VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Systém pro evidenci drobných součástek

System for Invetory of Small-sized Parts

2016 Martin Mareček

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Martin Mareček**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Systém pro evidenci drobných součástek
System for Invetory of Small-sized Parts

Jazyk vypracování: čeština

Zásady pro vypracování:

Smyslem práce je návrh a implementace modulárního informačního systému pro vedení skladu elektronických součástek. Evidovány budou jednotlivé nákupy součástek a samotné součástky s vhodnými parametry. Předpokládá se integrace s alespoň 1 návrhovým systémem CAD (např. Eagle), který umí exportovat seznam součástek použitých v projektu. Po načtení vygenerovaných dat dojde k ověření stavu na skladu a zobrazení chybějících součástek. Požadováno je využití mobilních zařízení pro načítání součástek pomocí čárových kódů.

1. Seznamte se s existujícími řešeními obdobných informačních systémů. Prostudujte možnosti exportu součástek ze zvolených CAD systémů.
2. Navrhněte vhodné řešení systému. Při návrhu dbejte na snadnou rozšiřitelnost a modularitu celého řešení.
3. Zvolte vhodný jazyk a nástroje pro implementaci s ohledem na multiplatformnost celého řešení. Navržené řešené implementujte.
4. Proveďte testování výsledného řešení v reálném provozu a vytvořte dokumentaci s ohledem na budoucí rozšiřování systému.

Seznam doporučené odborné literatury:

[1]PECINOVSKÝ, Rudolf. Návrhové vzory: [33 vzorových postupů pro objektové programování]. Vyd. 1. Brno: Computer Press, 2007, 527 s. ISBN 978-80-251-1582-4.
[2]FOWLER Martin, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee a Randy Stafford. Patterns of Enterprise Application Architecture. Vyd. 1. Addison-Wesley Professional, 2002, 560 s. ISBN 978-03-211-2742-6.
[3]MEIER, Reto. Professional Android 4 application development. Indianapolis: Wiley, c2012, xlii, 817 s. ISBN 978-1-118-10227-5.
[4]SHELDON, Robert a Ludvík ROUBÍČEK. SQL: začínáme programovat. 1. vyd. Praha: Grada Publishing, 2005, 499 s. ISBN 80-247-0999-6.
[5]HAWLITZEK, Florian a Jiří BRÁZA. Java 2: příručka programátora. 1. vyd. Praha: Grada Publishing, 2002, 315 s. ISBN 80-247-9060-2.
[6]URMAN, Scott, David KRCH, Jiří FADRNÝ, Ron HARDMAN a Michael MCLAUGHLIN. Oracle: programování v PL/SQL. Vyd. 1. Brno: Computer Press, 2007, 720 s. ISBN 978-80-251-1870-2.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Daniel Stříbný**

Datum zadání: 01.09.2015

Datum odevzdání: 29.04.2016

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 27. dubna 2016 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 27. dubna 2016 .....

Rád bych na tomto místě poděkovala všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Rád bych touto cestou poděkoval panu Ing. Danielu Stříbnému za konzultace a hodnocení postupu vykonání této práce

Abstrakt

Cílem této práce bylo vyvinout jednoduchý a univerzální informační systém pro evidenci skladu, avšak na úrovni malého domácího skladu pro domácí bastlíře a elekrotechniky, aby dotyčná osoba měla přehled o svých produktech, které má k dispozici k okamžitému použití, bez nutnosti uchovávání informací v papírové podobě. Výsledkem práce je malý, multiplatformní, uživatelsky přívětivý a rychle i zároveň jednoduše rozšířitelný systém evidence, který je propojen i pomocí interaktivních prvků dnešní moderní doby. Hlavním prvkem je modularita, čímž je zajištěna možnost rozšíření do budoucna.

Klíčová slova: evidence, informační systém, součástky, multiplatformní, sklad, elektrotechnika

Abstract

The aim of this study was to develop a simple and universal information system to record store, but at a small home store for home improvement and electrotechnics that the person should have an overview of their products, which is available for immediate use, without storing the information in paper form . The result is a small, multi-platform, user-friendly and fast and at the same time easily expandable filing system which is interconnected and interactive elements using today's modern age. The main feature is modularity, thus ensuring the possibility of future expansion.

Key Words: accounting, information system components, multiplatform, warehouse, electrotechnics

Obsah

Seznam použitých zkratek a symbolů **9**

Seznam obrázků **10**

Seznam tabulek **11**

Seznam výpisů zdrojového kódu **12**

1 Úvod **13**

2 Analýza problému **14**

2.1 Použité platformy a jazyky 14

2.2 Architektura 19

2.3 Specifikace elektronické součástky 35

3 Řešení požadavků **37**

3.1 Možnost skenování součástek 37

3.2 Integrace s návrhovým systémem 38

3.3 Komunikace s externími systémy 42

3.4 Komunikace s mobilní platformou 45

3.5 Auditování v systému 51

3.6 Třídní diagram 52

3.7 Zajištění modularity 52

3.8 Obsah přiloženého CD 52

4 Závěr **55**

Literatura **56**

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
ASM	–	Assembly language
CRUD	–	Create, Read, Update, Delete
CSS	–	Cascading Style Sheets
CSV	–	Comma-separated values
DLL	–	Dynamic-link library
DTL	–	Djnago template language
EEPROM	–	Electrically Erasable Programmable Read-Only Memory
HTML	–	HyperText Markup Language
IP	–	Internet Protocol
JSON	–	JavaScript Object Notation
MVC	–	Model-View-Controller
OpenGL	–	Open Graphics Library
SQLite	–	SQL lite => Structured Query Language lite
SSL	–	Secure Sockets Layer
URL	–	Uniform Resource Locator
XML	–	Extensible Markup Language

Seznam obrázků

1 MVC architektura 15
2 MTV architektura 15
3 Grafické znázornění architektury systému Android 18
4 Přehled vývoje oblíbenosti jazyků 19
5 Ukázka využívání metadat u modelů v Django 22
6 Ukázka layoutu v systému Android 32
7 Ukázka dialogů v systému Android 32
8 Ukázka fragmentu v systému Android 33
9 Kompletní přehled architektury 34
10 Náhled na dokumentaci součástky 36
11 Kontrola dostupnosti součástek na skladě 40
12 Export z programu EAGLE 41
13 Označení pro návrhový software 42
14 Cenové napojení na API obchodů 44
15 Notifikace v systému 51
16 Část architektury Django 53
17 Část datové vrstvy architektury Android 54

Seznam tabulek

2 Podíl operačních systémů smartphonů v druhém čtvrtletí v období roků 2012-2015 18
3 Prvních 11 top jazyků za rok 2015-2016 19
4 Ukázka CSV exportu z programu EAGLE 41

