebných přihlašovacích údajů. Tato metoda vrací *DataSource*, na který lze zavolat metodu *getConnection()*, čímž dojde k připojení k databázi. *GetConnection()* vrací datový typ *Connection*, který je uložen jako atribut třídy a který lze pomocí getteru volat mimo tuto třídu.

Do databáze ukládám vždy dlaždici a entity v ní se nacházející. Vzhledem k tomu, že dlaždice bez entit a entita bez dlaždice nenese žádnou informační hodnotu, musím se ujistit, že se vždy uloží do databáze spolu. Toho jsem docílil pomocí databázové transakce. Na začátku celé služby vytvořím instanci této třídy, getterem získám spojení a na něj zavolám metodu *setAutoCommit()* s parametrem *false*. Tím se vypne automatické potvrzení SQL dotazů po jejich vykonání. Všechny SQL dotazy se potvrdí najednou, až když na konci, po skončení všech výpočtů, zavolám metodu *commit()*. Výhodou transakcí je také možnost použít *rollback()*, který dokáže v případě chyby vrátit databázi do stavu před zahájením transakce. K tomu je ale potřeba si mezi metodami posílat instanci *DBFactory*, protože jinak by transakce nikdy neproběhla.

Všechny databázové dotazy jsou vykonávány pomocí *PreparedStatements*. Jedná se o třídu v balíčku *java.sql*, která umožňuje vykonávat SQL dotazy pomocí JDBC. Tento objekt se volá pomocí metody *connection.prepareStatement().* Dotaz je poslán do databáze, kde je předkompilován a je mu nastaven přístupový plán. Později jsou do něj přidány parametry. To je výhodné, pokud jeden a ten samý dotaz voláme mnohokrát, pouze s jinými parametry, což se v mé aplikaci děje často. V opačném případě by dotaz musel být pokaždé databází znovu parsován a znovu nastavován přístupový plán, což prodlužuje dobu volání.

Druhou výhodou je ochrana proti SQL injection neboli podsunutí kódu do neošetřeného vstupu a vykonání tak jiného příkazu. Volání očekává parametry se správnými datovými typy, které se nastavují pomocí příslušných setterů jako například *setInt(), setFloat()* atd.

5.7.4 Získání DMR dat

Na částech tohoto kódu proběhla spolupráce s panem Kubíčkem, který stejnou část bude využívat v jeho desktopové aplikaci.

Konstruktor třídy *DMRDownloader* přebírá tři parametry. První je instance databáze z důvodu *cachování* a další dva jsou adresy odkazující na server ArcGISu, ze kterých se postupně získají DMR data a textura dlaždice. Konstruktor zajišťuje oříznutí lomítka na konci zadané adresy, pokud je přítomno.

Po vytvoření instance třídy jsem na ni zavolal metodu *downloadDMR()*, čímž se stáhnou všechny potřebné dlaždice. Metoda přebírá jako parametry koordináty x a y v soustavě S-JTSK. Rovněž také přebírá počet dlaždic, který chceme dopočítat okolo námi hledané dlaždice. Tato metoda přetěžuje stejnojmennou metodu, která však místo počtu dlaždic okolo té hledané vyžaduje počet vertikálních a horizontálních dlaždic, které má počítat, a rovněž zadanou dlaždici nepovažuje za středovou, ale za levou horní. Souřadnice se tedy posunou do levého horního rohu a proměnná s počtem dlaždic okolo se podle vzorce $2x + 1$, kde x je počet dlaždic v okolí, převede na počet horizontálních dlaždic a podle stejného vzorce i na počet vertikálních dlaždic.

Jelikož znám přesný počet dlaždic, které ve výsledku dostanu, mohu vytvořit pole dlaždic pevné délky. Následně pomocí dvou vnořených cyklů procházím všechny dlaždice. Nejprve zjistím, jestli se v databázi hledaná dlaždice již nenachází. Pokud ano, použiji informace z databáze a přiřadím je objektu *Tile*, který přidám do výsledného pole. Pokud žádná taková dlaždice v databázi neexistuje, začnu s jejím vytvářením. K tomu slouží metoda *makeDMRTile()*, které předám souřadnice hledané dlaždice.

