TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Návrh a implementace monitorování vybraných parametrů vozidla prostřednictvím BBCU embedded jednotky

Diplomová práce

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

Autor práce: **Bc. Iuliia Ilinykh**
Vedoucí práce: Ing. Igor Kopetschke

Liberec 2021

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání diplomové práce

Návrh a implementace monitorování vybraných parametrů vozidla prostřednictvím BBCU embedded jednotky

Jméno a příjmení: **Bc. Iuliia Ilinykh**
Osobní číslo: M17000172
Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie
Zadávající katedra: Ústav nových technologií a aplikované informatiky
Akademický rok: **2020/2021**

Zásady pro vypracování:

1. Seznamte se s možnostmi a požadavky na komunikaci BBCU jednotky s primárním dispečinkem
2. Navrhněte komunikační schéma mezi BBCU jednotkou, primární a sekundárním dispečinkem a mobilním klientem s ohledem na sledované parametry
3. Navrhněte potřebné změny v komunikaci mezi primárním dispečinkem a BBCU při potenciálním přechodu na MQTT protokol
4. Implementujte komunikaci mezi jednotlivými výše uvedenými uzly za použití stávající technologie primárního dispečinku
5. Výsledné řešení otestujte. V závěru své práce uveďte srovnání možností komunikace při přechodu na MQTT

Rozsah grafických prací:	dle potřeby dokumentace
Rozsah pracovní zprávy:	40-50 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] GREENGARD, Samuel. The internet of things. Cambridge, Massachusetts: MIT Press, [2015]. ISBN 978-0262527736.

[2] LACKO, Ľuboslav. Vývoj aplikací pro Android. Brno: Computer Press, 2015. ISBN 9788025143476.

[3] CHOLLET, François. Deep learning v jazyku Python: knihovny Keras, Tensorflow. Přeložil Rudolf PECINOVSKÝ. Praha: Grada Publishing, 2019. Knihovna programátora (Grada). ISBN 978-80-247-3100-1.

Vedoucí práce:	Ing. Igor Kopetschke Ústav nových technologií a aplikované informatiky
Datum zadání práce:	10. února 2021
Předpokládaný termín odevzdání:	17. května 2021

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

Ing. Josef Novák, Ph.D.
vedoucí ústavu

V Liberci dne 10. února 2021

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracovala samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědoma toho, 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 nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

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

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědoma následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

17. 5. 2021 Bc. Iuliia Ilinykh

Návrh a implementace monitorování vybraných parametrů vozidla prostřednictvím BBCU embedded jednotky

Abstrakt

Práce se zabývá návrhem a implementací softwaru pro vzdálenou komunikaci s automobilovou řídící jednotkou. Tato jednotka umožňuje získávat data ze snímačů, které jsou součástí automobilu.

Serverová část softwaru je realizovaná ve webovém frameworku Flask a v jazyce Python. Klientskou část představuje mobilní klient pro operační systém Android, který je napsán v jazyce Java. Pro uchovávání dat byl použit relační databázový systém SQLite. Navržený a následně implementovaný software umožňuje registraci uživatele a přidání jeho vozidla do seznamu pomocí identifikačního čísla (VIN). Díky tomu je možné získávat data o daném vozidle.

Klíčová slova: Python, Java, Android, BBCU, REST, API, MQTT

Design and implementation of monitoring of selected car parameters using an embedded BBCU unit

Abstract

This Master Thesis focuses on design and implementation of software for remote communication with the car conrtol unit. This unit allows to obtain data from sensors that are part of the car.

The server part of the software is implemented in the Flask web framework and in the Python language. The client part is a mobile client for the Android operating system and is written in Java language. The relational database system SQLite was used for data storage. The designed and subsequently implemented software allows the user to register and then add his vehicle to the list using an identification number (VIN) in order to obtain data about said vehicle.

Keywords: Python, Java, Android, BBCU, REST, API, MQTT

Poděkování

Ráda bych poděkovala vedoucímu práce panu Ing. Igorovi Kopetschkemu za vedení mé diplomové práce a cenné rady, které mi umožnily tuto práci dokončit.

Obsah

Seznam zkratek

API	Application Programming Interface
ART	Android Runtime
AWS	Amazon Web Services
BBCU	Board & Body Control Unit
BCM	Body Control Module
CAN	Controller Area Network
CRUD	Create, Read, Update, Delete
DTO	Data Transfer Object
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IBM	International Business Machines
IIoT	Industrial Internet of Things
ISO	International Organization for Standardization
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MCU	Microcontroller Unit
MQTT	Message Queue Telemetry Transport
OASIS	Organization for the Advancement of Structured Information Standards
OBU	On-Board Unit
OS	Operační systém
QoS	Quality of Service
REST	Representational State Transfer
UI	Uživatelské rozhraní
URL	Uniform Resource Locator
UTF-8	Unicode Transformation Format
UUID	Universally Unique Identifier
WADL	Web Application Definition Language
XML	Extensible Markup Language

1 Úvod

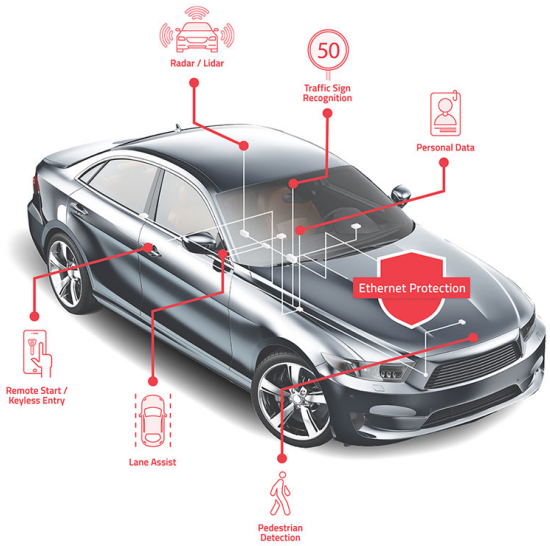
Diplomová práce se zabývá popisem návrhu softwaru pro komunikaci s automobilovou řídící jednotkou a jeho implementací. Mým cílem bylo vytvořit webovou REST službu v jazyce Python a s ní spojenou mobilní aplikaci pro operační systém Android. Mobilní klient umožňuje uživateli získávat informace o jeho autě a tímto způsobem kontrolovat jeho stav.

Následující kapitola popisuje konkrétní motivace včetně architektury samotného řešení pro vytvoření této práci. Kapitola číslo tři je zaměřena především představení technologií, které byly použity při psaní a programování aplikace a webového serveru. Čtvrtá a pátá kapitola je věnována celkovému návrhu práce a její následné implementace. Nejprve je detailně popsána komunikace mezi serverem a mobilní aplikací, další část popisuje použité knihovny a důvody, které vedly k jejich výběru. V této kapitole je také předveden návrh uživatelského rozhraní a popsán návrh databázového systému a jeho implementace. Kapitola také obsahuje informaci o MQTT protokolu a informaci o jeho komunikaci s BBCU jednotkou. Kapitola číslo šest popisuje testování aplikace a serveru a vyhodnocení tohoto testu.

V závěru práce jsou shrnuty dosažené cíle a navrženy možnosti rozšíření aplikace pro její následné použití nebo pro dosažení lepších výsledků, včetně rozšíření nabízených funkcí.

2 Motivace

Během posledních let se výrazně rozšířila míra využití elektroniky ve vozidlech. Vozidla obsahují různé kamery, asistenční systémy pro řidiče, senzory, displeje na palubní desce a další. V důsledku toho vznikla nutnost opatřit vozidla sítěmi, které jsou škálovatelné a schopné podporovat větší množství systémů a zařízení. Kromě toho musí sítě ve vozidle splňovat průmyslové standardy s ohledem na prostředí, ve kterém jsou používány (teplota, spotřeba energie, spolehlivost). Jednou z takových sítí je Automotive Ethernet. Je pravděpodobné, že automobilový Ethernet v budoucnu nahradí jiné automobilové sítě. V tradičním dopravním prostředí musí v případě potřeby



Obrázek 2.1: Automotive Ethernet [8].

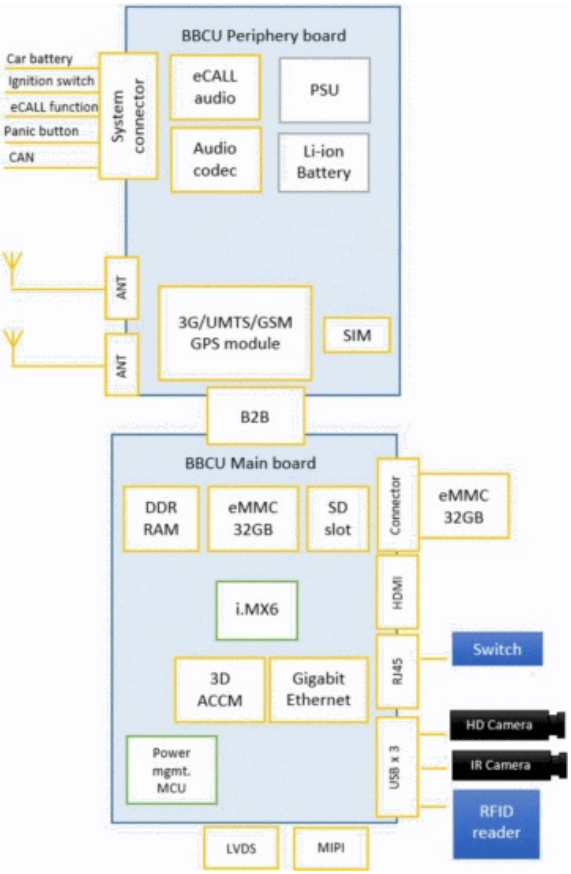
řidič rozhodovat o stavu svého vozidla sám na základě svých zkušeností. Naproti tomu majitel vozidla, jehož součástí je BBCU jednotka, může v kritickém okamžiku získat potřebné informace o svém autě za pomoci svého mobilního zařízení.

2.1 BBCU embedded jednotka

BBCU je hardwarový komponent určen pro automobily, ve kterých je komunikace mezi řídícími jednotkami a periferiemi realizovaná pomocí BroadR-Reach Automotive Ethernet. BBCU jednotka je vyvinuta tak, aby byla schopna komunikovat nejen

s interními zařízení, ale aby umožňovala připojení dalších externích komponentů. Tato jednotka je doplněna uživatelským rozhraním, a poskytuje základní funkce:

- schopnost sledovat údaje o cestě a dalších událostech a zaznamenávat událostí ze senzorů, které jsou umístěny uvnitř a vně vozidla;
- schopnost komunikovat s periferiemi (RFID čtečky, kamery atd.);
- schopnost analyzovat data získaná z periferních systémů.



Obrázek 2.2: HW architektura BBCU jednotky [15].

Jednotka se skládá ze dvou desek; z hlavní a periferní desky viz. obrázek 2.2 [15] Hlavní deska obsahuje paměť, akcelerometr, teplotní senzor, MCU, 3G/LTE modul. Periferní deska se používá pro napájení a komunikační spojení, jako například CAN a GPS. Tato deska disponuje také zvukovým kodekem a Li-ion bateriemi.

Data z BBCU jsou ukládány do databáze, která je umístěna na e-MMC kartě. Přístup k uloženým informacím je umožněn prostřednictvím webové aplikace, která běží přímo na BBCU, a to pomocí lokální nebo mobilní sítě.

Vývoj BBCU jednotky je stále v procesu.

2.2 Automotive Ethernet

Stejně jako sběrnice CAN, i automobilový Ethernet je paketizovaným systémem, ve kterém se informace přenášejí v paketech mezi uzly, viz. obrázek 2.1 [8]. Způsob použití je podobný jako u CAN, zároveň ale automobilový Ethernet nabízí mnohem větší šířku pásma a lze jím nejen nahradit sběrnice CAN, ale také ji rozšířit. Automotive Ethernet může být využit u libovolné sítě pro systémy ve vozidle, která funguje na bázi Ethernetu. Lze říct, že Automotive Ethernet je pojmem pro *BroadR-Reach (OPEN Alliance BroadR-Reach)* a *100Base-T1 (IEEE 802.3bw-2015)*. V obou případech je automobilový Ethernet přizpůsoben tak, aby umožňoval rychlou datovou komunikaci pro síťové propojení ve vozidle [3].

- BroadR-Reach Automotive Ethernet.

  Technologie BroadR-Reach [11] je základem fyzické vrstvy Ethernet, který byl navržen pro použití v automobilovém průmyslu. Tato technologie umožňuje použití několika systémů ve vozidle a přístup k informacím. Užití dané technologie dovoluje převést více aplikací do jedné škálovatelné sítě Ethernet. BroadR-Reach nabízí plně duplexní provoz na jediném páru vodičů s rychlost 100Mbit za sekundu, jak lze vidět na obrázku 2.3. Na koncích jsou čipy PHY, které odesílají a přijímají data v obou směrech zároveň. Rozhraní MAC má standardní IEEE 802.3 protokol. Jediný rozdíl v porovnáním s běžným Ethernetem je v části od jednoho PHY čipu do druhého [30].

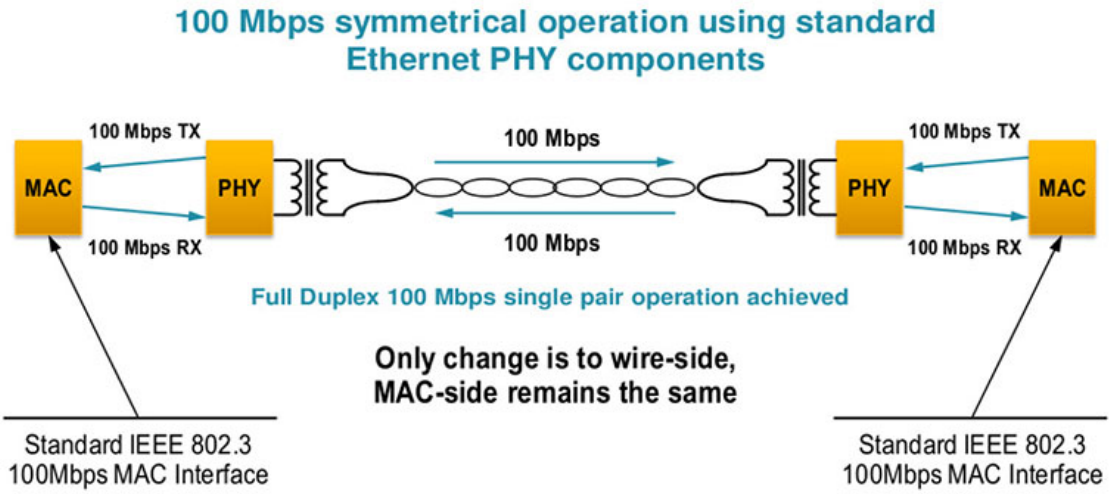
- 100Base-T1 (IEEE 802.3bw-2015).

  Technologie 100Base-T1 je interoperabilní technologie BroadR-Reach. Na rozdíl od BroadR-Reach definuje test (v testovací sadě fyzické vrstvy) pro maximální diferenciální výstup vysílacího vrcholu, který není přímo definován ve specifikaci BroadR-Reach. Další rozdíl je v časování protokolů specifikace 100Base-T1.

Automotive Ethernet nabízí větší šířku pásma, než většina standardů pro automobily. Kromě toho je výhodou použití technologie typu BroadR-Reach velká úspora nákladů na kabeláž. Technologie nabízí tenké, lehké a kroucené kabely. Další výhoda je provozování technologií, které nejsou proprietární, což výrazně snižuje náklady na výrobu.