Seznam výpisů zdrojového kódu

1 Ukázka tepmlate souboru v DTL . . . 14
2 Ukázka zkráceného zápisu v pythonu . . . 16
3 Ukázka stejného kódu v C# . . . 17
4 Ukázka zápisu modelu v Django . . . 20
5 Ukázka zápisu templatu v Django . . . 23
6 Formátování výstupu v Django layoutu . . . 24
7 Ukázka souboru urls.py . . . 24
8 Definování view . . . 25
9 Dekorátor pro view . . . 26
10 Ukázka výjimky . . . 27
11 Definice třídy BaseEntity . . . 28
12 Definice interfacu IGateway . . . 28
13 Interface IUserGateway se speciálními metodami pro UserGateway . . . 29
14 Abstraktní třída BaseMapper . . . 29
15 Výňatek třídy ElectricalComponentMapper . . . 30
16 Generování čárového kódu na platformě Django . . . 37
17 Příprava snímače čárového kódu na platformě Android . . . 37
18 Mapper funkce pro výběr importovacího dekodéru . . . 38
19 Výňatek validace parametrů z importovaného souboru pro software EAGLE . . . 39
20 Ukázka komunikace s API rozraním TME . . . 43
21 Výňatek definic url adres a kotrolérů pro API rozhraní . . . 45
22 Funkce genToken pro generování Tokenu pro přístup k API rozhraní . . . 46
23 Dekorátor REST_isValidRequest . . . 46
24 Kontrolér API rozhraní s použitým dekorátorem . . . 49
25 Serializér pro entitu User . . . 50

1 Úvod

Tématem bakalářské práce je systém pro evidenci drobných součástek. V dnešní době chce každý člověk nebo firma mít přehled o stavu svých záležitostí a problémů. Proto je nutné tyto informace uchovávat. Pro uchování svých osobních informací, neboli dat, nenastává problém, protože informace si můžete uchovat dle vlastní režie. Pokud půjde o více informací, už je vhodné tyto informace někde uchovávat na jednom místě pro možnost opětovného dohledání. Toto je běžná věc, kterou provádíme denně a ani netušíme, že ji děláme. Buď si tyto informace pamatujeme nebo si píšeme poznámky. Je to pro nás běžný automatizovaný proces. Problém nastává, když tyto informace musíme s někým sdílet a zároveň je i uchovávat pro pozdější zpracovávání. V omezeném počtu osob se ještě tento problém řešit dá klasickým způsobem, ale při větším působení osob nastává problém, kdy každý může mít odlišné informace a poté dochází ke ztrátě původní informace. Proto je tyto sdílené, a nejen ty, stačí, aby byly pouze velice obsáhlé, někde uchovávat neboli evidovat.

Evidence informací, v tomto případě drobných součástek, je funkčnost číslo jedna, co se týče informačního systému. Každý informační systém je založen na procesech pro evidenci, zpracování a šíření informací neboli dat v digitální podobě, které se následně mohou využít k plánování, rozhodování a řízení v oblasti neboli doméně, kde bude informační systém nasazen a provozován. Každý informační systém je jedinečný svojí funkčností a požadavky na funkčnost. Existují i již zpracované a hotové univerzální informační systémy, kde však obecná univerzálnost nemusí být dostačující. Tyto systémy nabízí i možnost rozšiřitelnosti dle požadavků zákazníka a domény, ale i přes tyto vymoženosti nemusí být univerzální informační systémy dostačující a nebo jsou příliš drahé pro pořízení, provoz a údržbu. V této práci se zabývám verzí informačního systému pro tzv. "domácí sklad", takže pořizovací cena, provoz a údržba by měli být co nejlevnější a nejjednodušší pro domácí podmínky.

Pod pojmem "drobných součástek"si můžeme představit veškeré elektrotechnické součástky, které nás napadnou. Od obyčejných plastových podložek pod výkonové tranzistory až po komplexní mikroprocesory. Každá taková součástka je specifická svými vlastnostmi či parametry, které mohou mít velice obecný charakter (rezistivita, kapacita, ...) nebo i složitější jako např. kapacita EEPROM atp. Každý z těchto parametrů je třeba evidovat pro dostatečné uchování a následné dohledání informací o dané elektronické součástce. Čím podrobnější informace v systému máme uvedeny, tím větší šanci např. pro vyhledání náhrady pro součástku, kterou nemáme a potřebujeme hledat co nejpodobnější alternativu.

2 Analýza problému

Cílem této práce je tedy vytvoření informačního systému pro evidenci položek v domácích podmínkách v oblasti elektrotechniky. Celý systém je založen na použití dvou platforem, které se od sebe liší, ale zároveň musí mezi sebou komunikovat pro výměnu informací. Každá platforma je specifikována svým programovacím jazykem, které se od sebe navzájem liší a proto nastaly při návrhu aplikací odlišnosti.

2.1 Použité platformy a jazyky

V zadání bakalářské práce je požadavek "na multiplatformnost celého řešení"a následné implementace. Dalším požadavkem je nutnost použití mobilního zařízení pro načítání součástek pomocí čárových kódů. Pro tyto dva požadavky byly zvoleny dvě různé platformy a každá je specifická svým jazykem.

2.1.1 Django: The Web framework

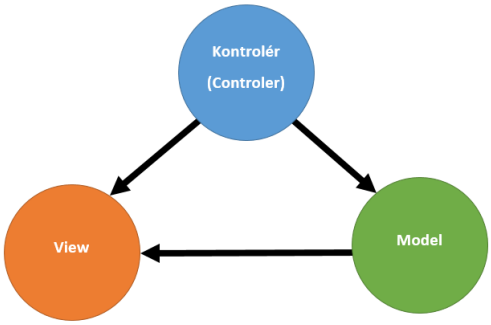
Pro požadavek na multiplatformnost jsem zvolil zpracování formou webového přístupu. Webové rozhraní je univerzální, dostupné na většině zařízení a je nezávislé na použité platformě (Windows [Microsoft], Distribuce linux [Linux], Mac [Apple], mobilní zařízení atp.). Pro vytvoření webového rozhraní jsem zvolil webový framework Django, který je založen na softwarové architektuře MTV (Model, Template, View), která vychází z MVC, tedy rozděluje celou aplikaci do tří hlavních bloků. Veškeré modely, starající se o data jsou definovány v souboru *models.py* u aplikace. Podobně je to s views, ty jsou uloženy v souboru *views.py*. Poslední jsou tedy templates, které jsou svázány s views. Templates se naleznou ve složce *templates*. Složka *templates* obsahuje soubory typu *název_templatu.html*, jejichž formát vychází z DTL, který je založen na template enginu Jinja[Jinja]. Jde o HTML kód rozšířený o značky, na jejichž místa jsou následně umístěny data zaslané z view, které byly získány z modelu. Programovacím jazykem pro Django framework je interpretovaný skriptovací jazyk Python [Python].

```
{% extends "layout.html" %}
{% block body %}
  <ul>
  {% for user in users %}
    <li><a href="{{ user.url }}">{{ user.username }}</a></li>
  {% endfor %}
  </ul>
{% endblock %}
```