Tato metoda stahuje ze služby DMR 5G poskytovatele ArcGIS výškové profily dlaždice. Služba tohoto poskytovatele se nazývá *GetSamples* a umožňuje získat výšky pro oblast vymezenou množinou bodů. Nevýhodou je, že lze na jeden dotaz zadat pouze tisíc bodů. Pokud tedy chceme získat celý vzorek, je nutné jeden dotaz rozdělit na pět, neboť je potřeba celkem 4096 vzorků. Tělo dotazu může vypadat například následovně:

```
geometry={"points": [-900128,-1000000],[-900000,-1000128], …}
&geometryType=esriGeometryMultipoint&f=json
```

Zdrojový kód 10: Tělo dotazu pro získání DMR

Celá množina bodů je vypsaná v atributu *geometry* a rovněž je specifikováno, že se jedná o *Multipoint* a že výsledek má být vrácen ve formátu JSON. Jakmile je tento řetězec zkonstruován, je předán metodě *downloadTileData()*, kde je pomocí POST metody odeslán do služby ArcGIS. POST metoda je využita, protože tak rozsáhlé množství bodů, které posíláme, se nevejde do limitů URL adresy. Výsledek, který je vrácen ve formátu JSON, se pomocí knihovny Gson zkontroluje a výšková data se postupně přidají do dvojrozměrného pole. Spolu se zaznamenává minimální a maximální hodnota výšky. Celá dlaždice je po pěti dotazech kompletně načtena. Mě z výstupu zajímá hodnota klíče *„value"*. To je mnou hledaná výška v daném bodě. Níže se nachází ukázka výstupu služby ArcGIS:


```
{"samples":
    [{
      "location":{
         "x":-686080,
         "y":-983552,
         "spatialReference":{…}
      },
      "locationId":0,
      "value":"397.556671143",
      "rasterId":1,
      "resolution":2
    }, …
    ]}
```


Zdrojový kód 11: Výstup ze služby getSamples

Poslední metodou v této třídě je *downloadTexture()*. Ta stahuje texturu pro danou dlaždici. Ty jsou využívány pro vizualizaci dlaždice v uživatelském prostředí. ArcGIS poskytuje funkci *ortofoto*, která pro zadané koordináty, udávající obdélníkový výběr, vrátí danou texturu. Práce s touto funkcí je prakticky totožná jako u výškových bodů. Při exportu je možné si zvolit formát a velikost obrázku. Zvolen tedy byl formát PNG a velikost 512 × 512 pixelů. Stažený obrázek je zakódován pomocí Base64 do řetězce a přiřazen k dlaždici. Tím máme získané všechny potřebné informace o dlaždici.

5.7.5 Získání entit

Spuštění stahování entit probíhá stejně jako u stahování dlaždic. Vytvoří se instance *EntityDownloaderu*, které se spolu s instancí databáze předá cesta, do které se budou

ukládat OSM soubory. Služba OpenStreetMap totiž na rozdíl od ArcGISu nevrací výsledky ve formátu JSON v prohlížeči, ale v jejich vlastním souboru OSM, který má formát XML. OSM soubory ukládám do adresáře Tomcatu pro dočasné soubory, přičemž každému souboru dávám pomocí časového razítka unikátní název.

Na tuto instanci je následně zavolána metoda *downloadEntities()*, která jako parametry přebírá souřadnice v S-JTSK spolu s počtem dlaždic v okolí a stáhne všechny potřebné entity. Stejně jako u *DMRDownloaderu*, i zde se nacházejí stejným způsobem přetížené metody. Lze tedy získat entity jak pro středovou dlaždici a její okolí, tak pro dlaždici s definovaným vertikálním a horizontálním počtem dlaždic. Začíná se tedy jako minule v levém horním rohu a zjišťují se všechny entity pro tuto dlaždici. Tímto způsobem se najdou entity pro každou další dlaždici.

Na rozdíl od dlaždic, kterých máme přesný počet, předem nevíme, kolik bude v jedné dlaždici entit. Z tohoto důvodu nevyužívám klasické pole entit, nýbrž *ArrayList*, který umí dynamicky realokovat svoji velikost. Nejprve si opět zjistím, zdali jsem již pro danou dlaždici nehledal entity. Pokud ano, získám je z databáze, přiřadím do objektu *Entity* a ty postupně přidám do *ArrayListu*. V opačném případě začne výpočet entit.