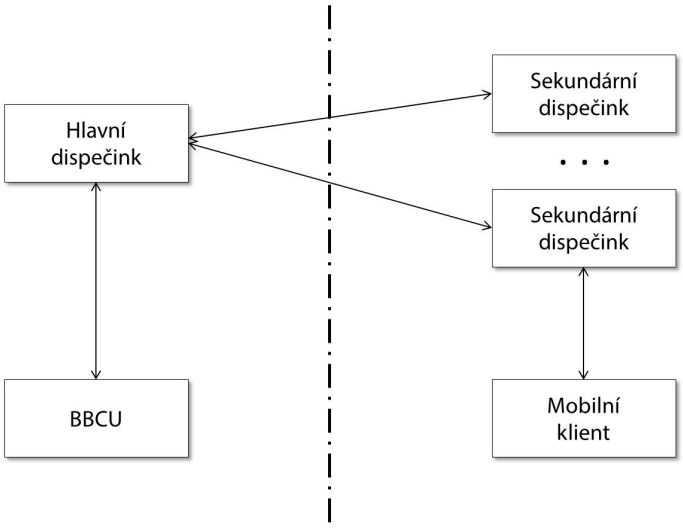
2.3 Popis řešení

BBCU embedded jednotka komunikuje s hlavním dispečinkem, do kterého jsou registrovány jeden nebo více tzv. sekundárních dispečinků, jejichž úkolem je komunikovat s mobilním klientem. Tato komunikace je znázorněna na obrázku 2.4. Do jednoho sekundárního dispečinku může být registrováno vice klientů. Komunikace mezi sekundárním dispečinkem a mobilní aplikací probíhá pomocí ”dotazování”: sekundární dispečink přijímá dotazy od mobilní aplikace a následně přijímá odpověď



Obrázek 2.3: BroadR-Reach Automotive Ethernet [3].

od hlavního dispečinku. Kromě zpracování zpráv z automobilu a jejich následného předávání mobilním klientům má sekundární dispečink za úkol uchovávání přijatých dat a zpracovávání pohotovostních hlášení v případě havárie nebo poruchy. Mobilní klient umožňuje registraci uživatele, zobrazení dat o konkrétním automobilu a zobrazování upozornění ze sekundárního dispečinku. Pro propojení serveru s mobilní



Obrázek 2.4: Schéma komunikaci.

aplikací se používá model typu client-server. Serverová část představuje webovou REST službu napsanou v jazyce Python. Klientská (mobilní) aplikace pro operační systém Android je napsaná v jazyce Java. Pro komunikaci se serverem používá mobilní aplikace knihovnu Retrofit, která bude podrobně popsána v kapitole 4.3.1. Webová služba bude v budoucnu rozšířena o knihovnu FLASK-RESTPlus (viz. kapitola 4.4.1), která podporuje rychlé vytváření REST API.

3 Použité technologie

Tato kapitola popisuje zvolené technologie. Jedná se o operační systém Android a REST architekturu.

3.1 OS Android

Pro vývoji mobilní aplikace byl zvolen OS Android. Android je mobilní operační systém na bázi jádra Linux, který lze používat nejen pro mobilní telefony a tablety, ale také pro nositelná zařízení (Android Wear), pro automobily (Android Auto) a pro chytré televizory (Android TV).

První neveřejná a nekomerční verze operačního systému byla vydána v roce 2007. Následovala první veřejná verze Android Beta. Od prvního vydání do současnosti vzniklo 18 verzí operačního systému (viz. tabulka 3.1), přičemž zveřejnění Beta verze Android 11 bylo plánováno na červenec roku 2020 [5].

Aby mobilní klient dokázal komunikovat s jednotkou BBCU, musí disponovat verzí 7.0 (Nougat) s API 24 nebo vyšší. Aplikace je plně funkční na více než 73% zařízeních, co je vidět na obrázku 3.1.

3.1.1 Architektura

Architektura operačního systému je rozdělena do 5 základních vrstev a jednu mezivrstvu [13]. Přehled vrstev je znázorněn na obrázku 3.2:

- Linux Kernel.

  Jedná se o nejnižší vrstvu oddělující hardware od softwaru, který je umístěn ve vyšších vrstvách. Na rozdíl od originálního Linuxu nepodporuje modifikovaná Android verze kompletní sadu GNU knihoven a X Windows. Umožňuje nicméně podporu správy sítě a paměti a správu procesů, jako je například souběžný běh aplikací.

- Hardware Abstraction Layer (HAL).

  HAL je mezivrstva, která umožňuje komunikaci hardwaru s vyššími úrovněmi rozhraní. Jsou v ní umístěny knihovny, které aplikují rozhraní na konkrétní typ hardwarového komponentu.

- Platform libraries.

Verze	Název	Verze API	Rok výroby
1.0	Apple Pie	1	2008
1.1	Banana Bread	2	2009
1.5	Cupcake	3	2009
1.6	Donut	4	2009
2.0-2.1	Eclair	7	2009
2.2	Froyo	8	2010
2.3.3-2.3.7	Gingerbread	10	2010
3.0-3.2	Honeycomb	13	2011
4.0.3-4.0.4	Ice Cream Sandwich	15	2011
4.1-4.3.1	Jelly Bean	16-18	2012
4.4	KitKat	19-20	2013
5.0-5.1	Lollypop	21-22	2014
6.0	Marshmallow	23	2015
7.0-7.1	Nougat	24-25	2016
8.0-8.1	Oreo	26-27	2017
9.0	Pie	28	2018
10	Android 10	29	2019
11	Android 11	30	2020

Tabulka 3.1: Verze OS Android.

Tyto knihovny jsou napsány v C nebo C++ a fungují na bázi Android Application Framework. Patří mezi ně například *media libraries* (slouží pro přehrávání video a audio formátů a pro zobrazení obrazových souborů), *libc* (standardní knihovna jazyka C pro embedded zařízení), *SQLite* (relační databázová knihovna), *OpenSSL* a další.

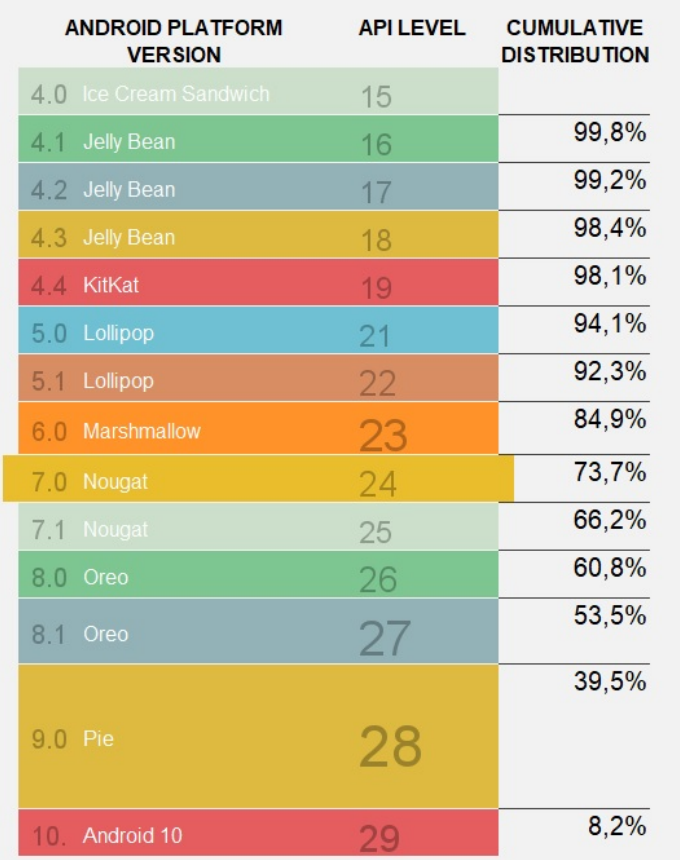
- Android Runtime (ART).

  ART je virtuální stroj pro běh aplikace a pro určité služby v systému. V roce 2014 nahradil původní virtuální stroj Dalvik. Android Runtime provádí formát Dalvik Executable a má specifikace Dex Bytecode. Představuje takzvanou kompilaci v předstihu, která zvyšuje výkon aplikace. Díky těmto specifikům je výrazně rychlejší než jeho předchůdce, Dalvik.

- Application framework.

  Umožňuje vytvořit vlastní aplikace, které disponují stejnými prvky jako aplikace systémové. Základní sada služeb obsahuje:

  - *View* — sada prvků pro sestavení UI (tlačítka, textové pole a tak dále);
  - *Content providers* — slouží pro přístup k obsahu jiných aplikací;
  - *Resource manager* — poskytuje přístup do přidaných souboru, do grafiky a dalších zdrojů, které nejsou programovatelné;
  - *Notification manager* — umožňuje zobrazení upozornění ve stavovém řádku;



Obrázek 3.1: Android OS Cumulative Distribution [14].

  - *Activity manager* — spravuje životního cyklus aktivit.

- Android aplikace.

  Nejvyšší vrstva, která se zobrazuje běžnému uživateli. Obsahuje stažené a předinstalované aplikace.

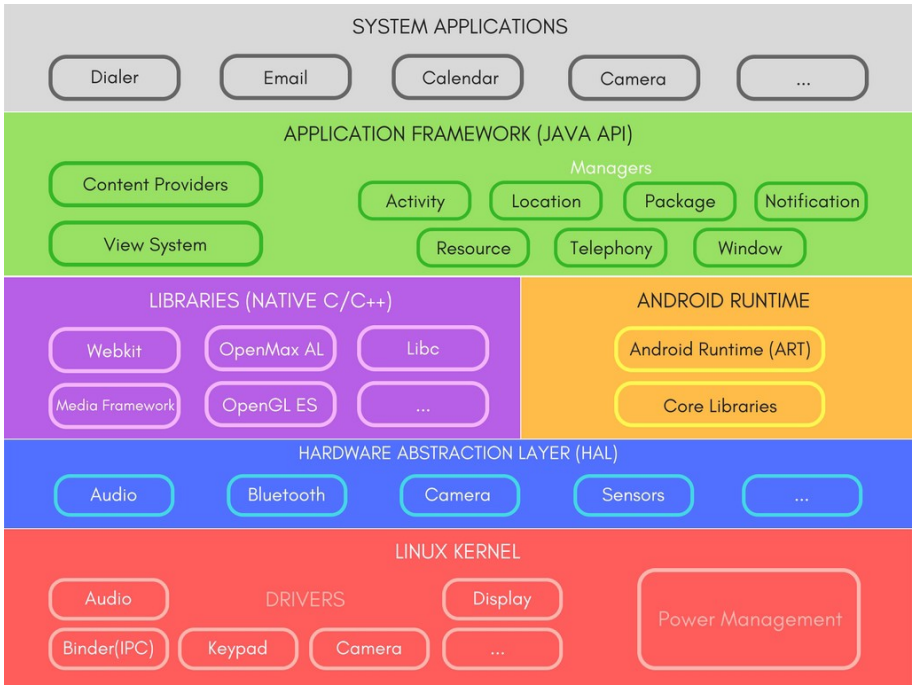
3.1.2 Základní součásti aplikace

Android aplikace se skládá z několika součástí:

- *Activity.*

  Základní komponent pro zobrazení aplikace. Jedna aktivita reprezentuje právě jednu obrazovku UI a slouží pro interakci s uživatelem. Každá aplikace obsahuje několik aktivit, mezi nimiž může uživatel přepínat. Činnosti v Android aplikacích jsou spravovány jako zásobník. Každá aktivita má svůj životní cyklus a vždy se nachází v jedné z jeho fází (znázorněno na obrázku 3.4) [19]:

  - Aktivita spuštěna — právě došlo ke spuštění aplikace;
  - Aktivita běží — aktivita je spuštěna a běží v popředí;

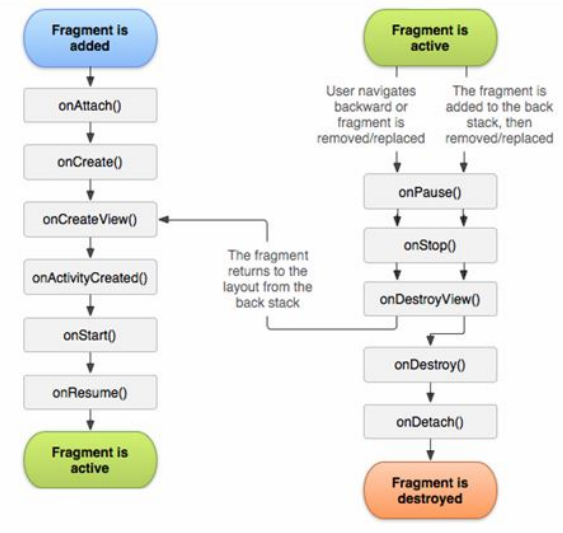


Obrázek 3.2: Architektura OS Android [13].

- Aktivita v pozadí — aktivita je spuštěna ale je překryta jinou aplikací (například příchozí SMS nebo jinou notifikací);
- Aktivita zastavéna — aktivita není vidět, uživatel k ní nemá přístup, není ale zcela ukončena;
- Aktivita ukončena — úplné ukončení aktivity a její odstranění z paměti.

V životním cyklu aktivity existují následující metody:

- onCreate() — vyvolaná při prvním spuštění aktivity, která doposud nebyla spuštěna anebo byla odstraněna z paměti. Tato metoda musí být implementována nebo překryta za každých okolností;
- onStart() — následuje metodu *onCreate()* a stejně jako ona je vyvolaná v případě, že je aktivita znovu aktivována po svém skrytí;
- onResume() — metoda, která je vyvolána těsně předtím, než se aktivita posune do popředí. V tomto okamžiku do aktivity vstupuje uživatel;
- onPause() — vyvolána před přechodem aktivity do pozadí. Systém získává pravomoc aktivitu násilně ukončit;
- onStop() — vyvolána při zastavení aktivity;
- onRestart() — volána v návaznosti na metodu *onStop()* při restartu aplikace;
- onDestroy() — vyvolána předtím, než je aktivita odstraněna z paměti.



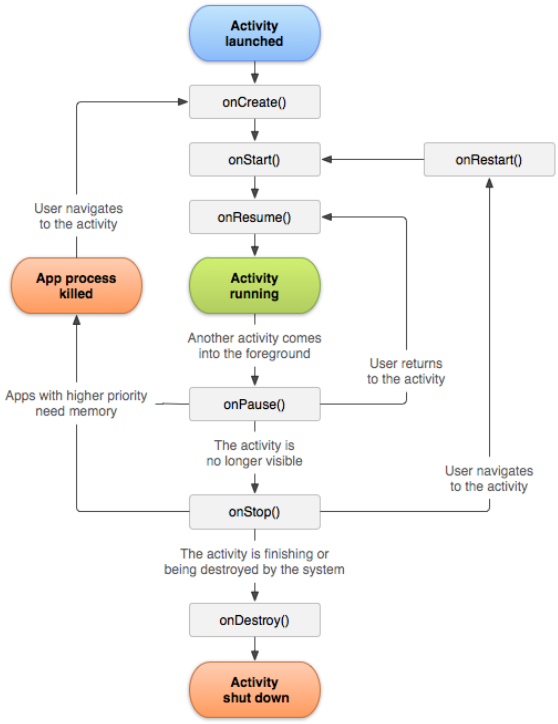
Obrázek 3.3: Životní cyklus fragmentu [4].