Výpis 1: Ukázka tepmlate souboru v DTL

2.1.1.1 MVC

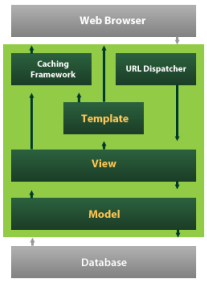
Architektura MVC je v dnešní době nejčastěji spojována právě s webovými aplikacemi, protože je jednoduchá na pochopení, implementaci, údržbu a je vhodná pro okamžité a rychlé změny v kódu. MVC se skládá ze tří bloků. **M**odel, který se stará o uchovávání dat a poskytování dat. **V**iew se stará o zobrazení dat uživateli a dává uživateli možnost zadávat příkazy na tyto data. **C**ontroler tyto příkazy akceptuje a provádí zpracování těchto příkazů a dat, které dostane od uživatele z vrstvy View. Nevýhodou MVC je problém s blokem Model. Je sice řečeno, že uchovává a zpracovává data, ale nikde není řečeno, jakým způsobem. Problém může nastat při použití více datových zdrojů současně. Pro tyto případy se musí vytvořit před datovou vrstvou další mezivrstva, která se bude starat o výběr dat z daného zdroje.



Obrázek 1: MVC architektura

2.1.1.2 MTV

MTV vychází z architektury MVC [str. 15]. Z původní architektury MVC byl kontrolér nahrazen tzv. templatem. **M**odel je stejný jako u MVC, náleží datové vrstvě a stará se o data, tzn. validace, přístup, vazby mezi daty atp. **T**emplate je na prezenční vrstvě a určuje, jak bude daný obsah zobrazen na webové stránce, např. jaká data a kde budou umístěna. Poslední vrstvou je aplikační vrstva, na které se nachází **V**iew, který určuje vztahy mezi modelem a templatem. Odděluje logický přístup k modelu a zobrazení dat v templatu. Zároveň se stará o zpracování dat od uživatele.



Obrázek 2: MTV architektura [MTV architektura]

2.1.1.3 Skriptovací jazyk Python

Python je skriptovací programovací jazyk. Jde o interpretovaný jazyk. Hlavním rysem každého interpretovaného programovacího jazyka je provádění kódu za běhu aplikace. Tudíž před spuštěním neprobíhá žádná kompilace do byte kódu případně přímo na instrukce dané instrukční sady daného procesoru. Kompilované jazyky mají výhodu (pokud se nebavíme o jazycích jako je C# nebo Java, kde se předkompilovává kód a následně se provádí tzv. byte kód), že dopředu víme, jak se bude daný program vykonávat a které instrukce bude procesor vykonávat. Kdežto u interpretovaných jazyků, jako je Python, potřebujeme na spuštění programu ještě tzv. interpret. Interpret je tedy základní jednotka, která provádí napsaný kód a musí být nainstalován na daném systému, aby mohl být program spuštěn. Tohle je nevýhoda oproti čistým kompilovaným jazykům, kde nemusíme na provedení programu mít speciální "nadprogram"(interpret). Potom může nastat situace, že budeme chtít spustit nějaký interpretovaný kód, ale na stroji, kde ho budeme chtít spustit, nebude interpret. Program nespustíme. I když je Python ve většině případů výchozí součástí každé distribuce Linuxu a Mac, i na Windows se dá interpret Pythonu přidat, tak může tato situace nastat. Jazyk Python je netypový, jako většina skriptovacích jazyků (např. JavaScript ve webových prohlížečích). To znamená, že až během provádění programu se za běhu aplikace určí, jakého datového typu daná proměnná, výsledek funkce nebo výsledek matematické operace bude. Toto může přinést někdy i velké problémy, pokud nejsou ošetřeny např. chyby vstupu od uživatele. Typickým příkladem je sčítání řetězce a čísla, kde následující příkaz očekává jako vstup číslo, ale dostane řetězcovou hodnotu. Tím může nastat vyvolání výjimky (exception) a po neošetření chyby neočekávané ukončení provádění programu. Mezi nevýhodu jazyka Python bych zařadil věc, se kterou jsem měl problém při zpracování této práce. Tou nevýhodou je, že Python nemá podporu rozhraní (interface). Interface je předpis který říká, jak se musí daný objekt chovat, pokud tento interface implementuje. Interface byl zaveden hlavně z důvodů, aby se zabránilo vícenásobné dědičnosti objektů (Jeden objekt dědí(přejímá vlastnosti a chování) z více předků, může nastat kolize vlastností a chování). Python pro svůj běh žádný interface nepotřebuje. Jelikož jde o skriptovací jazyk, tak se předem neví, jaká akce se bude volat a s jakými parametry. Proto se může při psaní kódu volat jakákoliv metoda nezávisle na tom, zda existuje či ne. Pokud se při vykonávání programu daná metoda nalezne, zavolá se a vše je v pořádku. Pokud metoda neexistuje, aplikace navodí výjimku. Proto skriptovací jazyky rozhraní nepotřebují. Přesto je dobré si předpis interfaců někde uchovávat, aby bylo zajištěno implementování všech metod. Python je hlavně významný svými jednoduchými a krátkými zápisy, které pro člověka, který vidí Python poprvé mohou hodně zmást a ani jim nemusí porozumět.

```
1 // ukazka ziskani sudych cisel
2 print([x for x in [0,1,2,3,4,5,6,7,8,9] if x % 2 == 0]) //[0, 2, 4, 6, 8]
```

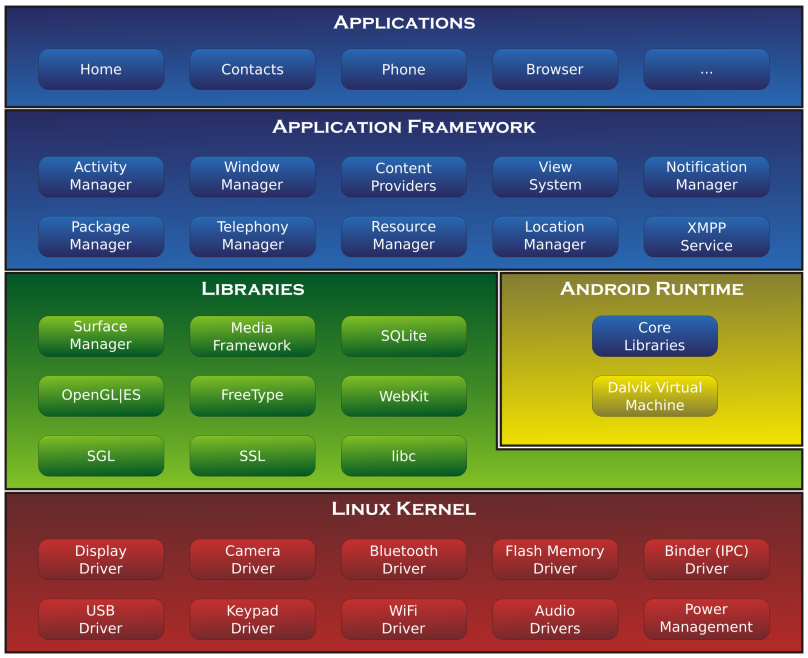
Výpis 2: Ukázka zkráceného zápisu v pythonu

```
1  // ukazka ziskani sudych cisel
2  int[] data = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
3  List<int> res = new List<int>();
4  foreach (int x in data) {
5        if (x % 2 == 0)
6              res.Add(x);
7  }
8  foreach(int x in res){
9        Console.WriteLine(x);
10 }
```