Metoda *getOSMData* ze souřadnic udělá obdélníkový výběr, který následně převede ze S-JTSK na WGS-84. Služba OpenStreetMapy totiž na rozdíl od ArcGISu pracuje s tímto formátem. Na volání služby se použije následující adresa doplněná o právě převedené souřadnice:

```
https://api.openstreetmap.org/api/0.6/map?bbox=50.732700578686625,15.72833
3810228309,50.731695101549406,15.730349430087132
```

Zdrojový kód 12: Adresa volající službu OpenStreetMap

Výstupem tohoto volání je XML soubor. Na začátku souboru se nachází seznam všech bodů, používaných ve zbytku dokumentu a jejich vlastností, jako je id, typ, gps poloha a kdo tento bod vytvořil. Po jejich výpisu následují již jednotlivé budovy, které jsou uzavřeny v tagu *<way>* a jsou utvořeny z referencí na již zmíněné body. Jednotlivé entity jsou popsány pomocí klíčů, které jsou zdokumentované na wiki stránkách OpenStreetMap [13]. Entity jsou rozděleny na více než dvacet různých typů, přičemž každý má několik dalších podtypů. Patří mezi ně například budovy, silnice, překážky, lesy, historické památky apod. Pro potřeby tohoto projektu jsme se rozhodli pracovat zatím

pouze s budovami a rostlinným pokryvem. Níže se nachází ukázka onoho XML souboru:

```
<osm version="0.6">
   <bounds minlat="50.0788734" minlon="14.4090110" maxlat="50.0801706" ma-
xlon="14.4105387"/>
   <node id="4749441936" visible="true" version="2" changeset="8473634"
user="Washek" uid="245252" lat="50.0833690" lon="14.4099639"/>
   …
   <node id="4749441934" visible="true" version="3" changeset="8472134"
user="Washek" uid="245252" lat="50. 0794121" lon="14. 4104440"/>

   <way id="482067463" visible="true" version="1" changeset="47076901"
user="fell3" uid="1643116">
     <nd ref="4749441936"/>
     <nd ref="4749441935"/>
     <nd ref="4749441934"/>
     <nd ref="4749441933"/>
     <nd ref="4749441932"/>
     <nd ref="4749441931"/>
     <tag k="highway" v="footway"/>
   </way>
   …
</osm>
```

Zdrojový kód 13: Výstup služby OpenStreetMap

Na zpracování OSM souboru se používá metoda *processOSMFile()*. XML soubory se v Javě procházejí poměrně snadno. Používám na to třídu *DocumentBuilder*, která soubor přetransformuje do stromové struktury, která se jednodušeji prochází. OSM soubor je třeba procházet v několika krocích. Nejprve v metodě *getNodesofEntities()* vyberu všechny entity, které jsou pro experimentální zařízení potřebné, a uložím je do *ArrayListu* prvků (potřebné entity jsou vypsány ve výčtovém typu *AllowedEntityEnum*).

Následně jsem v metodě *makeEntityObjectsFromEntityXML()* vytvořený *ArrayList* prošel a postupně z každého XML prvku vytvořil entitu. Při vytváření jsem si ke každému objektu uložil jeho reference z OSM souboru, abych podle nich později mohl dohledávat souřadnice. Při prvotním průchodu tedy získám id, typ a referenci entity.

V další metodě získám postupně půdorysy. Nejprve projdu znovu OSM soubor, avšak tentokrát si ukládám pouze seznam bodů na začátku souboru. Následně iteruji všemi entitami v mém *ArrayListu* a pro každou referenci u objektu hledám referenci v seznamu bodů. Jakmile ji najdu, přečtu si její polohu ve WGS-84, převedu ji do S-JTSK a uložím do dvojrozměrného pole všech bodů, dokud nemám u jedné entity

kompletní půdorys. Do entity zároveň ukládám i dvojrozměrné pole WGS-84 poloh, takže objekt je možné vypisovat s libovolnými souřadnicemi.