- *Fragments.*

  Fragment je součástí chování aktivity, která bez něj nemůže být správně zobrazena. Lze proto říct, že fragment je druh subaktivity. Jedna aktivita může mít několik fragmentů a jednotlivé fragmenty mohou být použity v různých aktivitách. Stejně jako samotná aktivita, i fragment má svůj životní cyklus, který je zobrazen na obrázku 3.3 [4]. Tento cyklus je v porovnání s životním cyklem aktivity komplikovanější.

  Metody, které existují v životním cyklu fragmentu, fungují obdobně jako metody aktivity, je jich ale o 4 více:

  - onAttach() — vyvolána při připojení tzv. instance fragmentu k aktivitě při instalaci aktivity. V tomto okamžiku ještě fragment a aktivita nejsou plně inicializovány;
  - onCreate() — metoda vyvolaná při prvním vytvoření fragmentu po vyvolání odpovídající metody aktivity;
  - onCreateView() — po svém vyvolání vytváří vizuální vzhled fragmentu;
  - onActivityCreated() — metoda vyvolaná po vytvoření aktivity. Od této chvíle jsou komponenty uživatelského rozhraní přístupné pomocí metody *findViewById()*;
  - onStart() — vyvolána v případě, že je fragment viditelný;
  - onResume() — vyvolána před obnovením fragmentu;
  - onPause() — slouží pro zastavení fragmentu, po vyvolání této metody je fragment umístěn do pozadí a překrytý jiným prvkem;
  - onStop() — metoda vyvolaná při zastavení fragmentu;
  - onDestroyView() — vyvolání této metody skryje zobrazení fragmentu;



Obrázek 3.4: Životní cyklus aktivit [19].

- onDestroy() — metoda vyvolaná před zničením fragmentu;
- onDetach() — metoda vyvolaná při zničení fragmentu.

Fragmenty se často používají pro konkrétní účely, proto lze vymezit třídy, ke kterém jsou již přiřazeny určité funkce: *ListFragment* (spravuje seznam položek), *DialogFragment* (používá se pro vytváření dialogových oken) a *PreferenceFragment* (slouží pro ovládání nastavení aplikace).

- *Service.*

  Tento komponent neposkytuje uživatelské rozhraní nýbrž představuje proces, který běží na pozadí. Používá se pro dlouhotrvající operace. Služby lze spustit dvěma způsoby; metodou *startService* nebo metodou *bindService.*

- *Content Provider.*

  Jedná se o rozhraní pro sdílení dat mezi aplikacemi nebo mezi jednotlivými aktivitami. Má standardní metody: *insert, update, delete a query.* Tyto metody mají stejné funkce jako metody databázové.

- *Broadcast Receiver.*

  Umožňuje aplikaci reagovat na příchozí zprávy (nízký stav baterie, stažení dat a podobné). Stejné jako *Content Provider* nedisponuje uživatelským rozhraním.

Všechny komponenty musí být definovány v souboru *AndroidManifest.xml* [7], který se nachází v kořenovém adresáři projektu. Tento soubor obsahuje informace o aplikaci. V manifestu musí být také vyznačeny oprávnění, která aplikace potřebuje pro komunikaci s jinými aplikacemi nebo pro přístup do chráněných součástí systému. Dále zde musí být uvedena oprávnění, která potřebují ostatní aplikace v případě, že potřebují přístup do definované aplikace.

3.2 REST

REST je architektura pro vytváření distribuovaných webových služeb, kterou navrhl v roce 2000 Roy Fielding [25]. Aby byl mohl být distribuovaný systém označen jako navržený podle REST architektury, musí splňovat následující kritéria:

- Systém by měl být rozdělen na klienty a servery. Základem takového rozdělení je diferenciace potřeb: oddělení potřeb a funkcí klientského rozhraní od potřeb serveru.
- Server by neměl ukládat žádné informace o stavu klienta (stav je uložen na zařízení klienta). Kromě toho může informace o stavu server přenášet do jiné služby (například do databáze), aby byl zachován aktuální stav.
- U každé odezvy musí být zřejmé, zda je uchovatelná nebo ne (cache), tzn. zdali je nebo není uložena do mezipaměti. Účelem je zabránit tomu, aby klienti přijímali nesprávná nebo zastaralá data.
- Musí existovat univerzální rozhraní mezi komponenty systému, které každé ze služeb umožňuje rozvíjet se nezávisle [2]. Pro získání takového rozhraní jsou stanoveny následující omezení [17]:
  - *Identification of resources.*
    V REST službě je zdrojem vše, co lze pojmenovat: uživatel, obrázek, soubor atd. Každý zdroj musí být označen stabilním identifikátorem, který se nezměnění při změně stavu zdroje. Zdroje jsou koncepčně odděleny od pohledů a server může odesílat informace z databáze ve formátu HTML, JSON nebo XML.
  - *Manipulation of resources through representations.*
    Zobrazení představuje aktuální nebo požadovaný stav zdroje. Pokud klient ukládá zdroje včetně metadat, má dostatek informací k jejich úpravě nebo odstranění.
  - *Self-descriptive messages.*
    Žádost a odpověď musí sami o sobě ukládat všechna potřebná informace pro jejich zpracování.
  - *HATEOAS (Hypermedia As The Engine Of Application State)* [27].
    K navigaci by měl být použit hypertext. Klienti mění stav systému prostřednictvím akcí, které jsou definovány hypermedii na serveru. Velmi

populárními formáty pro poskytování odkazů jsou RFC 5988 a JSON Hypermedia API Language.

- Systém musí být rozdělen do hierarchie vrstev způsobem, který každému komponentu umožňuje „vidět“ pouze komponenty vedlejší vrstvy. Klient obvykle není schopen rozlišit, zda komunikuje přímo se serverem nebo s uzlem.
- Klient by měl mít možnost načíst a spustit kódy; to znamená že REST umožní rozšířit funkčnost klienta pomocí staženého kódu ze serveru formou appletů nebo skriptů.

V případě, že systém splňuje uvedená kritéria, lze říct, že je založen na architektuře REST a může být nazýván RESTful servisem. V RESTful službě se využívá HTTP metoda [26]. Před použitím metody musí být stanoveny dvě charakteristiky, viz obrázek 3.5:

- Zabezpečení — metoda HTTP je považována za bezpečnou, když její volání nemění stav dat. Bezpečnou metodou je například GET, která neaktualizuje data na straně serveru.
- Idempotence — metoda je idempotentní, když její volání vrací stejnou odpověď právě tolikrát, kolikrát byla metoda vyvolána.

HTTP Method	Safe	Idempotent
GET	✓	✓
POST	✗	✗
PUT	✗	✓
DELETE	✗	✓
OPTIONS	✓	✓
HEAD	✓	✓

Obrázek 3.5: HTTP metody [26].

Jak už bylo řečeno, metoda *GET* je bezpečná a idempotentní. Obvykle se používá k extrahování informací. Metoda *POST* není považována ani za bezpečnou ani za idempotentní. Nejčastěji se používá k vytváření zdrojů. Metoda *PUT* není bezpečná, je ale idempotentní. Proto je vhodnější v případě, že je třeba informace aktualizovat, používat tuto metodu namísto metody POST. Metoda *DELETE* se používá k odstranění informace. Metoda *OPTIONS* se nepoužívá pro žádnou manipulaci se zdroji. Pokud však klient nezná jiné metody použitelné pro daný požadavek, může pomocí této metody získat více potřebných informací. *HEAD* se používá k získání zdrojů ze strany serveru. Je velmi podobná metodě GET, ale na rozdíl od ní odesílá požadavek a přijímá odpověď pouze v záhlaví. Podle specifikace HTTP metoda

nepoužívá tělo pro požadavek. HTTP definuje různé kódy odezvy na požadavek, které jsou podrobně znázorněny na obrázku 3.7. Dané kódy poskytují uživateli informace o operacích. Při navrhování REST architektury aplikace je nutné přemýšlet o aplikaci z hlediska zdrojů.

Obrázek 3.6: Typický REST response [26].

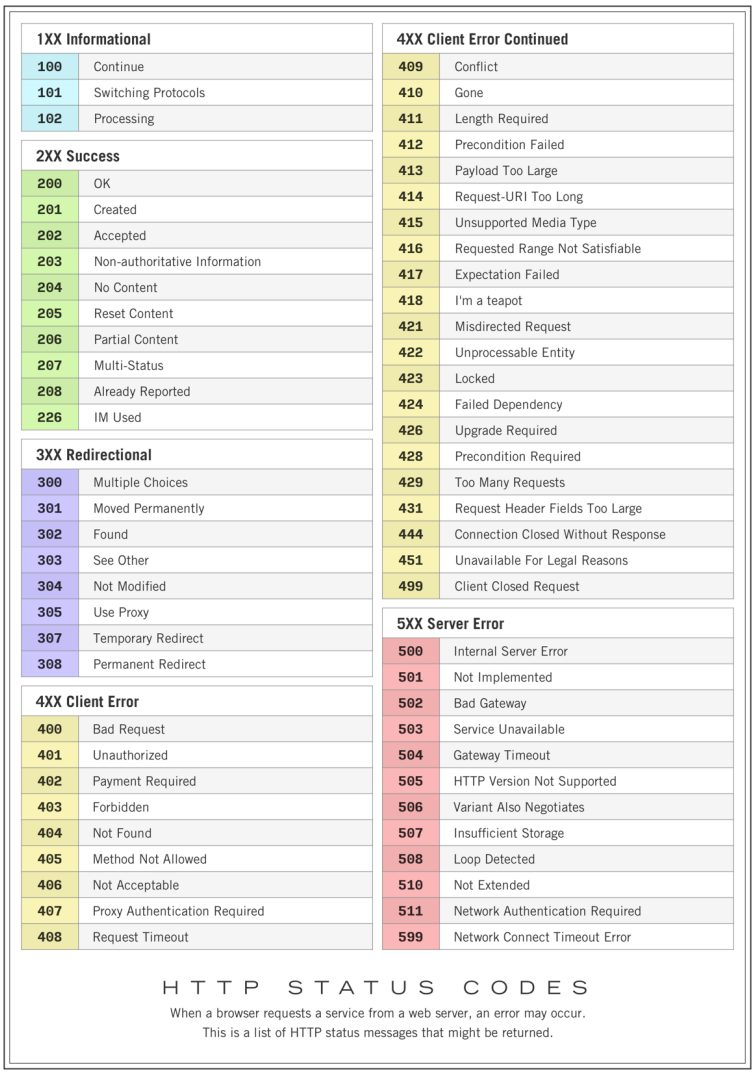
- Neexistují žádná omezení pro výměnu dat: lze použít například JSON formát, který je velmi populární, ale dostupné je i velké množství jiných variant, například XML.
- REST je architekturou, která je zcela založena na HTTP.
- REST je flexibilní. Neexistuje žádný standard pro definice služeb. Nejvyužívanějšími jazyky pro definování webových aplikací jsou WADL a Swagger.

Na obrázku 3.6 je znázorněna typická odpověď v RESTful službě. Požadavek *«GET http://localhost:8080/users»* tomto příkladu vrací data o třech různých uživatelích. Požadavek typu *«GET http://localhost:8080/users/1»* vrací informace pouze o prvním uživateli, který má id=1.

Výhody REST:

- Přenos dat v jejich nezměněné podobě;

- Každý zdroj je jednoznačně určen konkrétní URL adresou. Taková adresa je primárním klíčem pro jednotku dat, a nezáleží na tom, v jakém formátu jsou data uložena;
- Protokol přenosu dat: pro HTTP lze použít metody GET, PUT, DELETE a POST. Pro CRUD lze používat jak všechny 4 metody zároveň, tak pouze metodu GET a POST.



1XX Informational	
100	Continue
101	Switching Protocols
102	Processing

2XX Success	
200	OK
201	Created
202	Accepted
203	Non-authoritative Information
204	No Content
205	Reset Content
206	Partial Content
207	Multi-Status
208	Already Reported
226	IM Used

3XX Redirectional	
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
308	Permanent Redirect

4XX Client Error	
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout

4XX Client Error Continued	
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	Request-URI Too Long
415	Unsupported Media Type
416	Requested Range Not Satisfiable
417	Expectation Failed
418	I'm a teapot
421	Misdirected Request
422	Unprocessable Entity
423	Locked
424	Failed Dependency
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
444	Connection Closed Without Response
451	Unavailable For Legal Reasons
499	Client Closed Request

5XX Server Error	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required
599	Network Connect Timeout Error

Obrázek 3.7: HTTP Status Code [18].

4 Návrh

Tato kapitola popisuje návrh celého systému (klientské a serverové části) za použití technologií, které byly popsány v kapitole 3. Kapitola je rozdělena do 6 částí: první část je zaměřena na analýzu požadavků na software, druhá část popisuje procesu komunikace jednotky se softwarem. Následující části jsou věnovány návrhu databázového systému, uživatelského rozhraní a popisu samotného navrženého softwaru.

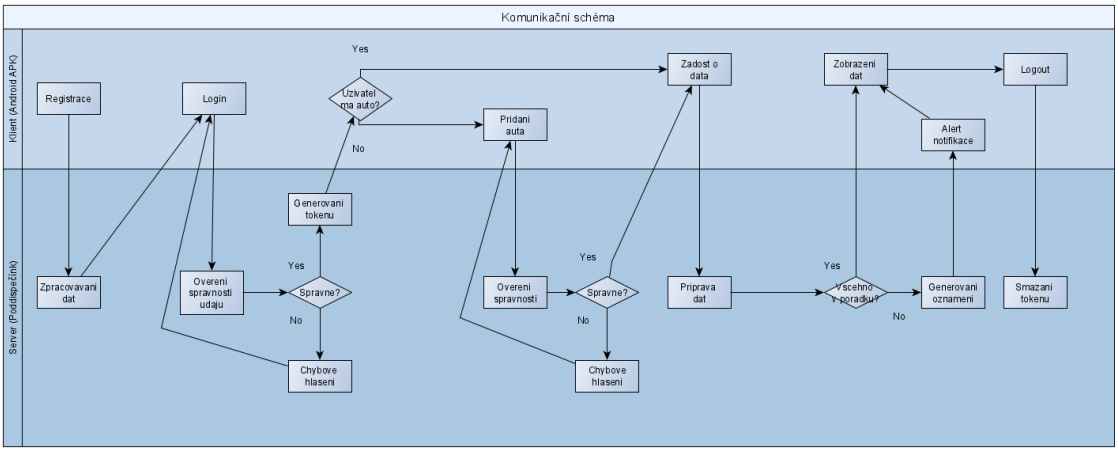
4.1 Analýza požadavků

4.1.1 Funkční požadavky

- Systém umožní registraci nových uživatelů;
- Systém umožní uživateli přístup k data, které patří do jeho vozidla;
- Mobilní klient bude zobrazovat notifikace o stavu vozidla;
- Po zadání kontrolních údajů, uživatel bude mít přístup k data svého vozidla;
- Mobilní aplikace bude ukládat data nového uživatele na příslušný server;
- Webová služba umožní kontrolu uživatelem uvedených dat.

4.1.2 Nefunkční požadavky

- Mobilní klient je kompatibilní s OS Android ve verzi 7.0 nebo vyšší;
- Jako úložiště systém využívá relační databázový systém SQLite a SharedPreferences;
- Komunikace mezi mobilní aplikací a webovou službou probíhá pomocí REST architektury;
- Mobilní aplikace je napsaná v jazyce Java;
- Webová služba je napsaná v jazyce Python.



Obrázek 4.1: Komunikační schéma.