Výpis 3: Ukázka stejného kódu v C#

2.1.2 Android: mobilní platforma

Pro mobilní platformu jsem zvolil nejrozšířenější a cenově nejdostupnější systém, operační systém Android. Android je nabízen pod open source licencí, to znamená, že zdrojové kódy jsou otevřeny, všem dostupné a každý si může provést libovolné úpravy dle vlastního uvážení. Vývoj operačního systému má na starost uskupení Open Handset Alliance[Open Handset Alliance], do které se zahrnují přední výrobci mobilních zařízení a hardwaru pro mobilní zařízení, a společnost Google[Google], která dnešní Android vlastní. 5. listopadu 2007 byla oznámena první verze operačního systému Android a dnes je to nejpoužívanější systém na mobilních platformách, viz tabulka 2.1.2 na straně 18. Architektura systému je rozdělena do 5-ti vrstev. Nejnižší vrstva j jádro operačního systému, které se stará o spojení mezi hardwarem a softwarem. Toto jádro je založeno na linuxovém jádře. Další vrstva je vrstva s knihovnami napsané v jazyce C/C++ jako např. SQLite, OpenGL, SSL a knihovny pro multimédia. Třetí vrstvou je Android Runtime, která se stará o správu paměti a organizace práce s vlákny. Dále také zajišťuje základní knihovny pro jazyk Java. Knihovny z pro uživatelské rozhraní z Javy byly nahrazeny knihovnami pro Android rozhraní a byla přidána knihovna Apache pro komunikaci se sítí. Application framework je nejdůležitější částí pro vývojáře. Tato vrstva jim umožňuje využívat veškeré součásti systému na běžné úrovni, jako jsou služby, hardware, aplikace běžící na pozadí, prvky uživatelského rozhraní (notifikace) atp. Poslední, nejvyšší vrstva je aplikační vrstva, kde už, podle názvu, jsou samotné aplikace pro uživatele jako např. aplikace pro příjem SMS. Nejčastěji používaným jazykem pro tvorbu aplikací pro platformu Android je jazyk Java. Tento fakt se v nejbližší době asi změní. Microsoft [Microsoft] dne 31.3 2016 oznámil uvolnění svého vývojového prostředí Xamarin [Xamarin] zdarma pro všechny, kde jedním jazykem, konkrétně C#, můžete psát jeden kód a následně jej přeložit pro 3 nejpoužívanější mobilní platformy (Android, Windows Phone, iOS), což povede, dle mého názoru, k velké oblibě u začínajících vývojářů.



Obrázek 3: Grafické znázornění architektury systému Android [Architektura systému Android]

Operační systém	Q2 2012	Q2 2013	Q2 2014	Q2 2015
Android	69,9%	79,8%	84,8%	82,8%
iOS	16,6%	12,9%	11,6%	13,9%
Windows Phone	3,1%	3,4%	2,5%	2,6%
BlackBerry OS	4,9%	2,8%	0,5%	0,3%
Ostatní	6,1%	1,2%	0,7%	0,4%

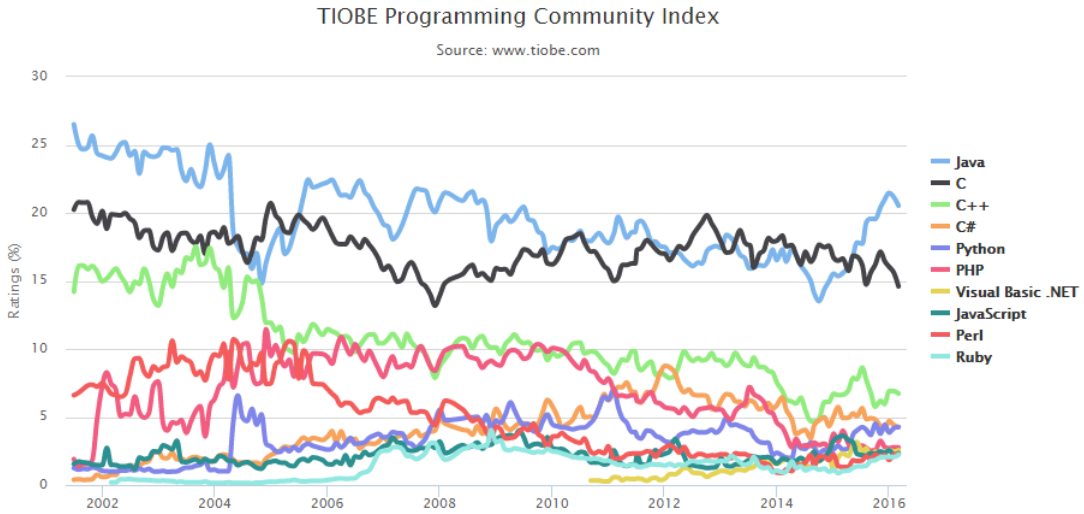
Tabulka 2: Podíl operačních systémů smartphonů v druhém čtvrtletí v období roků 2012-2015 [Hodnocení vývoje mobilních platforem]

2.1.2.1 Programovací jazyk Java

Java byla v historii mezi vývojáři velice oblíbená, protože se dá velmi jednoduše přenášet mezi platformami (Windows, Linux, Web). Dnes je dle žebříčků Java na druhém místě. Java není čistě kompilovaným jazykem. Zdrojové kódy se předkompilovávají do byte kódu a následně se tento předkompilovaný kód provádí při spuštění aplikace. Předkompilovaný kód je důležitý právě pro multiplatformnost. Při vykonávání se rozhodne, jak se předkompilovaný příkaz použije v závislosti na platformu (Windows, Linux), architekturu procesor (x86, x64), instrukční sadu procesoru atp. Tím pádem je Java nezávislá na platformě, z toho ale vyplývají problémy, které souvisí např. s uživatelským rozhraním. Java nemůže využívat specifické prvky platformy, pouze základní prvky. Proto si Java veškeré uživatelské rozhraní definuje a vytváří sama.