Poslední údaj o budově, který nám chybí, je nejvyšší a nejnižší výška entity. Data služby OpenStreetMap bohužel tyto údaje neobsahují, a proto jsem se musel obrátit na službu ArcGIS. Ta disponuje metodou *computeStatisticsHistograms*, která pro zadaný polygon spočítá statistiky, mezi kterými jsou právě i mnou hledané výšky. Metoda *getHeightForEntity()* pro danou entitu projde pole souřadnic v S-JTSK a postupně pomocí *StringBuilderu* vytvoří řetězec reprezentující polygon, který přijímá webová služba ArcGIS. Samotné volání probíhá v metodě *getHeightFromArcGIS()*. Na každou entitu proběhnou celkem dvě volání služby. První na DMP server, který poskytuje model povrchu, včetně staveb a rostlinného pokryvu, a zjistíme z něj maximální výšku daného polygonu. Pro minimální výšku byla použita stejná metoda, ale na serveru DMR 5G. Díky tomu mohu získat přesně výšku, ve které se nachází zemský povrch, neboť DMR neobsahuje budovy. Bohužel ani ArcGIS není schopný správně vypočítat výšku jakéhokoliv objektu. Měření výšek pro ArcGIS bylo provedeno letecky, kdy každé 2 metry byl spuštěn laser, který změřil výšku. Při testování jsem zjistil, že pokud je objekt dostatečně malý a trefil se mezi právě změřené dva metry, tak vrátil chybu, protože neměl údaje pro tyto body. Z tohoto důvodu v případě chyby nastavuji výšku objektu na 999999 metrů. Objekt jako takový nás stále zajímá a zařízení se mu může vyhnout, nicméně byl by špatně vykreslen. Naštěstí takovýchto objektů jsem v rámci testování objevil naprosté minimum a jedná se tedy spíše o velice vzácné výjimky. Tento proces zjišťování výšek se aplikuje na každou entitu v *ArrayListu* v metodě *findHeightsToEntities()*. Níže lze vidět ukázku výstupu DMR a DMP služby ve formátu JSON. U DMP mě zajímala hodnota *max* a i DMR hodnota *min*.:


```
{
 "statistics": [{
        "min": 382.29998779296875,
        "max": 437.7799987792969,
        "mean": 411.6357262168554,
        "standardDeviation": 12.296443484403301,
        "count": 8274,
        "median": 410.36634629193475,
        "mode": 403.4041488348269
 }],
 "histograms": [{
        "size": 256,
        "min": 382.29998779296875,
```

```
            "max": 437.7799987792969,
            "counts": […]
    }]
}
```

Zdrojový kód 14: Výstup DMR a DMR služby

Tím mám o entitě všechny potřebné informace a mohu ji uložit do databáze. Spolu s tím ale musím uložit záznam o tom, v jaké dlaždici se nachází. Entita se totiž může vyskytovat na více dlaždicích najednou a pro každou dlaždici je potřeba vždy dostat tu samou entitu. Metoda *saveEntitiesToDB()* přebírá jako parametry seznam entit, počet entit, které jsem již uložil do databáze, souřadnice v zobrazení S-JTSK a velikost dlaždice. Parametr s počtem již uložených entit je velice důležitý, protože mám pro všechny dlaždice jen jeden *ArrayList* se všemi entitami dohromady. Díky němu tedy neiteruji celým *ArrayListem* pokaždé znovu, ale pokračuji tam, kde jsem skončil. Ukládání probíhá pomocí *prepareStatements* a používám hromadné uložení více dat najednou místo opakovaného volání. Pomocí dvou instancí třídy StringBuilder si vytvořím seznam prázdných parametrů (obsahuje pouze otazníky), který je vložen do připraveného SQL dotazu. Za otazníky jsou následně v cyklu dosazeny proměnné dané entity a příkaz je vykonán. Tím výpočet entit končí.

5.7.6 Scéna

Jakmile mám vytvořené všechny dlaždice a entity, je potřeba tyto objekty zapouzdřit a odeslat jako výsledek. K tomu slouží třída *Scene*, která obsahuje pole všech dlaždic a *ArrayList* entit. Metoda *makeScene()* tyto atributy naplní právě vypočtenými údaji z jednotlivých *Downloaderů.*

Nakonec dojde k vykonání transakce pomocí příkazu *db.getConn().commit()* a následně se spojení s databází uzavře příkazem *db.getConn().close()*. Poté už se jen rozhodne, v jakém formátu budou data odeslána uživateli. Pokud uživatel neřekne, o jaký formát má zájem, dostane ve výchozím stavu formát JSON. Pokud se rozhodne přidat do adresy *&format=xml*, dostane výstup ve formátu XML.