4.2 Komunikace aplikace se serverem

Jak už bylo zmíněno na začátku, komunikace mezi serverem a klientem probíhá pomocí zasílání potřebných zpráv a následným zpracováváním odpovědi. Na obrázku 4.1 je znázorněno, jakým způsobem tato komunikace probíhá. Webová služba zprostředkovává přístup do dat z automobilu a implementuje REST architekturu. Pro předávání dat byl zvolen formát JSON.

Je nezbytné, aby byl uživatel registrován. Po zpracování přijatých dat serverem je klientovi umožněno přihlásit se do mobilní aplikace. Pokud uživatel při přihlášení zadá chybná data, obdrží ze serveru chybové hlášení a celý proces autentifikace je nutné opakovat. Po zadání správných údajů dojde o odeslání autorizačního tokenu.

Pokud jde o první přihlášení, uživatel je vyzván k přidání informací o jeho vozidle. Tyto data jsou následně serverem ověřeny.. Pokud jsou data vyhodnocena jako chybná, klient opět obdrží chybovou hlášku a proces přidání vozidla musí opakovat. Po zadání správných dat obdrží mobilní aplikace ze serveru odpověď s potřebnými informacemi. V případě, že status vozidla není v pořádku (například je poškozené), dostane uživatel na svůj telefon notifikaci.

Po odhlášeni uživatele server odstraní autorizační token.

4.3 Mobilní aplikace

Jak už bylo řečeno, pro mobilní aplikaci byl zvolen jazyk Java a vývojovým prostředím je Android Studio.

4.3.1 Použité knihovny

Pro komunikaci s webovou službou byla použita jednoduchá a lehká knihovna Retrofit [28]. Tato knihovna usnadňuje interakci s REST API, a umožňuje odesílání

požadavků na webové služby pomocí příkazů GET, POST, DELETE a PUT. Umožňuje práci v asynchronním režimu, což eliminuje zbytečný kód.

Kód 4.1: Implementace knihoven.

```
implementation 'com.squareup.retrofit2:retrofit:2.7.1'
implementation 'com.squareup.retrofit2:converter-gson:2.7.1'
implementation 'com.squareup.retrofit2:adapter-rxjava2:2.4.0'
```

Jedním z důvodů pro použití právě knihovny Retrofit je její schopnost pracovat s JSON, a to pomocí automatického analyzování a parsování tohoto formátu. Retrofit disponuje také na rozdíl od Volley [6] dobrou dokumentací, což zlehčuje jeho použití a pochopení principu postupů. Implementace Retrofit je jednoduchá: do souboru *build.gradle* je nutné zaznamenat závislosti, které jsou znázorněny na ukázce 4.1.

Lze říct, že Retrofit představuje most mezi napsaným kódem a REST službou. To umožňuje rychle implementovat rozhraní do aplikace HTTP.

Pro práce s Retrofit knihovnou je potřeba využít tři třídy:

- *Model* — třída, která představuje JSON model;
- *Interface* — rozhraní, které identifikuje možné operace;
- Třída *Retrofit.Builder* — instance, která používá API Builder k definování koncového URL pro HTTP operace.

Každá metoda rozhraní představuje možné volání API, viz ukázka kódu 4.2. Metoda musí obsahovat HTTP anotaci (GET, POST a jine), která upřesňuje typ požadavku. Také musí mít metoda relativní URL. Návratová hodnota dokončí odpověď v objektu Call s očekávaným výsledkem.

Kód 4.2: Metoda pro volání API.

```
@GET("users")
Call<List<User>> getUsers();
```

V metodě může být použita například anotace @Path a @Body, dotazy @Query a jiné náhradní parametry pro nastavení URL adresy, viz ukázka 4.3.

Kód 4.3: Používání anotaci @Query.

```
@GET("users")
Call<User> getUserById(@Query("id") Integer id);
```

Retrofit lze nakonfigurovat tak, aby byl použit konkrétní převodník dat. Kvůli tomu, že formát předávaných dat je JSON, byl zvolen převodník GSON. Jeho implementace je znázorněna v ukázce 4.1. Retrofit také podporuje možnost implementace vlastního převodníku.

Tuto knihovnu lze rozšířit o adaptéry, které budou sloužit pro interakci s jinými knihovnami, jako jsou například RxJava 2.x, Java 8 a Guava. Pro mobilní aplikace byl použit jenom jeden adaptér, konkrétně pro interakci s RxJava.

```
retrofit = new Retrofit.Builder()
        .baseUrl("URL")
        .addConverterFactory(GsonConverterFactory.create())
        .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
        .client(client)
        .build();
```

Obrázek 4.2: Konfigurace Retrofit pro mobilní aplikace.

Po implementaci všech rozšíření potřebných pro Retrofit bude jeho konečná konfigurace pro mobilní aplikace vypadat tak, jako je znázorněno na obrázku 4.2.

4.4 Serverová část

Pro serverovou část byl zvolen programovací jazyk Python, framework Flask a vývojové prostředí PyCharm.

4.4.1 Použité knihovny

Kromě předem nainstalovaných knihoven ve Flasku byly použité i některé další. Defaultně je pro práci s REST API možné použít několik knihoven, například FLASK-RESTPlus a FLASK-RESTful. Pro serverovou část byla zvolena knihovna FLASK-RESTPlus. Dvě výše uvedené knihovny fungují velmi podobným způsobem. V podstatě jediný rozdíl spočívá v tom, že RESTPlus má předem nainstalovaný Swagger. Danou knihovnu lze implementovat pomocí příkazu znázorněného v 4.4.

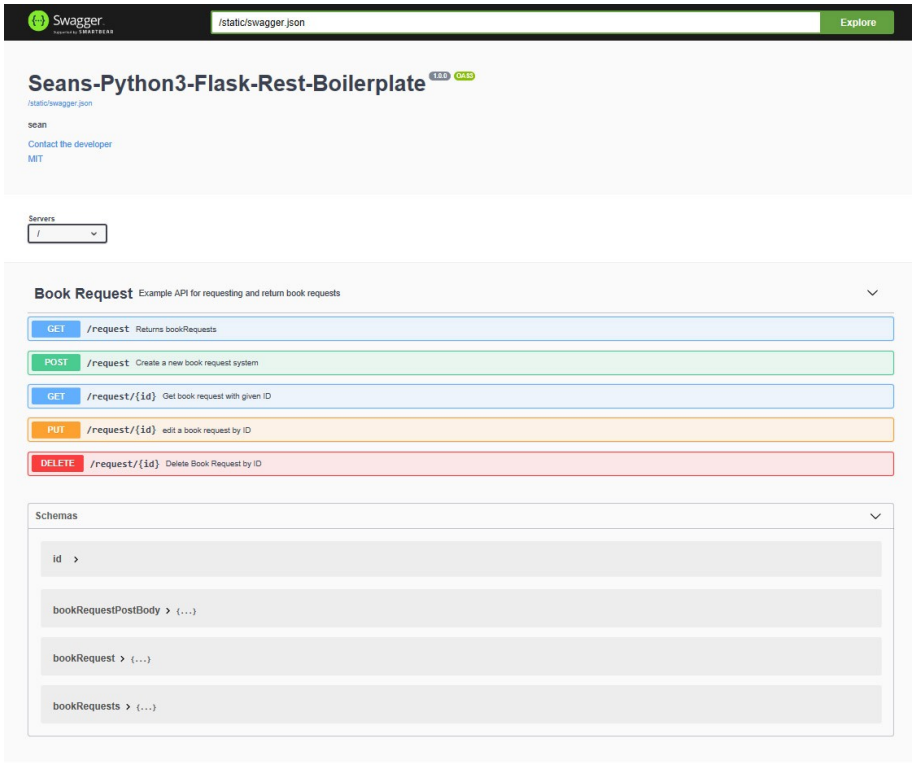
Kód 4.4: Implementace knihovny FLASK-RESTPlus.

```
1 pip install flask-restplus
```

Swagger je otevřeným frameworkem pro tvorbu, návrh a dokumentaci REST API. Kromě základních částí pro tvorbu rozhraní disponuje Swagger [10] také nástrojem pro vizualizaci a vyzkoušení API, viz obrázek 4.3.

Swagger má několik nástrojů pro tvorbu, návrh a interakci s REST API:

- *Editor* — nástroj, který umožňuje manuální tvorbu rozhraní. Kromě toho také označuje chyby, poskytuje tipy pro formátování a kontroluje, zda dokumentace splňuje pravidla specifikace OpenAPI.
- *Inspector* — slouží pro generování dokumentace a umožňuje testování libovolné existující API.



Obrázek 4.3: Swagger UI [10].

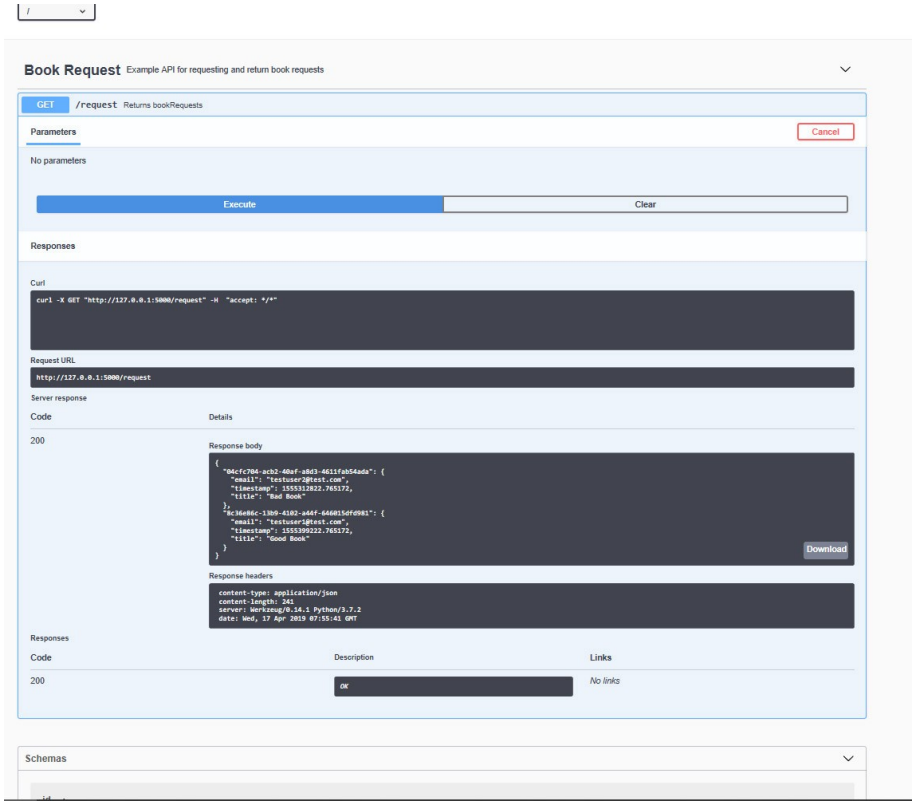
- *Codegen* — generuje zdrojový kód pro server a pro klienta. Daný kód pomáhá integrovat API na konkrétní požadovanou platformu a poskytuje robustnější implementace.
- *Swagger UI* — slouží pro testování vlastní API a umožňuje tvořit a popisovat rozhraní. Příklad lze vidět na obrázku 4.4. Swagger UI odesílá požadavek a následně zobrazuje odpověď.

Pro ukládání dat byla použita knihovna Flask-SQLAlchemy, která bude podrobněji popsaná v části 4.6.1.

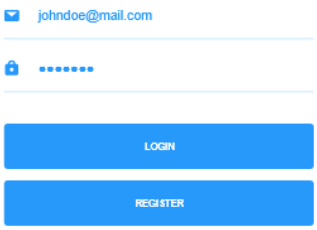
4.5 Návrh uživatelského rozhraní

Uživatelské rozhraní musí být komfortní a jednoduché pro použití. Následný návrh byl vytvořen v prostředí Adobe XD a obsahuje celkem čtyř obrazovky: přihlášení, registrace, profil a obrazovku pro data z automobilu. Každá z nich ukazuje, jaké akce mohou být v rámci této obrazovky prováděny.

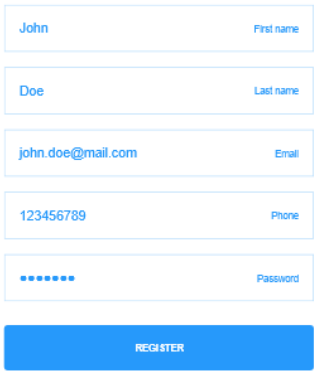
Na obrázcích 4.5 a 4.6. jsou znázorněny ukázkové obrazovky pro registraci uživatele a pro jeho následné přihlášení. Když se uživatel úspěšně přihlásí, bude přesměrován na jednoduchou obrazovku (viz. 4.7), která mu zobrazí informace o něm a o jeho vozidle. Obrazovka, která je zaměřena na registraci nového uživatele ho po odeslání požadavku na server nasměruje také na přihlašovací obrazovku.



Obrázek 4.4: Swagger UI GET request [10].

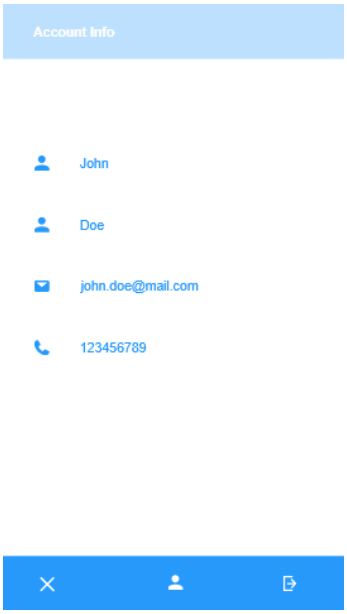


Obrázek 4.5: Obrazovka pro přihlášení.

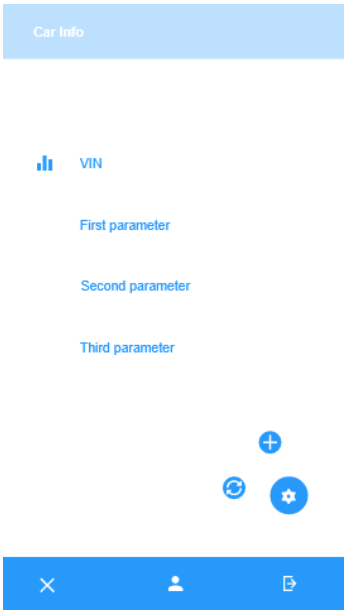


Obrázek 4.6: Obrazovka pro registraci.

Na obrázku 4.7 lze vidět, že pro navigaci bylo použito *bottom navigation view*,



Obrázek 4.7: Obrazovka pro profil uživatele.



Obrázek 4.8: Obrazovka pro informace z vozidla.

aby bylo přepínání mezi aktivitami snadno dostupné. Položky v menu jsou rozděleny do třech ”přepínačů”: jeden pro data o vozidle, druhý pro data samotného uživatele a třetí pro odhlášení.

Obrazovka 4.8, která slouží pro zobrazení dat o automobilu, musí podle návrhu obsahovat *floating action button*, rozdělenou na dvě tlačítka. První slouží pro přidání vozidla, druhá pro aktualizaci dat. Po kliknutí na první tlačítko se zobrazí dialogové okno, které uživatele vyzve k uvedení potřebných dat. Bez splnění tohoto požadavku nedojde k úspěšnému přidání vozidla. Po kliknutí na druhé tlačítko dojde k obnovení dat.

Na základě tohoto návrhu byla zpracována mobilní aplikace s požadovanými funkcemi.

4.6 Ukládání dat

Pro ukládání serverových a klientských dat byla použita relační databáze SQLite. Důvodem k jejímu použití bylo to, že Flask i Android v sobě mají integrovaný potřebný systém.