Programovací jazyk	Pořadí Březen 2016	Pořadí Březen 2015	Podíl	Změna
Java	1	2	20.528%	+4.95%
C	2	1	14.600%	-2.04%
C++	3	4	6.721%	+0.09%
C#	4	5	4.271%	-0.65%
Python	5	8	4.257%	+1.64%
PHP	6	6	2.768%	-1.23%
Visual Basic .NET	7	9	2.561%	+0.24%
JavaScript	8	7	2.333%	-1.30%
Perl	9	12	2.251%	+0.92%
Ruby	10	18	2.238%	+1.21%
Delphi/Object Pascal	11	13	2.005%	+0.85%

Tabulka 3: Prvních 11 top jazyků za rok 2015-2016 [Top programovací jazyky]



Obrázek 4: Přehled vývoje oblíbenosti jazyků [Top programovací jazyky]

2.2 Architektura

Architektura informačních systémů definuje řešení informačního systému. Určuje směr vývoje. Jelikož architektura je hlavním prvkem, musí být dostupná všem, kteří na systému spolupracují, tudíž musí být velice jednoduchá a srozumitelná pro všechny. Tato práce je primárně založena na webové platformě Django, která tvoří jádro systému, bez kterého systém nemůže existovat, ta má svoji architekturu MTV[2.1.1.2], popsanou na straně 15, ale díky požadavku na mobilní platformy, je na mobilní platformě pro systém Android architektura odlišná. Proto se v této práci budu zabývat oběma architekturami, jak pro web, tak pro mobilní zařízení.

2.2.1 Architektura Django: MTV

Architektura MTV [2.1.1.2] je podrobně popsána na straně 15, takže se zde budu zabývat pouze specifikami, které souvisí přímo s vypracováním systému.

2.2.1.1 Model

Veškeré modely se nacházejí na datové vrstvě. Tyto modely jsou uloženy v souboru *models.py*. Každý model je třída, která dědí od předka *models.Model*. Model obsahuje proměnné a metody, které definují jeho vlastnosti a chování. Každý model by měl být jednoznačně identifikovatelný. Pro tuto vlastnost se nejčastěji využívá proměnné id, která obsahuje jedinečné číslo pro skupinu objektů stejného typu.

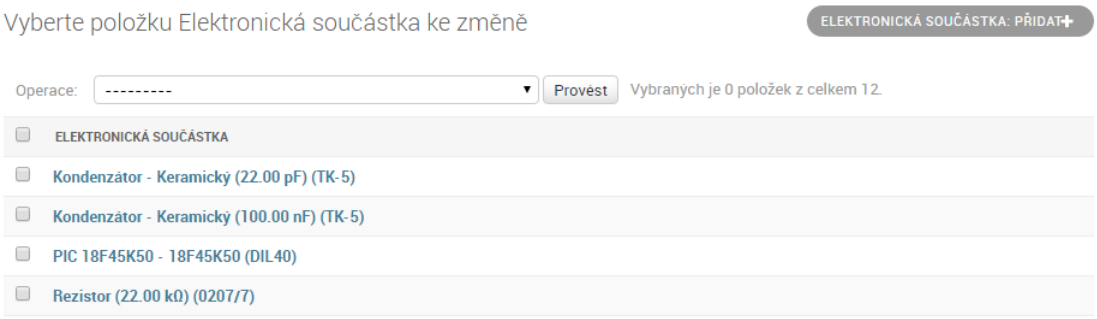
```
1  class ElectricalComponent(models.Model):
2      def datasheet_directory_path(instance, filename):
3          return 'datasheets/datasheet_{0}/{1}'.format(instance.id, filename)
4
5      def preview_directory_path(instance, filename):
6          return 'previews/preview_{0}/{1}'.format(instance.id, filename)
7
8      def schematic_directory_path(instance, filename):
9          return 'schematics/schematic_{0}/{1}'.format(instance.id, filename)
10
11
12     id = models.AutoField(primary_key=True, unique=True)
13     name = models.CharField('Nazev', max_length=100)
14     description = models.CharField('Popis', max_length=100, blank=True, null=
           True)
15     designation = models.CharField('Oznaceni (BC337,...)', max_length=100,
           blank=True)
16     pinCount = models.IntegerField('Pocet pinu', blank=False)
17     mainValue = models.ForeignKey(EC_MainValue, verbose_name='Hlavni hodnota',
           default=None, blank=True, null=True)
18     package = models.ForeignKey(EC_Package, verbose_name='Packgage', default=
           None)
19     componentParams = models.ManyToManyField(EC_Param, verbose_name='Paramtery'
           , blank=True)
20     deviceDesignations = models.ManyToManyField(DesignSoftwareDesignations,
           verbose_name='Oznaceni ve schematu', blank=False)
```

```
21      datasheet = models.FileField('Dokumentace', upload_to=
            datasheet_directory_path, blank=True)
22      preview = models.FileField('Nahled', upload_to=preview_directory_path,
            blank=True)
23      schematic = models.FileField('Schema', upload_to=schematic_directory_path,
            blank=True)
24
25
26      def __unicode__(self):
27          resstr = self.name
28          if not self.description == "":
29              resstr += " - %s" % self.description
30          if self.designation:
31              resstr += " - %s" % self.designation
32          if self.mainValue:
33              resstr += " (%s)" % self.mainValue
34          if self.package:
35              resstr += " (%s)" % self.package
36
37          return resstr
38
39      def __str__(self):
40          return self.__unicode__()
41
42      class Meta:
43          ordering = ['name']
44          verbose_name = 'Elektronicka soucastka'
45          verbose_name_plural = 'Elektronicke soucastky'
```