Převod do formátu XML je dosažen pomocí anotací poskytnutých balíčkem *javax.xml.bind.annotation*. Nad název třídy Scene stačilo přidat anotaci *@XmlRootElement* a jednotlivé její atributy označit anotací *@XmlElement*. Tím jsem

třídu označil jako kořenovou a že chci vypsat její dva atributy. Podobně musím ale přidat anotace i do samotných tříd objektů *Tile* a *Entity*. U entity se nad třídu přidala anotace *@XmlAccessorType(XmlAccessType.FIELD)* a *@XmlType(propOrder={"id", "type", "maxZ", "minZ", "SJTSK"})*. Díky první anotaci se mi serializují všechny atributy, které nejsou statické nebo označeny modifikátorem *transient*. Druhá anotace určuje pořadí, v jakém budou jednotlivé atributy vypsány. Obdobně je tomu i ve třídě *Tile*.

5.7.7 Metody pro vizualizaci map

Další zveřejněné metody fungují obdobně jako ta, kterou využívá experimentální zařízení. Jádro logiky je stejné, pouze se liší zpracováním počátečních dat a voláním jiné z oněch několika přetížených metod.

Aplikace pro autonomní navigační systém pana Kubíčka využívá metodu *getTilesMxNPost*. Do ní je odesíláno více proměnných, které se již nehodí pro posílání pomocí URL adresy, proto tato služba odpovídá na metodu POST.

Pomocí knihovny Gson zanalyzuji příchozí JSON a jednotlivé hodnoty přiřadím do proměnných. Souřadnice odpovídají levé horní dlaždici, ne střední. S tím souvisí fakt, že místo počtu dlaždic v okolí získávám celočíselné hodnoty *width* a *height* znázorňující, kolik dlaždic na výšku a šířku od levého horního rohu chci počítat. Navíc ještě JSON obsahuje *movementMode*, ve kterém se specifikuje způsob pohybu zařízení. Lze nastavit předem určenou cestu pomocí bodů nebo například náhodný let. Pokud je cesta předem nastavená, body se nacházejí v dalším atributu *movementPoints*. Tyto atributy nejsou zatím využívány, jsou pouze koncepčně připraveny na budoucí implementaci. Níže lze vidět ukázku posílaného požadavku pro dvě dlaždice pod sebou a s náhodným pohybem:

```
{
"x":-743808.0,
"y":-1043584.0,
"width":1,
"height":2,
"movementMode":"RANDOM"
}
```

Zdrojový kód 15: Struktura požadavku pro metodu getTilesMxNPost

Posílané souřadnice se navíc nemusí už zarovnávat do mřížky. Přijdou z aplikace již zarovnané. To je potřeba z toho důvodu, že kdyby přišly informace nezarovnané a moje serverová komponenta je posunula, byly by informace mezi tím, co se nachází v desktopové aplikaci, a tím co odešle server, do určité míry posunuté. Rovněž by nesouhlasila trasa, kterou se má zařízení pohybovat. Při testování se nám stal nejhorší možný případ, kdy desktopová aplikace odeslala souřadnice, které začínaly v pravém dolním rohu dlaždice. Serverovým zarovnáním se posunuly o skoro celý rozměr dlaždice do levého horního rohu a vznikla nepřesnost zhruba 100 metrů.

Webová aplikace pro monitorování autonomních zařízení slečny Němečkové využívá metodu *getTilesAroundMap.* Ta pomocí metody GET přebírá souřadnice středové dlaždice a počet dlaždic do všech stran okolo ní. V této metodě již zarovnání do mřížky probíhá a lze tedy zvolit velikost mapy bez závislosti na rychlosti letu/jízdy zařízení. Výpočet dlaždic je identický jako u zbytku metod.

6. Testování aplikace

Aplikace byla testována několika způsoby. Prvotní testování probíhalo na lokálním stroji s databází. Proto jsem napsal metodu *getTilesMxN*(), ve které jsem si mohl zvolit libovolný počet dlaždic, které chci vypočítat. Metoda reaguje na požadavek typu GET, na rozdíl od POST, který běží na produkčním serveru.

Bylo potřeba otestovat správnost údajů, které vrací služby OpenStreetMap a ArcGIS. Bylo zjištěno, že při zaslání některých dat služby vracejí prázdné pole s odpovědí. K tomu dochází v několika případech. Prvním je, že vybraná zkoumaná oblast se nenachází na území České republiky. Jelikož tyto služby podporují pouze Českou republiku, při výběru ostatních lokací je vrácena chyba 500 s informací o tom, že daná lokace není zmapována.