4.6.1 Serverová databáze

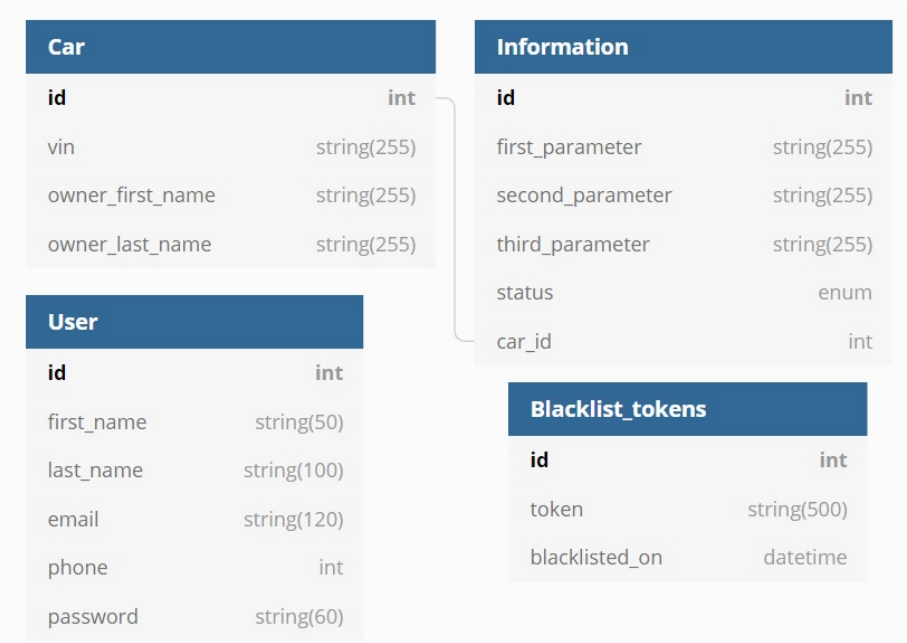
Kvůli tomu, že Flask zpočátku nepodporoval databáze, byl zvolen alternativní databázový systém, který lépe splňovat požadavky. Relační databáze jsou vhodné pro ukládání vztahů mezi datovými prvky. Proto byl pro všechny manipulace s daty

zvolen databázový systém SQLite, který lze importovat pomocí knihovny Flask-SQLAlchemy 4.5, která umožňuje aplikaci přístup do balíčku SQLAlchemy.

Kód 4.5: Implementace knihovny Flask-SQLAlchemy.

```
1 pip install flask-sqlalchemy
```

Data, která budou uložena v databázi, jsou reprezentovány souborem tříd, které se nazývají databázové modely. Celá databáze se skládá ze 4 tabulek, které jsou znázorněny na obrázku 4.9: "User", "Auto", "Blacklist_tokens" a "Information". Do tabulky "User" jsou ukládány údaje o uživateli. Kromě toho tvoří tato tabulka oddělenou část databáze, protože uživatelé nejsou předem známi; stejně jako "Blacklist_tokens", do které se ukládají autorizační tokeny vyvolané po odhlášení uživatele. Tabulka "Auto" obsahuje základní informace o automobilu, jako jsou jeho identifikační číslo a jméno a příjmení majitele. "Information" slouží pro ukládání dat z automobilu. Kromě dat, která reprezentují informace z různých snímačů, tato tabulka obsahuje také položku "status", která má výčtový typ a slouží pro ukládání informací o stavu vozidla (například: vozidlo je poškozeno, má příliš vysokou teplotu atd.). Pro lepší pochopení databázového modelu ve Flasku bude dále rozebrán jeden



Obrázek 4.9: Serverová databáze.

model (třída), konkrétně tabulka "Auto", která reprezentuje informace o vozidle a je zobrazena na obrázku 4.10. Stejným způsobem jsou zpracovány ostatní tabulky.

- *Id* — je primární klíč pro dané auto. Má celočíselný datový typ Integer;
- *Vin* — slouží pro ukládání identifikačního čísla vozidla. Musí být unikátní a není možné, aby byla daná položka nevyplněná;

```python
class Auto(db.Model):
    """ Auto Model for storing car related details """
    __tablename__ = "auto"

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    vin = db.Column(db.String(255), unique=True, nullable=False)
    owner_first_name = db.Column(db.String(255), nullable=False)
    owner_last_name = db.Column(db.String(255), nullable=False)
    information = db.relationship('Information', backref='information', lazy="joined", uselist=False)

    def __init__(self, owner_first_name, owner_last_name, vin):
        self.owner_first_name = owner_first_name,
        self.owner_last_name = owner_last_name,
        self.vin = vin

    def __repr__(self):
        return '<Auto {}>'.format(self.body)
```

Obrázek 4.10: Příklad databázového modelu.

- *Owner_first_name* — uchovává jméno vlastníka automobilu;
- *Owner_last_name* — uchovává příjmení vlastníka automobilu;
- *Information* — funkce, která reprezentuje vztah mezi tabulkami. V daném případě se tabulka "Auto" vztahuje k tabulce "Information". Parametr *uselist=False* značí, že vztah mezi těmito tabulkami je 1:1. *Lazy="joined"* je parametrem, který přikazuje, aby používaný vztah načetl data pomocí příkazu JOIN.

Každý model obsahuje své vlastní předdefinované metody, kromě toho lze vytvářet vlastní metody, které mají svou specifickou funkci. V daném případě existují dvě metody:

- ___*repr*___ — slouží pro vrácení těla objektu (vozidla);
- ___*init*___ — používá se pro inicializace třídy.

Databázový systém je navržen tak, aby poskytoval možnost jeho dalšího rozšíření, konkrétně se jedná o zpracování dat z automobilu. Na obrázku 4.9 jsou tyto data pojmenovány jako první, druhý a třetí parametr. Toto označení přispívá ke snadnému a pochopitelnému znázornění, a také k tomu že se jednotka může stále vyvíjet a že se mohou tyto parametry v různých generacích a etapách lišit.

4.6.2 Klientská databáze

Operační systém Android umožňuje přístup k relačnímu databázovému systému SQLite. SQLite je podobný systému Oracle, MySQL, PostgreSQL a SQL Server. Na rozdíl od uvedených databází však nepodporuje model "klient-server". To znamená, že SQLite je integrována do konečného programu a uživatel ho může připojit k aplikaci a tím získat přístup ke všem databázovým funkcím. Databáze pro aplikace představuje lokální databázový systém, který se skládá ze 3 tabulek: "User", "Car" a "Compar", viz obrázek 4.11. Do tabulky "User" se ukládají data ze serverových

Obrázek 4.11: Klientská databáze.

odpovědí po autorizaci uživatele. ”Car” obsahuje data z odpovědí, které následují po požadavku doplnit data o konkrétním vozidle.

Kvůli tomu, že použitá databáze je lokálním databázovým systémem, může dojít ke kompromitaci dat. To znamená, že když se na jednom zařízení přihlásí další uživatel, bude mít k dispozici informace o automobilu svého ”předchůdce”. Tento problém lze řešit mazáním databází mezi jednotlivými uživatelskými připojeními. V konkrétním případě to bude vypadat tak, že e-mail uživateli bude po ukončení používání aplikace uložen do zvláštní tabulky ”Compar” a po následném přihlášení bude uvedený e-mail porovnán s dříve uloženou adresou. Jestliže mobilní klient zjistí, že jsou vyplněné e-mailové adresy různé, databáze bude smazána a do srovnávací tabulky bude uložena nová e-mailová adresa.

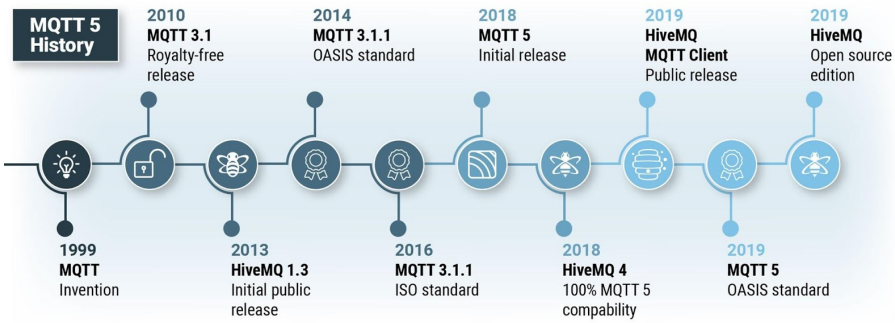
4.6.3 SharedPreferences

Kromě databáze přístupné v mobilní aplikaci byl použit způsob ukládání dat SharedPreferences. Tento přístup slouží pro perzistentní ukládání dat typu klíč-hodnota při ukládání autorizačního tokenu přihlášeného uživatele, který bude ze serverové databáze ”smazán” po odhlášení (jinými slovy bude uložen do tabulky ”Blacklist_tokens” jako neplatný).

4.7 MQTT protokol

MQTT (Message Queue Telemetry Transport) – je specializovaný protokol určený pro přenos dat na vzdálená místa. Používá se především v systémech M2M (machine-to-machine) a IIoT Systems (Industrial Internet of Things). MQTT je

standardem OASIS (Organization for the Advancement of Structured Information Standards) a specifikace protokolu je spravována technickou komisí OASIS MQTT. Vývoj MQTT protokolu lze vidět na obrázku 4.12.



Obrázek 4.12: Vývoj MQTT protokolu [16].

MQTT protokol má čtyři verze [20]:

- *MQTT-SN v1.2* — dříve známý jako MQTT-S, standard IBM, viz [24]. Protokol pro senzorové sítě, který byl navržen speciálně pro práci v bezdrátových sítích, a pokud je to možné, aby fungoval stejným způsobem jako MQTT. MQTT-SN zvyšuje účinnost přenosu dat i spotřeby energie. Tato verze také obsahuje proceduru, která umožňuje zařízením přejít do režimu spánku, když nejsou potřeba, a přijímat jakékoli informace, které na ně čekají, když se probudí.

- *MQTT 3.1 Specification* — standard Eurotech a IBM. Hlavní rozdíly oproti předchozí verzi jsou řetězce v MQTT podporující plnou UTF-8, uživatelské jméno a heslo lze odeslat pomocí paketu CONNECT, viz [21].

- *MQTT 3.1.1 Specification* — starší standard ISO a OASIS. Obecně platí, že MQTT 3.1.1 je zlepšení specifikace 3.1. Protokol verze 3.1 měl limit 23 bytů pro ID klienta, což bylo velmi nepohodlné a vedlo k mnoha problémům, například při použití UUID (Universally unique identifier) pro identifikátory klienta. Další změny jsou název protokolu – v záhlaví CONNECT se změnil z MQIsdp na MQTT, všechna kódování řetězců jsou konzistentně UTF-8 a bajt na úrovni protokolu byl zvýšen ze 3 na 4. Podrobnou specifikaci lze najít na stránkách [22].

- *MQTT 5 Specification* — standard OASIS. Ve verzích MQTT protokolu starších než 5 mohl odesílat odpojovací paket pouze klient. To znamenalo, že v každém případě, kdy server chtěl ukončit konverzaci s klientem, bylo nutné ukončit TCP připojení. V MQTT 5 může klient chvíli počkat, než se pokusí o opětovné připojení, protože ví, že server nemusí být vždy k dispozici. Také v dané verzi protokolu existuje koncept sdílených témat, což umožňuje vyrovnávání zatížení. Zprávy o těchto tématech jsou odesílány do jedné skupiny

účastníků, ostatní se mohou přihlásit k odběru pomocí speciálního tématu. Kromě toho v páté verzi protokolu může server inzerovat omezení funkčnosti, kterou poskytuje. O dalších funkcích lze přečíst na [23].

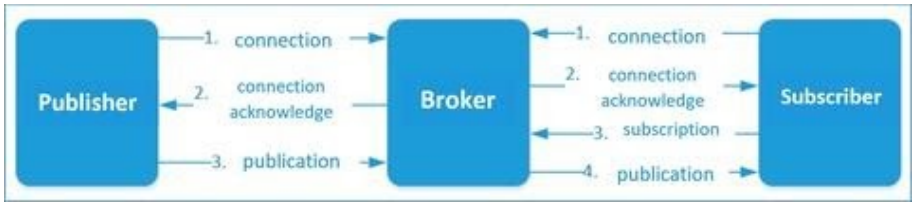
Vlastnosti protokolu MQTT:

- asynchronní;
- přenáší kompaktní zprávy;
- umožňuje provoz v podmínkách nestabilního připojení;
- podporuje několik úrovní kvality služeb (QoS);
- umožňuje snadnou integraci nových zařízení.

Výpočetní požadavky na protokol MQTT jsou velmi malé, protože je navržen pro nízkoenergetická vestavěná zařízení. I když je šířka pásma sítě nízká, MQTT udržuje vysoce kvalitní komunikaci a prakticky nepřetíží systém. To je jedna z hlavních výhod tohoto protokolu. Ve srovnání s jinými komunikačními protokoly nejsou v datové struktuře téměř žádné funkční informace, které protokol přenáší (například HTTP přenáší veškerá data služeb).

Tento protokol funguje na bázi „poskytovatel–příjemce" s minimálním počtem možností implementace úkolů. Takový způsob komunikace zlepšuje a zrychluje fungování samotného protokolu. Charakteristickým rozdílem takového typu komunikace od typu „klient–server" je, že klienti odesílající zprávy a klienti přijímající zprávy jsou obvykle odděleni:

- operace na obou stranách by neměly být pozastaveny během odesílání nebo přijímání informací;
- poskytovatel a příjemce mohou pracovat v různých časech;
- poskytovatel a příjemce se nemusí navzájem znát.



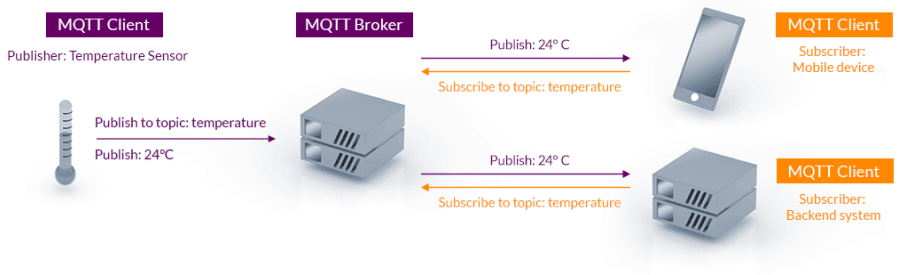
Obrázek 4.13: MQTT protokol [29].

Všechny úkoly jsou prováděny prostřednictvím spolupráce s brokerem a vedou k práci s různými tématy a signály. Vykonávající klienti poté kontaktují broker a buď tam zveřejní zprávy a témata, nebo se přihlásí k odběru témat. Jak už bylo řečeno, komunikační systém postavený na MQTT se skládá z poskytovatele zpráv (publisher), brokeru zpráv (broker) a příjemce zpráv (subscriber), viz obrázek 4.13.

- *Publisher* — odesílá zprávy –jsou to například senzory z teploměrů a dalších zařízení IoT;
- *Broker* — centrální rozbočovač MQTT, který je zodpovědný za komunikaci mezi poskytovatelem a příjemcem zpráv. Broker přijímá data od publisheru, zpracovává je a pak předává data subscriberu. Broker také kontroluje doručení. Jako broker obvykle funguje serverový software (MQTT Server) nebo řadič;
- *Subscriber* — hlavní příjemce dat ze senzorů.