Výpis 4: Ukázka zápisu modelu v Django

Nyní se zaměřím na podrobný popis jednotlivých řádků výše uvedeného kódu. Na řádku č.1 je definován název modelu a v závorce je uvedena třída, ze které dědí. Na řádcích 2 až 9 jsou definovány 3 metody, pomocí kterých se určují cesty pro ukládání multimediálních souborů. Pro každé jedinečné id modelu se vytvoří zvláštní adresář, do kterého se zaznamenávají veškerá multimédia, včetně historických změn. To znamená, že veškeré změny zůstávají uloženy na serveru a jsou zpětně vyhledatelné. Tyto metody se používají jako parametry na řádcích 21, 22 a 23 ke kterým se ještě dostanu. Následují řádky 12 až 23, kde jsou definovány vlastnosti daného modelu. Každá vlastnost musí být předem daného datového typu, jako např. CharField pro řetězce (znaky), IntegerField pro celá čísla, AutoField pro automatické generování hodnoty pro nově

vytvořené modely, které následně budou uloženy do databáze, BooleanField pro hodnoty typu ano/ne, EmailField, DecimalField atp. Pro komplexnější datové typy, jako např. pro reference na jiné modely, jsou použity typy ForeignKey pro odkázání na cizí model, ManyToManyField pro odkázání na více modelů současně nebo FileField pro odkaz na uložený multimediální soubor. Dále tyto vlastnosti můžeme parametrizovat, a to parametry jako unikátnost (***unique***=*True*), maximální velikost (***max_length***=*100*), jestli může daná hodnota být prázdná (***blank***=*True*), zda může nabývat hodnoty typu null (***null***=*True*) a mnoho dalších. Mezi další možnosti modelu patří chování, tedy procedury a funkce. Na řádku 26 vidíme funkci ___*unicode*___, která je spíš více známa pod názvem *toString()* z většiny programovacích jazyků. Tato funkce zajistí převod modelu na řetězcovou hodnotu. Tato metoda nemusí být definována, poté funkce vrací hodnotu svého předka. Ten vrací jako výsledek funkce číslo, kde je objekt umístěn v paměti v hexadecimálním formátu. Funkce se nejčastěji definuje, pokud má objekt participovat nějakým způsobem na uživatelském rozhraní, hlavně tedy v administraci databáze. Na řádku 39 je funkce ___*str*___, která jen volá funkci ___*unicode*___. Tento způsob volání je nutný pro zpětnou kompatibilitu mezi verzemi Pythonu 2.7 a Python 3.x. Na řádku 42 je definována třída *Meta*. Tato třída slouží pro reprezentaci dat v databázovém rozhraní pro tzv. metadata. Informace v této třídě poskytují informace, jak se daný model bude v databázovém rozhraní zobrazovat (jednotné [verbose_name] a množné [verbose_name_plural] číslo názvu modelu) a jak bude tříděn (ordering)



Obrázek 5: Ukázka využívání metadat u modelů v Django

2.2.1.2 Template

Templaty jsou umístěny na prezentační vrstvě. Už od názvu prezentační jde odvodit, že se bude jednat o předávání informací a dat. Templaty se nachází v projektovém adresáři *templates* a jsou ve formátu HTML souborů. Tyto soubory obsahují značky, kde se budou předaná data zobrazovat. Data jsou dodávána z předchozí vrstvy (aplikační), z View. Jde pouze o předávání dat. Grafickou podobu dat zajišťuje jazyk CSS, který může být podpořen jazykem JavaScript pro příjemnější interakci s uživatelem. V této práci je pro jednoduchost, grafické zpracování a variabilnost použit **Bootstrap** framework 2.2.1.2.1

```
{% extends 'structure_index.html' %}
{% load staticfiles %}
{% block wrap %}
    {% if product %}


        <div class="panel panel-primary">
            <div class="panel-heading">
                <h4><b>{{ product.eComp }}</b></h4>
            </div>
            <div class="panel-body">
                <ul>
                    <li><b>Mnozstvi [ks]: </b>{{ product.amount }}</li>
                    <li><b>Barcode: </b>{{ product.barcode|stringformat:"012d" }}
                        </li>
                </ul>
                <br>

                <div style="text-align: center;">
                    <img src="{% url 'barcode' product.barcode %}">
                    <br>
                    <a href="{% url 'barcodePrint' product.barcode %}" target="
                        _blank"><i class="fa fa-print">Vytisknout</i></a>
                </div>

                <br><br>
                <td><a href="{% url 'component' product.eComp.id %}" target="
                    _blank"><img style="width:60px;" src="{% static "eye.png" %}"
                    > Zobrazit soucastku</a></td>
            </div>
        </div>


    {% else %}
        Produkt nenalezen!
    {% endif %}
{% endblock %}
```

Výpis 5: Ukázka zápisu templatu v Django

Veškeré značky týkající se DTL začínají nejčastěji dvěma symboly. První značka {{ *datova_polozka* }} se vztahuje k výpisu informací/dat. V této značce je možnost za datový zdroj např. přidat formátování dat. Typickým příkladem je formát data.

```
{{ datum_vytvoreni|date:"c" }}
```

Výpis 6: Formátování výstupu v Django layoutu

Další nejpoužívanější značkou je {% *výraz* %}, kde místo výrazu můžeme umístit nejrůznější bloky jako if, while, for, url, load, block, extends. Na první řádku kódu lze vidět hned využití příkazu extends, díky kterému přidáme do aktuálního template souboru jiný, většinou nadřazený, template. Díky těmto extends příkazům můžeme stránku rozložit do bloků a ty následně skládat do jednoho celku dle toto, jak aktuálně potřebujeme. Řádek 2 slouží pro načtení lokálních souborů, které potřebujeme v template zobrazit, viz. řádek 25, kde pomocí {% static "eye.png"%} předáváme odkaz na soubor pro zobrazení obrázku. Následující řádek, řádek 3, definuje blok, ve kterém bude daný obsah. Pokud tento blok existuje v předkovi (templatu) z řádku 1, tak se tento blok vloží do něj. Řádek č.4 kontroluje, zda byla předána data "product". Pokud ano, provádí se blok kódu mezi řádky 5-29 (včetně). Pokud ne, provede se kód na řádku 31. Poslední výraznou značkou je {% url ***název_url*** *parametry_url* %}. Na toto místo, kde je tato značka použita, se vygeneruje URL na základě souboru *urls.py* přiloženého u Django projektu.

```
urlpatterns = [
    url(r'^$', 'ElectricalComponents.views.index', name='index'),
    url(r'^orders/$', 'ElectricalComponents.views.orders', name='orders'),
    url(r'^order/([0-9]+)', 'ElectricalComponents.views.order', name='order'),
    url(r'^products/$', 'ElectricalComponents.views.products', name='products')
        ,
    url(r'^product/([0-9]+)$', 'ElectricalComponents.views.product', name='
        product'),
      .
      .
      .
]
```

Výpis 7: Ukázka souboru urls.py

Každý záznam url definuje formát url, do kterého patří název a případné parametry (^order/([0-9]+)). Následujícím parametrem url záznamu je View, které bude požadované url zpracovávat (ElectricalComponents.views.order). Posledním parametrem je název url, který slouží pro identifikaci v Template (name='order')

2.2.1.2.1 Bootstrap

Bootstrap [Bootstrap] je webový framework, který zahrnuje moderní prvky jazyků HTML, CSS a JavaScript pro vývoj moderních, responzivních a mobilních webů pomocí jednoho kódu. Pro použití různých prvků na webové stránce se nemusí psát vlastní kódy nebo dohledávat komponenty třetích stran. Veškeré moderní a základní prvky obsahuje Bootstrap a jak je stačí jen využít.