Druhou chybou byla chyba výšek. Pro některé entity aplikace vyhazovala výjimku *nullPointerException*, protože služba opět vracela prázdné pole. To bylo zapříčiněno způsobem, jakým ArcGIS měřil výšky objektů. Model výškopisů vznikl metodou leteckého laserového skenování, které probíhalo každé dva metry [16]. Pokud se však podařilo nalézt budovu, která měla tak malý rozměr, že se vešla mezi tyto dva metry, služba nevrátila žádný výsledek. Tento problém jsem vyřešil nastavením velice vysoké výšky danému objektu. Na mapách nebude možné jej vizualizovat právě z důvodu velké výšky, ale v databázi uložen bude, protože se mu dron stále může vyhnout alespoň pomocí půdorysu.

Funkčnost celé webové aplikace byla vyzkoušena pomocí programu Postman, ve kterém si lze jednoduše zkoušet volání všemi metodami. Rovněž byla funkčnost ověřena díky desktopové aplikaci pro autonomní navigační systém pana Kubíčka a taktéž díky webové aplikaci slečny Němečkové pro monitorování a tracking autonomních zařízení. Jelikož v obou případech bylo možné data vizualizovat na mapě, měl jsem testování správnosti výpočtů jednodušší. I jen jedna dlaždice obsahuje enormní množství informací, že jej nelze pouze z dat všechny zkontrolovat.

Dále byla aplikace nasazena na školní server, aby se otestovala komunikace s dalšími klienty. Díky aplikaci slečny Němečkové bylo nalezeno několik chyb. V prvotní verzi aplikace se stávalo, že některé budovy nebyly korektně zasazené v půdě a vznášely se ve vzduchu. To bylo zapříčiněno nepřesností služby ArcGIS. Metoda *computeStatisticsHistogram* pomocí DMP sice počítala správně maximální výšku budov, ale minimální občas nesouhlasila. Bylo to opět způsobeno metodou měření. Ve většině případů

jsou budovy dostatečně velké na to, aby se jeden z laserů snímající výšku trefil mimo půdorys budovy. V tu chvíli byla naměřena výška země, na které budova stála, a vše se zdálo být v pořádku. Nicméně v momentě, kdy se lasery trefily všechny přesně do půdorysu budovy (což je také ideální stav), tak minimální výška se rovnala té maximální a budova se následně vznášela ve vzduchu. Řešením bylo pro zjištění minimální výšky místo DMP měření použít DMR. V DMR se nenacházejí žádné budovy a mám tak přesně přehled o minimální výšce budovy.

Dalším nalezeným problémem byla redundance entit v databázi, když se načítalo pro jednu dlaždici více entit, které v ní neměly být. Jelikož jsou všechny entity všech dlaždic uloženy v jednom výsledném *ArrayListu* a postupně se do něj přidávají další, tak nastal problém s ukládáním dat. Ukládání probíhalo průchodem onoho celého *ArrayListu* a uložením každého prvku v něm. Jelikož byly všechny entity ukládány rovnou do výsledného pole, tak pokud první dlaždice měla například 15 entit, které jsem již uložil a druhá 10, při dalším ukládání entit do databáze, se oněch prvních 15 entit uložilo znovu a vznikaly duplikáty. Jedním z možných řešení bylo mít dvě pole. Jedno na ukládání mezivýsledků
a druhé sloužící jako seznam všech entit, které počítám. Zbytečně bych ale vytěžoval paměť dalším velkým polem, tak jsem jako řešení zvolil vytvoření jedné celočíselné proměnné, která počítala, kolikátou entitu jsem uložil jako poslední. Poté jsem již neiteroval celý ArrayList entit od začátku, ale od již zmíněné proměnné.

Při testování aplikace u desktopové aplikace bylo zjištěno, že původní zarovnávání souřadnic do mřížky s ní nebylo kompatibilní. V desktopové aplikaci totiž dochází k výběru oblasti přetáhnutím myši a vytvořením obdélníkového výběru. Když se ale uživatel špatně trefil do mřížky, serverová komponenta souřadnice zarovnala k levému hornímu rohu dané dlaždice, čímž v nejhorším případě posunula původní bod až o 120 metrů. Vrácené informace pak tedy vůbec nekorespondovaly s oblastí, kterou si uživatel vybral ke stažení. Proto se vytvořil obdélníkový okraj České republiky, podle kterého zarovnání probíhá už v desktopové aplikaci, a na server chodí již zarovnané souřadnice. Není třeba je tedy posouvat v mé aplikaci. Zároveň desktopová aplikace ví o tom, jak mřížka vypadá, a nemusí čekat až na odpověď serveru, aby to věděla.