Zjednodušený komunikační proces lze popsat takto, viz obrázek 4.14:

- Publisher odešle datovou zprávu (například informace z teplotních senzorů) brokeru s uvedením tématu, k němuž data patří (například „Teplota“).
- Broker analyzuje, kteří z subscriberů mají předplatné určitých témat, v tomto případě tématu „Teplota“.
- Subscribery, kteří jsou přihlášeni k odběru tématu „Teplota“, obdrží od brokeru zprávu s informacemi z teplotních čidel.



Obrázek 4.14: Komunikační proces [12].

V případě komunikace s BBCU jednotkou v roli publisheru je samotná jednotka, brokerem je hlavní dispečink, subscriberem je sekundární dispečink. Mnoho subscriberů se tedy může přihlásit k odběru různých témat a v závislosti na těchto předplatných dostávat potřebné informace bez přímé komunikace s publisherem. K sekundárnímu dispečinku se stále připojují mobilní zařízení, která dostávají informace o vozidle, ovšem pouze ty, ke kterým je přihlášen dispečink.

Témata mají podobu znaků s kódováním UTF-8. Jejich hierarchie má stromovou podobu, což usnadňuje organizaci a přístup k datům. Každé téma má alespoň jednu úroveň. Jednotlivé úrovně jsou odděleny lomítkem. V případě komunikace s BBCU jednotkou vypadají jednotlivé zprávy takhle: *“BBCU/”+str(BBCU_ID)+”/telemetry/fuelLevel”*, kde lze vidět, že publisher odesílá informace o palivu. Pokud by bylo nutné dostat informace ze všech čidel, která jsou používána, bylo by vhodné použít zástupný znak (#) a výsledná zpráva by vypadala následovně: *“BBCU/”+str(BBCU_ID)+”/telemetry/#”*. V takovém případě by subscriber dostal informace o množství paliva, rychlosti, teplotě a další.

Pokud několik čidel měří například teplotu, lze také použít zástupný znak (+): *"BBCU/"+str(BBCU_ID)+"/+/temperature"*. V takovém případě by subscriber dostal data o teplotě, ale ze všech možných senzorů.

Zařízení MQTT používají určité typy zpráv ke komunikaci s brokerem, a to například:

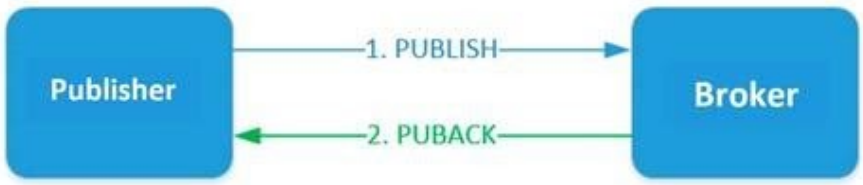
- *connect* — slouží pro navázání spojení s brokerem;
- *disconnect* — používá se pro přerušení komunikace s brokerem;
- *subscribe* — používá se pro přihlášení k odběru témat;
- *unsubscribe* — používá se pro přihlášení k odběru témat;
- *publish* — používá se pro publikování zpráv do tématu.

Přenášená data mají různé hodnoty a priority doručení, takže MQTT poskytuje tři úrovně kvality služeb - QoS (Quality of Service):

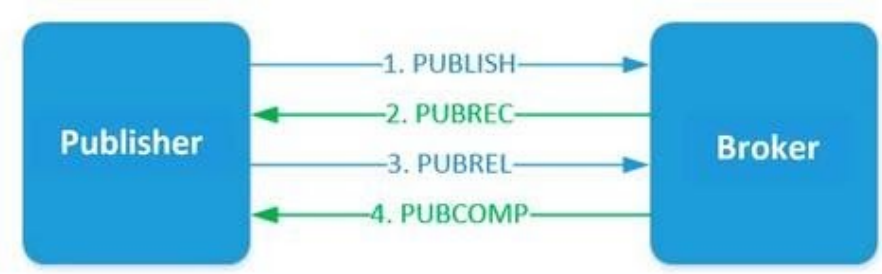
- *QoS 0 (At most once)* — zpráva je doručena maximálně jednou a publisher nečeká na žádnou odpověď. Pokud je doručení přerušeno, může dojít ke ztrátě zprávy – nedojde k žádným pokusům o opakování, viz obrázek 4.15.

Obrázek 4.15: QoS 0 [29].

- *QoS 1 (At least once)* — zpráva je doručena alespoň jednou a příjemce doručení potvrdí. V takovém případě existuje možnost duplicitních zpráv, viz obrázek 4.16.



Obrázek 4.16: QoS 1 [29].



Obrázek 4.17: QoS 2 [29].

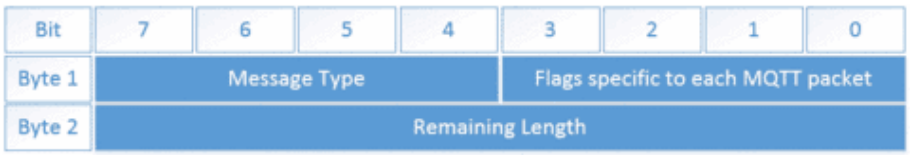
- *QoS 2 (Exactly once)* — zpráva je doručena pouze jednou, a to navzdory problémům a rušení. Z důvodu selhání může dojít ke zpoždění doručení, ale zpráva se stále dostane k adresátovi, například po obnovení připojení, viz obrázek 4.17.

Čím horší je kvalita připojení, tím obtížnější je QoS. Proto se QoS 0 používá, když je velké množství dat, která přenáší pravidelně, a ztráta jedné nebo několika zpráv nic neovlivní. Například senzor přenáší data o teplotě zařízení jednou za sekundu, ale pro analýzu se používá pouze denní průměr. V takovém případě je ztráta několika zpráv zanedbatelná. Ale při přenosu nějakých finančních údajů jsou ztráty nepřijatelné, protože zůstatek na účtu klienta nebude konvergovat.

Zpráva MQTT se skládá z několika částí:

- Fixní hlavička (přítomná ve všech zprávách).
- Variabilní hlavička (přítomná pouze v určitých zprávách).
- Data, „náklad” (přítomné pouze v určitých zprávách).

Na obrázku 4.18 je znázorněna zpráva s fixní hlavičkou, protože klíčové vlastnosti protokolu MQTT jsou realizovány přesně pomocí polí tohoto záhlaví. První bajt záhlaví obsahuje čtyři pole, z nichž tři jsou speciální příznaky (DUP, QoS Level, RETAIN), čtvrté označuje typ zprávy.



Obrázek 4.18: Zpráva MQTT [29].

- *DUP (Duplicate)* — duplicitní zpráva. Tento příznak označuje příjemci, že zpráva, kterou obdržel, je znovu vysílána. Používá se v PUBLISH, SUBSCRIBE, UNSUBSCRIBE, PUBREL. Tento příznak hraje důležitou roli při přenosu informací přes nespolehlivé kanály, kde je možná ztráta signálu.

- *QoS* — jak uvedeno výše, hlavním rozlišovacím znakem protokolu MQTT je schopnost používat různé úrovně služeb, které jsou určeny hodnotou tohoto příznaku.
- *RETAIN* — při publikování dat s nastaveným příznakem Retain, broker to uloží. Při dalším odběru na toto téma broker okamžitě odešle zprávu s tímto příznakem. Používá se pouze ve zprávách typu PUBLISH.

Druhý bajt se používá k označení zbývající délky zprávy, která je součtem velikosti záhlaví proměnné (pokud existuje) a velikosti užitečného zatížení.

Použití protokolu MQTT:

- *Monitorovací systémy zařízení.*

  Ve velkých továrnách jsou inteligentní senzory instalovány na strojích, transformátorech a tak dále. Senzory monitorují provoz průmyslových zařízení a přenášejí data do centrálního analytického systému. To umožňuje sledovat výkon zařízení v reálném čase, předvídat opotřebení a hodnotit jejich výkon.

- *Environmentální a strukturální monitorovací systémy.*

  MQTT se používá k analýze klimatických vlastností, seismické aktivity a odolnosti budov. To umožňuje předpovídat přírodní katastrofy a kataklyzmata a předcházet ničení budov.

- *Vysoce spolehlivé systémy pro práci s důležitými daty.*

  Ačkoli byl protokol MQTT původně vytvořen pro IIoT, používá se i při fakturaci mobilních operátorů a poskytovatelů. Je pro ně důležité přenášet informace o pohybu peněz na zákaznických účtech.

Při implementaci MQTT protokolu byla použita python knihovna AWSIoTPythonSDK, kterou lze nainstalovat pomocí příkazu 4.6. AWSIoTPythonSDK slouží pro připojení IoT zařízení ke službám AWS. Tato knihovna poskytuje základní synchronní operace MQTT spolu s konfiguracemi hlavních funkcí:

- automatické opětovné připojení, resubscribe;
- postupné připojení zpět;
- offline publikování požadavků.

Kód 4.6: Implementace knihovny AWSIoTPythonSDK.

```
1 pip install flask-restplus
```

Syntax [9]:

Kód 4.7: Syntax knihovny AWSIoTPythonSDK.

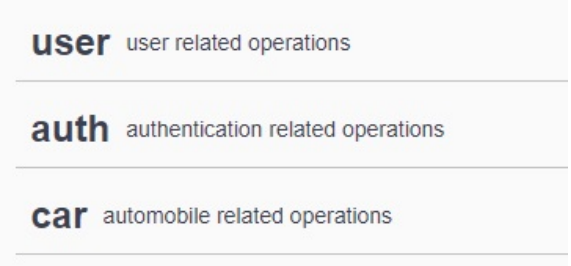
```
1 import AWSIoTPythonSDK.MQTTLib as AWSIoTPyMQTT
2 # Create an AWS IoT MQTT Client using TLSv1.2 Mutual ↩
     Authentication
3 myAWSIoTMQTTClient = AWSIoTPyMQTT.AWSIoTMQTTClient("↩
     testIoTPySDK")
4 # Create an AWS IoT MQTT Client using Websocket SigV4
5 myAWSIoTMQTTClient = AWSIoTPyMQTT.AWSIoTMQTTClient("↩
     testIoTPySDK", useWebsocket=True)
```

5 Implementace

Na základě návrhu byla provedena implementace webové služby a samotné mobilní aplikace. Daný proces bude podrobně popsán v této kapitole.

5.1 Server

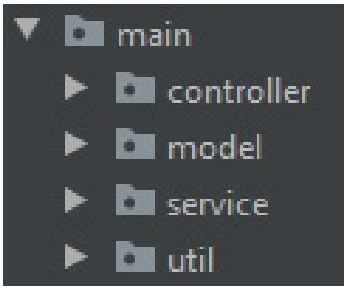
Jak bylo již zmíněno, webová služba je implementována v programovacím jazyce Python s použitím vývojového prostředí PyCharm 2018.3.4. Pro vizualizaci byl použit systém Swagger. Server přijímá a zpracovává požadavky ve formátu JSON a ve stejném formátu je následně odpověď odesílána klientovi. Implementovanou webovou službu lze rozdělit do třech logický částí (operací), které jsou znázorněny na obrázku 5.1.



Obrázek 5.1: API operace.

- *User* — obsahuje operace, které se přímo týkají uživatele. Konkrétně se jedná o registraci nového uživatele do systému, přístup k seznamu všech uživatelů a vyhledávání uživatele podle e-mailové adresy;
- *Auth* — je zásobníkem pro autorizační operace (přihlášení a odhlášení);
- *Car* — odpovídá za operace, které jsou zaměřeny na získávání dat o vozidle (zobrazení seznamu všech automobilů v databázi, přidání vozidla do mobilní aplikace, vyhledávání a následné zobrazení vozidla podle identifikačního čísla).

Mobilní aplikace nepoužívá všechny výše popsané metody. Například metody pro zobrazení seznamu byly implementovány pouze za účelem sledování dat v databázovém systému. Samotný projekt je rozdělen do čtyř částí (viz. obrázek 5.2): *controller, model, service, util*. Složka *model* poskytuje data a operace pro práci s nimi. Mohou



Obrázek 5.2: Rozdělení Python projektu na součástí.

to být různé dotazy do databáze. Model tedy v podstatě poskytuje přístup k datům a umožňuje jejich správu. Jsou zde umístěny soubory, které odpovídají databázovým tabulkám popsaným v kapitole 4.6.1.

Složka *controller* je zodpovědná za interpretace akcí vykonaných uživatelem a za následné upozornění o změnách. Jinými slovy controller zajišťuje komunikaci mezi systémem a uživatelem.

Service je složkou pro soubory, ve kterých jsou definovány funkce pro určitý cíl. Jedná se o pět souborů:

- *auth_helper* — slouží pro funkce, které se týkají autorizačního procesu (přihlášení a odhlášení). Je zde popsané chování webové služby po příjmu požadavku pro daný účel. Použitou funkcí definovanou v souboru je controller, který je určený pro výše uvedený proces. Tato skutečnost je znázorněna na obrázku 5.3;
- *car_helper* — má stejné využití jako *auth_helper*, je ale zaměřený na metodu přidání auta do mobilní aplikace;
- *data_service* — obsahuje funkce pro získání dat podle určitého kritéria. Může se jednat o získání dat podle identifikačního čísla nebo podle provozovatele;
- *user_service* — používá se pro stejné účely jako *data_service*, funkce jsou ale zaměřeny na manipulaci s uživatelem;
- *blacklist_service* — popisuje pouze jednu funkci, která se zabývá odstraněním autorizačního tokenu po odhlášení uživatele.

Util obsahuje pouze jeden soubor pro modely, které budou popsány v kapitole 5.1.2.

5.1.1 Způsob kontroly údajů

Kontrola údajů je důležitou funkcí webové služby. Aby mohl uživatel obdržet data o svém automobilu, bylo použitý způsob verifikace pomocí údajů dříve uvedených uživatelem.

Jako základ pro výběr způsobu kontroly bylo zvoleno osvědčení o registraci zvláštního motorového vozidla a zvláštního přípojného vozidla, což je znázorněno na obrázcích 5.4 a 5.5. Osvědčení obsahuje položky jako je registrační značka,

```python
@api.route('/login')
class UserLogin(Resource):
    """
        User Login Resource
    """
    @api.doc('user login')
    @api.expect(user_auth, validate=True)
    def post(self):
        # get the post data
        post_data = request.json
        return Auth.login_user(data=post_data)
```

```python
class Auth:
    @staticmethod
    def login_user(data):
        try:
            user = User.query.filter_by(email=data.get('email')).first()
            if user and user.check_password(data.get('password')):
                auth_token = user.encode_auth_token(user.id)

                if auth_token:
                    response_object = {
                        'status': 'success',
                        'first_name': user.first_name,
                        'last_name': user.last_name,
                        'email': user.email,
                        'phone': user.phone,
                        'Authorization': auth_token.decode()
                    }
                    return response_object, 200
            else:
                response_object = {
                    'status': 'fail',
                    'message': 'email or password does not match.'
                }
                return response_object, 401

        except Exception as e:
            print(e)
            response_object = {
                'status': 'fail',
                'message': 'Try again'
            }
            return response_object, 500
```

Obrázek 5.3: Příklad použití funkce ze souboru auth_helper (zprava) v příslušném controlleru (zleva).

Obrázek 5.4: Vzor osvědčení o registraci zvláštního motorového vozidla a zvláštního přípojného vozidla. Lícová strana [1].

Obrázek 5.5: Vzor osvědčení o registraci zvláštního motorového vozidla a zvláštního přípojného vozidla. Rubová strana [1].

provozovatel, rodné číslo, identifikační číslo vozidla a jiné. Na základě těchto údajů bylo rozhodnuto použít pro funkci přidání automobilu data jako je jméno a příjmení provozovatele a VIN vozidla.