2.2.1.3 View

Poslední součástí je View, která se stará o zprostředkování dat pro template a případně zpracování dat z template od uživatele. Jelikož je template oddělen od dat a data jsou nezávislá na způsobu zobrazení, je tímto zajištěna modularita. To znamená, že pokud by nastala nutnost výměny nějaké vrstvy, nenastane žádný problém se závislostí na dalších vrstvách. Data se přebírají od vrstvy, kde jsou umístěny modely, tedy od datové vrstvy.

```
1 @isLogged()
2 def product(request, productid=-1):
3     productoutput = Product.objects.filter(id=productid).first()
4 
5     params = {'pageTitle': 'Produkt', 'product': productoutput}
6     return render(request, 'product.html', params)
```

Výpis 8: Definování view

Na první řádku je vidět tzv. dekorátor [2.2.1.3]. Tento dekorátor se provede ještě před samotným provádění View. Použití tohoto dekorátoru je následující: Pokud je uveden před View tento dekorátor, znamená to ověření, zda je daný uživatel, který akci na view vyvolal, přihlášen. Pokud ano, tak se standardně začne view provádět. Na řádku 2 se definuje název view a jeho parametry, které mají být předány z url. U těchto parametrů je dobré zadávat i výchozí hodnotu pro případ, kdyby nepřišel z url žádný parametr. Následující řádek, řádek 3, se vztahuje již ke komunikaci s modelem, konktrétně modelem **Product**, na kterém se dotáže, zda-li existuje produkt takový, že jeho identifikátor (id) je roven předanému parametru view. Výsledek tohoto příkazu může být samotný produkt nebo tento příkaz vrátí tzv. None neboli Null. Na 5-tém řádku se výsledky, které se mají předat na template sestaví do slovníku (**klíč**=*hodnota*). Tento slovník se na řádku 6 předá renderovací funkci spolu s názvem templatu, který se má vygenerovat.

```
1  class isLogged(object):
2      def __init__(self):
3          pass
4
5      def __call__(self, f):
6          def wrap(request, *args, **kwargs):
7              if not request.user.is_authenticated():
8                  return HttpResponseRedirect(reverse('login'))
9              else:
10                 return f(request, *args, **kwargs)
11
12         wrap.__doc__ = f.__doc__
13         wrap.__name__ = f.__name__
14         return wrap
```

Výpis 9: Dekorátor pro view

Zde je ukázka dekorátoru, který se stará o kontrolu přihlášení uživatele. Nejpodstatnější jsou řádky 7 až 10, kde se dle požadavku **request** zjistí, zda je daný uživatel autorizován. Pokud ano, tak se zavolá funkce, která následovala za dekorátorem. Pokud uživatel přihlášen nebyl, provede se přesměrování na přihlašovací stránku dle url **name**="login"

2.2.1.4 Servisní vrstva

Pro požadavek na zapracování mobilních zařízení byla přidána čtvrtá vrstva, servisní vrstva, která se stará o poskytování API rozhraní pro externí systémy. Díky API rozhraní zůstane skrytá logika pro externí systémy a data mohou být poskytována bez vyzrazení know-how zpracování

2.2.2 Architektura Android: Třívrstvá architektura

Architekturu aplikace pro systémy Android se rozdělil také na 3 části, tedy 3-vrstvou architekturu. Vrstvy jsou děleny na datovou, aplikační a prezentační vrstvu.

2.2.2.1 Datová vrstva (DataLayer)

Je nejnižší vrstva celé aplikace. Na této vrstvě probíhají veškeré komunikace s externími systémy pro získávání informací a následný převod těchto informací na formát pro aplikační vrstvu. Veškeré chyby, které mohou během komunikace nastat jsou buďto ošetřeny přímo v této vrstvě nebo jsou přeneseny do další vrstvy pro následné ošetření. Každá chyba má vytvořenou svoji specifickou výjimku pro přehlednější ošetření.

```
public class No200Exception extends Exception {
    private int code;
    private String message;

    public No200Exception(CompleatedAction action){
        this.code = action.getStatusCode();
        this.message = action.getResult();
    }

    public String getRawMessage(){
        return this.message;
    }

    @Override
    public String getMessage() {
        return String.format("Server navratil kod %d. %s", this.code, this.
            message);
    }
}
```

Výpis 10: Ukázka výjimky

Pro návrh této vrstvy byl zvolen návrhový vzor Factory [Design Pattern - Factory]. V tomto vzoru je základním prvkem jednotnost všeho. To znamená, že každý objekt musí splňovat určité vlastnosti, aby se dal použít libovolný objekt na libovolném místě, protože bude mít stejné chování jako všechny ostatní objekty. K tomuto účelu návrhový vzor využívá interfacy. Základním prvkem datové vrstvy je generický interface IGateway a následně generická abstraktní třída BaseGateway. Pro přístup k datům se používají Gateways, neboli **brány**. Pokud potřebujeme

vytvořit novou komunikační třídu, musí daná třída implementovat zmíněný interface IGateway a dědit z abstraktní třídy BaseGateway. Tímto nová třída získá veškeré atributy a metody pro získávání dat, které mohou být volány z vyšších vrstev. Interface obsahuje základní CRUD operace a jednu funkci navíc, pro možnost vyhledávání přímo pomocí atributu ID. Tento atribut obsahují veškeré entity v platformě Django, takže je možnost tuto funkcionalitu využívat. Aby tuto možnost vyhledávání bylo možno zavést i do této aplikace, musí mít každá entita, která bude pracovat s daty z externího prostředí, uvedena svůj identifikátor ID. Tohoto požadavku je docíleno pomocí třídy BaseEntity z aplikační vrstvy, která tento parametr obsahuje. Každá entita v systému z této entity musí dědit.

```
public class BaseEntity implements Parcelable {
    public Integer id;

    public BaseEntity(int p_id) {
        this.id = p_id;
    }
```

Výpis 11: Definice třídy BaseEntity

Interface IGateway je zároveň generický. To znamená, že metody, které obsahuje a daná třída je bude později případně implementovat, budou pracovat nebo vracet data, u kterých předem nevíme, jakého budou datového typu. Tyto informace se zjišťují až během kompilace. Výše je uvedeno, že entity musí obsahovat atribut ID, pokud tedy budeme implementovat interface IGateway s nesprávným generickým objektem, může nastat nefunkčnost při kompilaci případně při spuštění programu. Proto je v definici tohoto interfacu uvedeno, jakého datového typu generická třída musí být. Při kompilaci se kontroluje, zda dosazená třída je daného typu.

```
1 public interface IGateway<T extends BaseEntity>{
2     List<T> All();
3     Boolean Add(T entity);
4     Boolean Delete(T entity);
5     Boolean Update(T entity);
6     T FindById(Integer Id) throws Is500Exception, ConnectionException,
          DecodeRecordException, No200Exception;
7     List<T> FindBy(KeyValuePair prms);
8 }
```