7. Závěr

Cílem práce bylo vytvořit serverovou aplikaci, která by dle ad hoc požadavků zařízení aktualizovala jeho 3D model okolí. K tomu bylo potřeba navrhnout model a implementovat jej v jazyce Java za pomoci mapových podkladů služeb OpenStreetMap a ArcGIS. Nejprve jsem navrhnul strukturu informací, které server vrací. Ty je možné získávat ve formátu JSON nebo XML.

Dále jsem vytvořil databázi v systému MariaDB, která má strukturu již navrhnutého modelu. Do ní ukládám již vypočtené informace o dlaždicích a entitách. To je důležité z důvodu vysoké časové náročnosti na výpočty. Jelikož serverová aplikace hodněkrát kontaktuje služby třetích stran, je zpomalena o dobu, kdy čeká na jejich výsledek. Konkrétně standardní rozměr devíti dlaždic počítá přes dvě minuty. Při dotazu na stejné, již vypočtené, dlaždice se doba zkrátila na pět sekund.

Aplikace běží na školním serveru, na kterém je nainstalován a nakonfigurován aplikační server a servlet kontejner Tomcat. Ten se stará o vyřizování všech požadavků a o komunikaci s databází. Rovněž ze své podstaty se stará o to, aby na více požadavků najednou byla vytvořena na sobě nezávislá vlákna.

Jelikož pracuji s dvěma rozdílnými souřadnicovými systémy, bylo potřeba mít možnost mezi nimi souřadnice převádět. K tomu jsem využil SDK ArcGIS 10.2.4 a knihovnu, která implementuje metody tohoto SDK. Tuto knihovnu poskytl vedoucí práce.

Server komunikuje s několika cílovými zdroji. Prvním z nich je stolní aplikace pro autonomní navigační systém, ve které lze vybrat obdélníkovým výběrem lokaci, v níž se zařízení bude pohybovat. Po výběru dojde ke kontaktování této webové služby, která na základě poslaných dat pomocí metody POST vypočítá informace o všech dlaždicích ve výběru a o všech požadovaných entitách, které se v nich nacházejí. Pro výpočet půdorysů jednotlivých entit se používá API poskytovatele OpenStreetMap. Pro výpočet výšek těchto entit spolu s reliéfem a texturou krajiny, bylo použito REST API poskytovatele ArcGIS.

Dalším cílovým zdrojem je webová aplikace pro monitorování autonomních zařízení. Pro ni byla napsána metoda, umožňující stáhnout si dlaždici a její okolí. Okolí je možné určit celočíselnou hodnotou a reprezentuje počet dlaždic do všech směrů od té středové. Jelikož je vykreslování velkého množství dlaždic náročné, je ve výchozím režimu nastavena jedna dlaždice do všech směrů od středové, tedy devět dlaždic celkově. Pro dotazy je využívána metoda GET a výsledek je vracen ve formátu JSON. Velice podobnou

metodu využívá i samotné experimentální zařízení, které však navíc posílá směr pohybu. Aplikace rovněž poskytuje možnost převádění libovolného počtu souřadnic ze soustavy S-JTSK do WGS-84 a opačně.

Aplikace splňuje požadavky, které na ni byly kladené, byla naprogramována v řádném termínu a webová služba je konkrétně dostupná na adrese https://drone.fm.tul.cz:18443/DroneComponent/service. Při tvorbě aplikace jsem navíc myslel na její jednoduchou rozšiřitelnost. V budoucnu lze velice jednoduše například změnit rozlišení textury dlaždice nebo její velikost. Zatím aplikace pracuje s velikostí 128 × 128 metrů. Tomu je uzpůsobena i struktura databáze, kdy jednotlivé dlaždice jsou uložené pro daný rozměr. Lze tedy mít v databázi uloženou mapu pro dlaždice různých velikostí a libovolně se na ně dotazovat.