To znamená, že když server dostane od klienta požadavek obsahující identifikační číslo vozidla, jméno a příjmení provozovatele, odešle zpět odpověď odpovídající požadavku. Pokud jsou zadaná data správná, odpověď bude obsahovat informace o vozidle. Pokud ne, klient obdrží chybové hlášení.

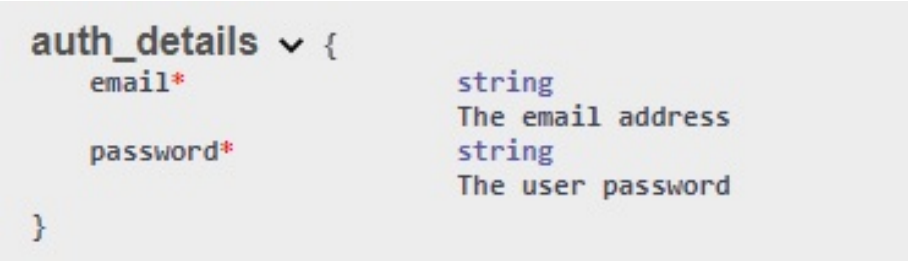
Správnost přijatých dat server ověřuje pomocí jejich porovnávání s daty, které

má již uložené ve své databázi. Tento návrh byl popsán v kapitole 4.6.1.

5.1.2 Modely

Pro zpracování požadavků a následné odesílání odpovědí se používají takzvané modely, konkrétně DTO. Webová služba obsahuje celkem čtyři modely: *user, auth_details, details a car.*

- *User* je modelem, který obsahuje položky, které uživatel potřebuje pro registraci. Lze říct, že každá položka odpovídá jednomu řádku příslušné databázové tabulky. Jedná se o jméno, příjmení, e-mail, telefonní číslo a heslo;
- *Auth_details* — model, který odpovídá datům potřebným pro autorizaci. Konkrétně se jedná o e-mailovou adresu a heslo. Tento model si lze prohlédnout na obrázku 5.6;
- *Details* je modelem, který se skládá z položek dvou databázových tabulek: Auto a Information. Odpověď v podobě tohoto modelu odesílá server klientovi, když obdrží požadavek na poskytnutí dat;
- *Car* je posledním modelem a obsahuje položky potřebné pro přidání vozidla (jméno a příjmení majitele, identifikační číslo auta).



Obrázek 5.6: Ukázka Auth_details.

Odpovídající model, napsaný v Pythonu, je znázorněn na obrázku 5.7. Lze zde vidět, že se třída AuthDTO skládá ze dvou částí: první slouží pro definování API (v daném případě bude výsledný URL obsahovat jako svou součást ”auth”, což bylo znázorněni již na obrázku 5.7. Druhá část definuje samotný model ”auth_details”, který se používá při autorizaci uživatele. Obě položky jsou označeny jako povinné k vyplnění. Pro každý z výše uvedených modelů existuje podobná reprezentace.

Obrázek 5.7: Ukázka AuthDTO.

5.2 Klient

Jak bylo zmíněno již v předchozích kapitolách, aplikace je implementovaná v programovacím jazyce Java s použitím vývojového prostředí Android Studio 3.6.2. Pro sestavení aplikace byl použit systém Gradle. Hlavní informace pro sestavení aplikace jsou uvedeny na obrázku 5.8.

```
defaultConfig {
    applicationId "com.example.fab"
    minSdkVersion 16
    targetSdkVersion 29
    versionCode 1
    versionName "1.0"

    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}

buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
}
```

Obrázek 5.8: Soubor build.gradle.

Jako povolení pro aplikace v manifestu jsou uvedeny:

- <uses-permission android:name=”android.permission.INTERNET” />
- <uses-permission android:name=”android.permission.VIBRATE” />

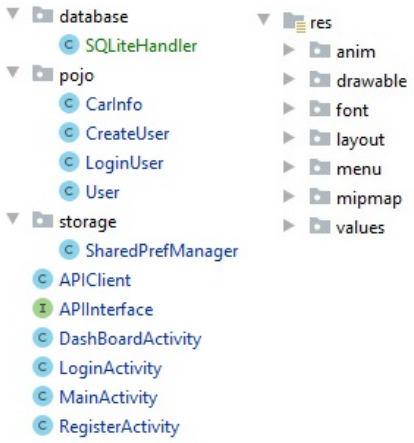
5.2.1 Organizace aplikace

Celková organizační struktura aplikace je znázorněna na obrázku 5.9.

Ve složce ”database” se nachází jeden Java soubor, který je zodpovědný za všechny akce, které se týkají databázového systému. Třída SQLiteHandler implementuje lokální databázi, která byla popsána v kapitole 4.6.2, a je rozšířena třídou SQLiteOpenHelper.

Složka ”pojo” obsahuje soubory, se kterými pracuje Retrofit. Mobilní aplikace má celkem čtyři soubory. Konkrétní případy jejich použití a vztah s definovanými metodami lze vidět na obrázku 5.9. Kromě toho třídy *User* a *CarInfo* reprezentují serverovou odpověď na příslušný požadavek mobilního klienta. Třídy *LoginUser* a *CreateUser* se používají pro stejnojmenné akce. Ukázku třídy používané pro přihlášení lze vidět na obrázku 5.10.

Složka ”storage” obsahuje (stejně jako databázová složka) jenom jednu třídu SharedPrefManager, která slouží pro ukládání autorizačního tokenu, o čemž už bylo referováno v kapitole 4.6.3.



Obrázek 5.9: Organizace aplikace.

```
public class LoginUser {

    @SerializedName("email")
    public String email;
    @SerializedName("password")
    public String password;

    public LoginUser(String email, String password) {
        this.email = email;
        this.password = password;
    }
}
```

Obrázek 5.10: Ukázka POJO třídy ”LoginUser”.

Jak již bylo zmíněno v kapitole 4.3.1, pro komunikaci s REST API byla použita knihovna Retrofit. Její implementace je popsána ve třídě *APIClient*, která následně implementuje rozhraní *APIInterface*. Metody, které používá aplikace, jsou definovány v rozhraní a znázorněny na obrázku 5.11.

APIInterface obsahuje pět základních metod: metodu pro přihlášení, pro registraci, pro přidání vozidla, pro odhlášení a pro získávání dat o vozidle. Jednotlivé aktivity budou podrobně popsány v kapitole 5.2.2.

Adresář *res* obsahuje sedm složek:

- *res/anim* obsahuje XML soubory, které popisuje chování tlačítek;
- *res/drawable* obsahuje XML soubory, které definují grafický vzhled různých prvků aplikace (například grafické objekty, nastavení okrajů tlačítek a tak dále);
- *res/font* slouží pro ukládání XML vzhledů potřebných písem;
- *res/layout* obsahuje XML soubory pro vzhled jednotlivých obrazovek. Každý soubor v této složce odpovídá příslušné aktivitě;
- *res/menu* slouží pro ukládání XML souborů, které popisují vzhled menu v jednotlivých aktivitách;
- *res/mipmap* obsahuje hlavní ikonu aplikace;
- *res/values* oobsahuje XML soubory pro různá data. Jsou zde takové soubory jako colors.xml (definuje barvy), dimens.xml (definuje rozměry), font_certs.xml (definuje stažená písma), integers.xml (slouží pro definování celočíselných hodnot), preloaded_fonts.xml (definuje předinstalovaná písma), strings.xml (slouží pro definování řetězců uživatelského rozhraní) a styles.xml (definuje vzhled UI).

```
@POST("/auth/login")
Call<User> loginUser(@Body Map<String, String> jsonObject);

@POST("/user/")
Call<User> createUser(@Body CreateUser createUser);

@GET("/auto/{vin}")
Call<CarInfo> getCar(@Path("vin") String vin);

@POST("/auto/data")
Call<CarInfo> getCarByData(@Body Map<String, String> jsonObject);

@POST("/auth/logout")
Call<User> logoutUser(@Header("Authorization") String Authorization);
```

Obrázek 5.11: Ukázka používaných metod.

5.2.2 Aktivity

Mobilní aplikace má celkem čtyři aktivity. Každé z nich je přidělena jedna obrazovka. Tyto obrazovky byly navrhnuty v kapitole 4.5: *LoginActivity, RegisterActivity, MainActivity a DashBoardActivity*. Jejich finální vzhled bude podrobně rozebrán dále, v kapitole 6.1.

LoginActivity je aktivitou, která se spustí, když uživatel zapne aplikaci. Obsahuje tlačítka pro potvrzení uvedených údajů (login) a pro přesměrování na obrazovku RegisterActivity (register). Dále jsou na obrazovce rozmístěna políčka pro zadání e-mailové adresy a hesla. Políčko sloužící pro zadání hesla má atribut textPassword a políčko pro e-mail má atribut textEmailAddress. Atribut textPassword způsobuje, že zadávané údaje nejsou přímo zobrazeny, což přispívá k bezpečnosti. Po stisknutí tlačítka «login» se odešle požadavek na server s uvedenými údaji. Po úspěšné odpovědi je uživatel přesměrován na obrazovku MainActivity. Pokud klient dostane chybovou odpověď, zobrazí se mu zpráva Toast s hláškou „přihlášení selhalo“. Po úspěšné autentifikaci jsou také uloženy data do lokální databáze data ze serverové odpovědi a autorizační token je uložen do SharedPreferences.

RegisterActivity je obrazovkou, která slouží pro registraci uživatele. Obsahuje všechna políčka potřebná pro registraci dat: jméno, příjmení, e-mailovou adresu, telefonní číslo a heslo. Po stisknutí tlačítka «register» budou uvedená data na server odeslána v JSON formátu. Server je následně uloží do databáze a uživatel se vrátí na původní obrazovku (LoginActivity). Jestliže bude některé z políček prázdné, zobrazí se oznámení (provázené vibrací) o tom, že některá data byla zadána chybně.

MainActivity slouží pro zobrazení informací o přihlášeném uživateli, které aplikace obdrží ze serveru: jméno, příjmení, e-mail a telefonní číslo. Je to stránka, která se zobrazí uživateli ihned po úspěšném přihlášení. Zde je také implementované dolní navigační menu se třemi položkami: jedna pro přesměrování na obrazovku DashBoardActivity, druhá označuje danou obrazovku a třetí slouží pro odhlášení uživatele. Po stisknutí odhlašovací položky se zobrazí upozornění dialog a systém se uživatele

zeptá, zda se opravdu chce odhlásit. Pokud uživatel vybere ”ano”, aplikace odešle požadavek na server s autorizačním tokenem a po následném úspěšném odhlášení se otevře LoginActivity a server smaže token. Jinak bude dialog uzavřen a uživatel se vrátí na obrazovku, ze které tuto funkci vyvolal.

DashBoardActivity je aktivitou pro zobrazení dat o vozidle. Stejně jako MainActivity obsahuje dolní navigační menu, které funguje obdobně. Kromě toho je zde implementováno plovoucí tlačítko. Po jeho stisknutí se zobrazí další dvě tlačítka: jedno pro přidání vozidla, druhé pro aktualizaci dat. Když uživatel stiskne tlačítko pro přidání vozidla, zobrazí mu dialog s jedním políčkem pro vložení identifikačního čísla vozidla. Při návrhu bylo zmíněno, že dialog bude obsahovat tři políčka pro údaje. Při implementaci bylo nicméně rozhodnuto použít jen jedno a ostatní data si vyžádat z lokální databáze. Uživatel tedy nebude muset pro kontrolu uvádět své jméno a příjmení. Po potvrzení uvedených data odešle klient požadavek na server. Pokud byla data zadána správně, dostane odpověď obsahující potřebné informace, které budou uloženy do lokální databáze. V opačném případě obdrží uživatel chybové hlášení a bude muset proces provést znovu. Po stisknutí tlačítka, které slouží pro aktualizaci, odešle klient ještě jeden požadavek na server. Po obdržení odpovědi budou data v databázi aktualizovány a v případě špatného stavu vozidla dostane uživatel notifikaci o tom, že je s jeho autem něco v nepořádku.

6 Testování a vyhodnocení

V této kapitole bude popsáno testování implementovaného softwaru z předchozí kapitoly. Mezi testovanými položkami jsou základní funkce uživatele jako je přihlášení a registrace, kontrola uvedených údajů a příjem odpovědí ze serveru s následným zpracováním této serverové zprávy.

6.1 Zkoušení softwaru

Testování mobilní aplikace probíhalo na emulátoru a na několika mobilních zařízeních s různými verzemi operačního systému Android. Konkrétně se jednalo o Android 7.0 (API verze 24) a o Android 8.0 (API verze 26), které jsou zahrnuty do zvoleného rozmezí pro požadovanou verzi mobilního klienta (jako minimální verze byla zvolena 7.0, což bylo objasněno v kapitole 3.1).

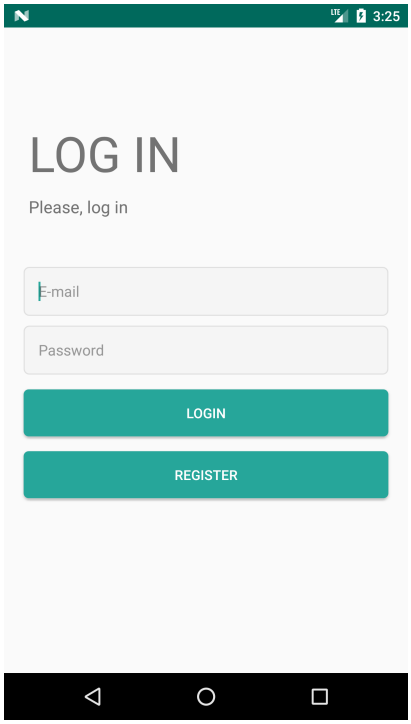
Na obrázcích 6.1 a 6.2 jsou znázorněny výsledné obrazovky UI pro přihlášení a pro registraci uživatelů. Mají téměř stejný design, ale obrazovka pro registraci obsahuje více polí k vyplnění. Pokud bude některé z registračních polí prázdné, uživatel dostane upozornění v podobě vibrace. Po správném vyplnění formulářů a následné serverové kontrole dostává mobilní klient odpověď ve formátu JSON, viz ukázka 6.1.

Kód 6.1: JSON odpověď.

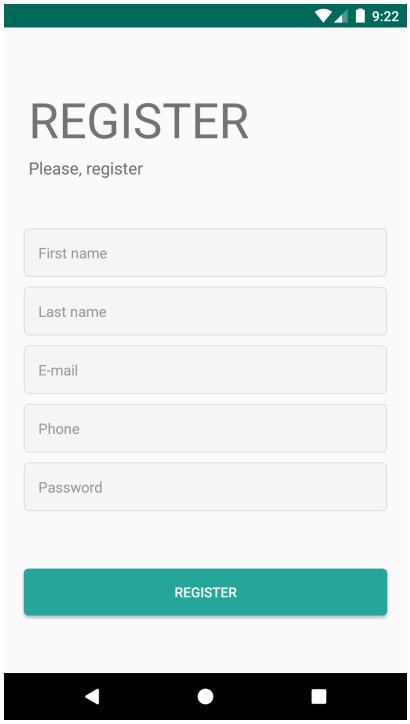
```
1 {
2     "status": "success",
3     "first_name": "test",
4     "last_name": "test",
5     "email": "user@blog.com",
6     "phone": 1548962,
7     "Authorization": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9"
8  }
```

Jestliže dojde k chybě, odpověď ze serveru bude obsahovat zprávu o tom, že použitá data jsou špatná (6.2) a notifikaci informující o tom, že přihlášení nebylo úspěšné.

Obrázek 6.3 ukazuje, jak vypadá obrazovka po úspěšné autentifikaci. Jak bylo navrhnuto v kapitole 4.5 a následně implementováno v kapitole 5.2.2 tato obrazovka obsahuje data uživatele: jméno, příjmení, telefonní číslo a e-mail. V této verzi není možné data měnit; slouží pouze ke kontrole.



Obrázek 6.1: Obrazovka pro LoginActivity.



Obrázek 6.2: Obrazovka pro RegisterActivity.

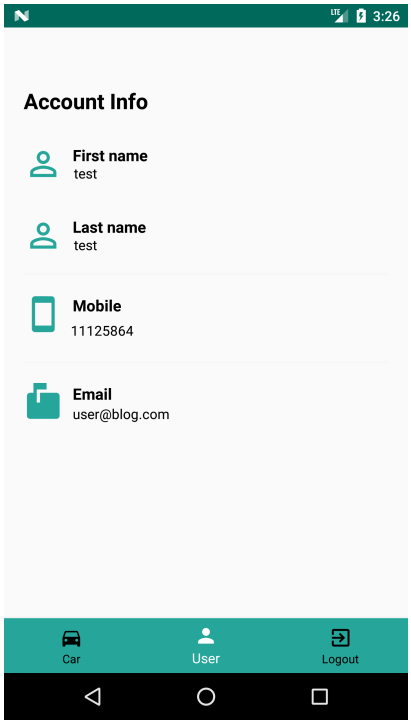
Kód 6.2: JSON odpověď v případě neúspěchu.

```
{
    "status": "fail",
    "message": "e-mail or password does not match."
}
```

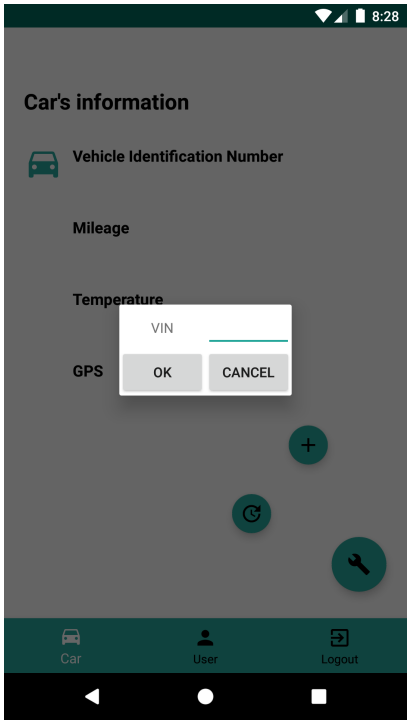
Kód 6.3: JSON odpověď s data vozidla.

```
{
    "vin": "1542369856",
    "owner_first_name": "test",
    "owner_last_name": "test",
    "mileage": "1956",
    "temperature": "60",
    "gps": "47.407614681869745, 8.553115781396627",
    "status": null
}
```

Na obrázku 6.4 lze vidět obrazovku, která odpovídá aktivitě DashBoard z kapitoly 5.2.2, konkrétně jak vypadá dialogové okno pro přidání vozidla po implementaci. Stejně jako v případě s přihlášením uživatele dostává mobilní klient JSON zprávu ze serveru 6.3. Tenotkrát se nicméně jedná o data, která se týkají vozidla. Tuto zprávu



Obrázek 6.3: Obrazovka pro MainActivity.



Obrázek 6.4: Dialogové okno DashBoardActivity.

mobilní klient následně zpracovává a výsledek se zobrazuje v jednoduché podobě, jak je zobrazeno na obrázku 6.5. Hodnota v bodě "status" je *null*, což znamená, že vozidlo je v dobrém pracovním stavu, tj. nemá žádnou poruchu. Na dané obrazovce také lze vidět, jak vypadá plovoucí tlačítko a dolní menu po implementaci.

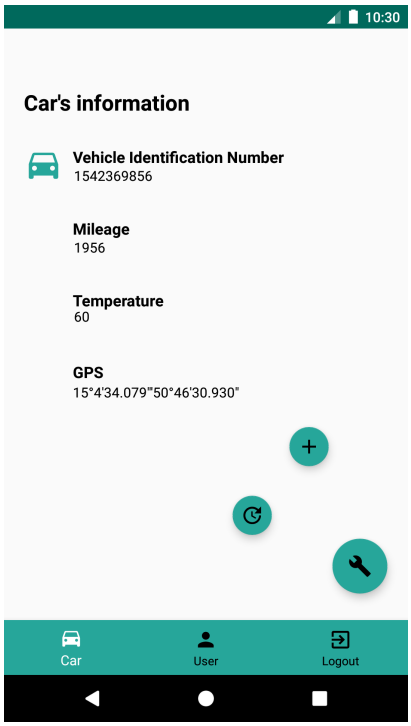
Obrázek 6.6 ukazuje, jak vypadá zpracovaná zpráva, pokud dojde k poškození vozidla: informace o autě bude zobrazena stejným způsobem, jako v předchozím případě. Bude zde ale také zobrazeno upozornění o současném stavu vozidla (hodnota v bodě "status" bude mít nějakou konkrétnost, v daném případě hodnota je *crash*).

6.2 Vyhodnocení a možná rozšíření

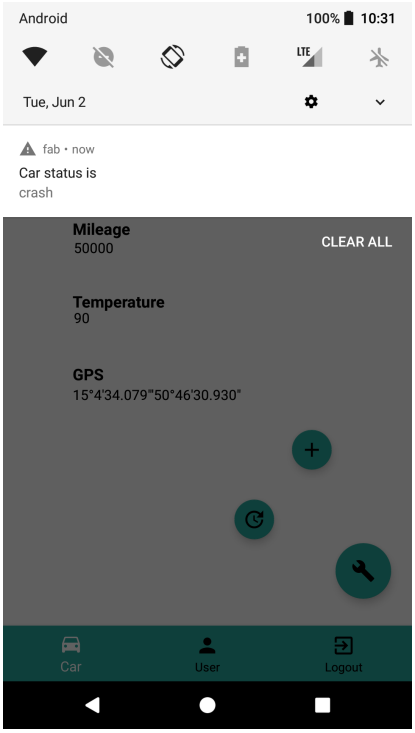
Pokračování vývoje softwaru by mohlo být zaměřeno na rozšíření stávajících funkcí, především na možnost sledování více aut z jednoho zařízení. V návrhu na implementací daného softwaru nebyla tato možnost prozkoumána. Prozatím může majitel více vozů použít funkci „přidání vozidla". Data se následně změní dle aktuálního požadavku, čehož je docíleno pomocí procesu, který byl popsán v kapitole 5.1.1.

Další novou funkci může být verifikace přidaného vozu pomocí SMS zprávy; po vyplnění identifikačního čísla vozidla by byl uživateli zaslán do SMS ověřovací kód.

V průběhu testování bylo zajištěno, že není vhodné použít relační databázový systém v implementačním systému. Dalším krokem k rozšíření možností aplikace by



Obrázek 6.5: Obrazovka pro DashBoardActivity.



Obrázek 6.6: Příklad notifikace.

tedy mohl být přechod na jiný databázový systém, který by byl lépe uzpůsobený práci s velkým objemem dat. Jednou z možností je databázový systém reálného času Firebase, který ukládá data přímo do JSON.

Zajímavou inovací by mohlo být také zobrazení GPS polohy na mapě.

7 Závěr

Cílem této diplomové práce bylo vytvořit software, který by umožňoval uživateli získávat informace o svém vozidle. Aby bylo možné takový systém realizovat, bylo nutné seznámit se nejprve s BBCU jednotkou, s pojmem automobilový Ethernet a také s požadavky na komunikaci s řídící jednotkou. Nezbytnou součástí procesu byl výběr vhodného způsobu kontroly údajů, aby měl přístup k informacím o vozidle pouze jeho majitel.

Webová služba a mobilní aplikace byly úspěšně vytvořeny a otestovány; tento postup je v souladu se stanovenými zásadami pro vypracování diplomové práce. Celkem bylo provedeno několik testů na různých zařízeních s rozlišnými verzemi operačního systému Android a také na Android emulátoru. Výsledná verze je plně funkční a je možné ji v průběhu měnit základě nových požadavků, které mohou vzniknout v průběhu vývoje řídící jednotky BBCU. Mobilní aplikace má jednoduché uživatelské rozhraní, což umožňuje její rychlé a intuitivní ovládání. Uživatel si tak aplikaci bude moci snadno s rychle osvojit.

Podařilo se odhalit a eliminovat nedostatky implementovaného softwaru a navrhnout jejich řešení či funkční rozšíření.

Vzhledem k tomu, že BBCU jednotka se stále vyvíjí, lze předpokládat, že software bude možné rozšířit nejen o data z dalších senzorů, ale také o jiné funkce, které pomohou uživateli lépe chápat procesy odehrávající se v jeho vozidle.

Při přechodu na MQTT lze očekávat snížení spotřeby energie baterie a zvýšení počtů zpráv přenášených za jednotku času, a to všechno díky tomu, že má protokol malou režii na straně zařízení. Kromě velmi malého zatížení systému nabízí MQTT vysokou efektivitu komunikace i v sítích s malou šířkou pásma. Současně je ve struktuře dat přenášených pomocí MQTT velmi málo informací o službě, to znamená, že protokol nezatěžuje síť. Také propustnost MQTT je několikrát (v 3G síti) vyšší než propustnost REST přes HTTP protokol. Kromě toho má MQTT protokol několik možností QoS, které lze použít k zajištění doručení. Také publisher může publikovat svá data bez ohledu na stav subscriberu, což je velmi dobré v podmínkách nestabilního připojení.

Bibliografie

[1] *343/2014 Sb. - Beck-online*. URL: https://www.beck-online.cz/bo/chapterview-document.seam?documentId=onrf6mrqge2f6mzugmwti (cit. 16. 05. 2021).

[2] *A Brief Introduction to REST*. InfoQ. URL: https://www.infoq.com/articles/rest-introduction/ (cit. 03. 06. 2020).

[3] *An Inside Look At The Automotive Ethernet Protocol*. Electrical Engineering News and Products. 6. srp. 2018. URL: https://www.eeworldonline.com/an-inside-look-at-the-automotive-ethernet-protocol/ (cit. 03. 06. 2020).

[4] *Android program to implement Fragment*. CODEDOST. URL: https://codedost.com/get-started-android/android-programs/android-program-implement-fragment/ (cit. 03. 06. 2020).

[5] *Android version history*. In: *Wikipedia*. Page Version ID: 960341921. 2. červ. 2020. URL: https://en.wikipedia.org/w/index.php?title=Android_version_history&oldid=960341921 (cit. 03. 06. 2020).

[6] *Android Volley vs Retrofit | Better Approach?* Codeplayon. 10. dub. 2019. URL: http://www.codeplayon.com/2019/04/android-volley-vs-retrofit-better-approach/ (cit. 03. 06. 2020).

[7] *App Manifest Overview*. Android Developers. URL: https://developer.android.com/guide/topics/manifest/manifest-intro (cit. 03. 06. 2020).

[8] *Automotive Ethernet: The Future of In-Car Networking?* Electronic Design. Dub. 2018. URL: https://www.electronicdesign.com/markets/automotive/article/21806349/automotive-ethernet-the-future-of-incar-networking (cit. 03. 06. 2020).

[9] *AWSIoTPythonSDK — AWSIoTPythonSDK 1.4.0 documentation*. URL: https://s3.amazonaws.com/aws-iot-device-sdk-python-docs/sphinx/html/index.html (cit. 16. 05. 2021).

[10] Sean Bradley. *Add Swagger UI to your Python Flask API*. Medium. 29. ún. 2020. URL: https://medium.com/@sean_bradley/add-swagger-ui-to-your-python-flask-api-683bfbb32b36 (cit. 03. 06. 2020).

[11] *BroadR-Reach*. In: *Wikipedia*. Page Version ID: 922723414. 23. říj. 2019. URL: https://en.wikipedia.org/w/index.php?title=BroadR-Reach&oldid=922723414 (cit. 03. 06. 2020).

[12] *Build an MQTT Intercom with Wio Terminal (with Code!)* Latest open tech from seeed studio. 19. ún. 2021. URL: /blog/2021/02/19/build-an-mqtt-intercom-with-wio-terminal-with-code/ (cit. 16. 05. 2021).

[13] Deepam Goel. *Understanding Android Architecture*. Medium. 13. květ. 2018. URL: https://medium.com/@deepamgoel/understanding-android-architecture-1f0fb4b52f90 (cit. 16. 05. 2021).

[14] *Google Will No Longer Show The Android Distribution Chart On The Web*. URL: https://www.digitalinformationworld.com/2020/04/google-will-no-longer-show-the-android-distribution-chart-on-the-web.html (cit. 16. 05. 2021).

[15] Jiri Havlik et al. "BBCU in Smart Cities applications". In: *2016 Smart Cities Symposium Prague (SCSP)*. 2016 Smart Cities Symposium Prague (SCSP). Květ. 2016, s. 1–4. DOI: 10.1109/SCSP.2016.7501041.

[16] Mary Brickenstein Hofschen. *Meet MQTT 5: The most extensive and feature-rich update to the MQTT protocol ever*. URL: https://www.hivemq.com/upgrade-to-mqtt5-now (cit. 16. 05. 2021).

[17] *Is your service RESTful? Everything you need / must know about web services and REST*. URL: https://habr.com/ru/post/319984/ (cit. 03. 06. 2020).

[18] *List of HTTP status codes | Public APIs*. URL: https://public-apis.io/learn/http-status-codes (cit. 16. 05. 2021).

[19] Droid By Me. *Activity Life cycle of Android*. Medium. 29. pros. 2017. URL: https://medium.com/@droidbyme/activity-life-cycle-of-android-2e298809df6a (cit. 03. 06. 2020).

[20] *MQTT Specification*. URL: https://mqtt.org/mqtt-specification/ (cit. 16. 05. 2021).

[21] *MQTT V3.1 Protocol Specification*. URL: https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html (cit. 16. 05. 2021).

[22] *MQTT Version 3.1.1*. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html (cit. 16. 05. 2021).

[23] *MQTT Version 5.0*. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html (cit. 16. 05. 2021).

[24] *OASIS Open: Committees*. URL: https://www.oasis-open.org/committees/document.php?document_id=66091&wg_abbrev=mqtt (cit. 16. 05. 2021).

[25] *Representational State Transfer*. In: *Wikipedie*. Page Version ID: 17357691. 14. červ. 2019. URL: https://cs.wikipedia.org/w/index.php?title=Representational_State_Transfer&oldid=17357691 (cit. 03. 06. 2020).

[26] *REST API — What is HATEOAS?* URL: https://habr.com/ru/post/483328/ (cit. 03. 06. 2020).

[27] *REST API — HATEOAS?* URL: https://www.pvsm.ru/java/342751 (cit. 16. 05. 2021).

[28] *Retrofit*. URL: https://square.github.io/retrofit/ (cit. 03. 06. 2020).

[29] Viktoriia Shuvalova. *Co je MQTT a k čemu slouží ve IIoT? Popis protokolu MQTT*. iPC2U s.r.o. 14. pros. 2020. URL: https://ipc2u.tech/blogs/news/mqtt-protokol (cit. 16. 05. 2021).

[30] *What's the Difference Between BroadR-Reach and 100Base-T1?* Electronic Design. Květ. 2018. URL: https://www.electronicdesign.com/markets/automotive/article/21806576/whats-the-difference-between-broadrreach-and-100baset1 (cit. 03. 06. 2020).