Dále se počítá s rozšířením aplikace o výpočet počtu dlaždic, které bude zařízení potřebovat v závislosti na rychlosti letu/jízdy. To bude možné odvodit až z delšího pozorování provozu. Momentálně se podle kompasu a gyroskopu na zařízení pozná, jakým směrem se pohybuje a pro něj se vypočte zatím konstantních devět dlaždic.

Finální testování mělo dle původního záměru probíhat na konkrétním fyzickém zařízení. Bohužel v době odevzdání práce ještě nebyl ani na jednom zařízení implementován klientský software. Nicméně testování proti zmíněným aplikacím prokázalo plně funkčnost navrhované serverové komponenty a lze předpokládat, že zařízení budou služby serveru využívat ve stejné míře, jako softwaroví klienti.

POUŽITÁ LITERATURA A ZDROJE

[1] KUBÍČEK, Ondřej. Aplikace pro autonomní navigační systém. Liberec, 2017. Diplomová práce. Technická univerzita v Liberci, Fakulta mechatroniky, informatiky a mezioborových studií. Vedoucí práce Igor Kopetschke.

[2] NĚMEČKOVÁ, Simona. Webová aplikace pro monitorování a tracking autonomních zařízení. Liberec, 2017. Diplomová práce. Technická univerzita v Liberci, Fakulta mechatroniky, informatiky a mezioborových studií. Vedoucí práce Igor Kopetschke.

[3] ArcGIS REST API. ArcGIS REST API [online]. Český úřad zeměměřický a katastrální, 2016 [cit. 2017-05-06]. Dostupné z: http://ags.cuzk.cz/arcgis/sdk/rest/index.html

[4] API v0.6 – OpenStreetMap Wiki. OpenStreetMap Wiki [online]. [2017] [cit. 2017-05-06]. Dostupné z: http://wiki.openstreetmap.org/wiki/API_v0.6

[5] MariaDB.org – Ensuring continuity and open collaboration. MariaDB.org [online]. [2017] [cit. 2017-05-03]. Dostupné z: https://mariadb.org/

[6] Apache Tomcat. *Apache Tomcat* [online]. [2017] [cit. 2017-05-06]. Dostupné z: http://tomcat.apache.org/

[7] Java Servlets – predstavenie technológie. *Interval.cz* [online]. [2003] [cit. 2017-05-06]. Dostupné z: https://www.interval.cz/clanky/java-servlets-predstavenie-technologie

[8] Understanding WAR. *Spring.io* [online]. [2017] [cit. 2017-05-06]. Dostupné z: https://spring.io/understanding/WAR

[9] Java Platform, Enterprise Edition (Java EE) | Oracle Technology Network | Oracle. *Oracle.com* [online]. [2017] [cit. 2017-05-06]. Dostupné z: http://www.oracle.com/technetwork/java/javaee/overview/index.html

[10] Java™ Servlet Specification Version 2.4. *Oracle.com* [online]. [2003] [cit. 2017-05-06]. Dostupné z: http://download.oracle.com/otn-pub/jcp/servlet-2.4-fr-spec-oth-JSpec/servlet-2_4-fr-spec.pdf?AuthParam=1493656003_3f9f725e89f652f87436fbd3fb0154dd

[11] Úvod do JDBC | Interval.cz. *Interval.cz* [online]. [2003] [cit. 2017-05-06]. Dostupné z: https://www.interval.cz/clanky/uvod-do-jdbc/

[12] GitHub - google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back. *GitHub – The world's leading software development platform* [online]. [2008] [cit. 2017-05-06]. Dostupné z: https://github.com/google/gson

[13] Map Features - OpenStreetMap Wik. *OpenStreetMap Wiki* [online]. [2012] [cit. 2017-05-06]. Dostupné z: http://wiki.openstreetmap.org/wiki/Map_Features

[14] Tessellation. *Math is Fun – Maths Resources* [online]. [2014] [cit. 2017-05-06]. Dostupné z: http://www.mathisfun.com/geometry/tessellation.htm

[15] KOPETSCHKE, Igor, Petr KRETSCHMER a Jan NOVÁK, Data Preprocessing for Autonomous Navigation Systems. International Conference on Military Technologies, Brno, 2017, ISBN 978-1-5386-1988-9.

[16] ArcGIS: The Mapping and Analytics Platform [online]. Redlands: Esri.com, 2017 [cit. 2017-05-06]. http://www.esri.com/arcgis/aboutarcgis