Výpis 12: Definice interfacu IGateway

V interfacu IGateway 2.2.2.1 na řádku 1 vidíme názornou ukázku. Generická třída T musí dědit (*extends*) z třídy BaseEntity. Pokud potřebujeme specifičtější dotazy na datový zdroj pomocí metody, musí daná třída, která tuto metodu vyžaduje, mít vytvořen svůj interface pro tyto

metody. Zvláštní interface je nutný z důvody komunikace aplikační a datové vrstvy.

```
public interface IUserGateway {
    UserApiKeyInfo getApikey(String username, String pass) throws
        Is500Exception, ConnectionException, No200Exception;
    Boolean userExist(String username) throws Is500Exception,
        ConnectionException, No200Exception;
}
```

Výpis 13: Interface IUserGateway se speciálními metodami pro UserGateway

Jelikož se o zpracování výjimek nemusí starat jen datová vrstva a některé výjimky se mohou předat vyšším vrstvám pro zpracování, musí některé metody v interfacech mít uvedeny na klíčovým slovem ***throws*** ty výjimky, které neobsluhují a musí být obslouženy při volání těchto metod.

2.2.2.2 Aplikační vrstva (ApplicationLayer)

Přístup k datům nemusí být pouze k jednomu datovému zdroji, to znamená, že můžeme mít více bran. O to, který typ brány se bude vyžívat, je jedna z funkčností aplikační vrstvy. Další funkčností je správa entit v systému. Pro komunikaci s datovou vrstvou se používají Data Mappery [Design Pattern - Data Mapper]. Každý mapper dědí z generické abstraktní třídy **BaseMapper**. Třída BaseMapper poskytuje všem třídám, které z ní budou dědit, instanci třídy, která implementuje interface typu IGateway. Proto to je každý data mapper napojen na jakýkoliv datový zdroj pro daný generický typ. Generický typ, stejně jako v datové vrstvě u Gateway, opět musí dědit z entity BaseEntity, aby byla zajištěna dostupnost parametru ID (identifikátor).

```
public abstract class BaseMapper<T extends BaseEntity> {
    protected IGateway<T> gateway;

    public BaseMapper(IGateway<T> p_gateway)
    {
        this.gateway = p_gateway;
    }
}
```

Výpis 14: Abstraktní třída BaseMapper

Každý mapper má svůj interface pro definování metod. Uchování interfaců u mapperů byla chyba při návrhu architektury. Ovšem ne chyba typu nefunkčnosti, ale chyba typu nadbytečnosti (redundance). Jelikož mapper pro danou entitu bude vždy jeden a tím bude mít předem dané metody, není třeba k němu vytvářet speciální interface pro univerzálnost. Proto by interfacy

mohli být odstraněny. Základním interfacem pro mappery je **IMapper**. V něm jsou definovány všechny metody pro CRUD operace a jedna metoda navíc (*void compleateEntity(T entity)*) pro doplňování dat do dané entity, pokud se entita skládá z více entit v systému.

```
1  public class ElectricalComponentMapper extends BaseMapper<ElectricalComponent>
       implements IMapper<ElectricalComponent> {
2      private DocumentMapper documentMapper;
3
4      public ElectricalComponentMapper(IGateway<ElectricalComponent> p_gateway,
           DocumentMapper documentMapper) {
5          super(p_gateway);
6          this.documentMapper = documentMapper;
7      }
8
9
10       .
11       //Misto pro metody pro CRUD operace implementovane z interfacu IMapper
12       .
13
14     @Override
15     public void compleateEntity(ElectricalComponent entity) throws
           No200Exception {
16         if(entity.getPath_previewImage()!=null)
17             entity.setPreviewImage(this.documentMapper.getDocumentFromPath(
                   entity.getPath_previewImage(), true));
18
19         if(entity.getPath_schematicImage()!=null)
20             entity.setSchematicImage(this.documentMapper.getDocumentFromPath(
                   entity.getPath_schematicImage(), true));
21
22         if(entity.getPath_datasheet()!=null)
23             entity.setDatasheet(this.documentMapper.getDocumentFromPath(entity.
                   getPath_datasheet(), false));
24     }
25 }
```

Výpis 15: Výňatek třídy ElectricalComponentMapper

Na řádku 1 vidíme, že třída *ElectricalComponentMapper* dědí z třídy **BaseMapper** a jako generický parametr je použita entita (třída) ElectricalComponent. Následně třída mapperu im-

plementuje interface **IMapper**, kde je opět předán jako generický parametr entita ElectricalComponent. V konstruktoru na řádcích 4 až 7 je předáván parametr *documentMapper* typu **DocumentMapper**, který se využívá v již zmíněné metodě pro dokončování entit, compleateEntity(T entity) z interface IMapper<T>. Tato metoda se nachází na řádku 15, ve které se využívá na řádcích 17, 20 a 23 právě předaný parametr v konstruktoru *documentMapper*, na kterém se volá metoda *getDocumentFromPath*.

Další funkčností aplikační vrstvy je definice a správa entit. Všechny entity dědí z třídy **BaseEntity**, jejichž výňatek byl již zmíněn v datové vrstvě, 2.2.2.1 na straně 28. Obsahuje parametr ID, který budou obsahovat všechny třídy, které z této třídy dědí. Za názvem je definováno, že třída dědí z interfacu Pacrelable. Tento interface nesouvisí s návrhem mé architektury samotné. Jde o interface systému Android, pomocí kterého můžeme objekt převést na skupinu bajtů a následně jej opětovně převést zpět na objekt. Jde o "balíčkování"objektů do podoby čistých binárních dat. Této vlastnosti se využívá pro přesun objektů mezi aktivitami (podobné jako formuláře) v prezentační vrstvě nebo při posílání objektů přes síťové rozhraní. Posílání objektů přes síť je nestandardní, většinou se posílají pouze data jednoduchých datových typů (boolean, integer, char), ale díky interfacu parcelable je možné zaslat celý objekt. Ovšem přijímací a vysílací strana musí pracovat na stejném principu, aby nenastala chyba při rozkládání/skládání objektu.

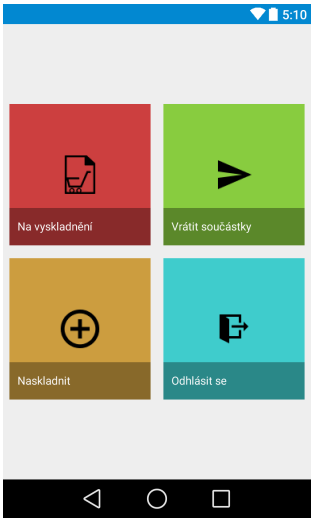
Poslední funkčností je definice třídy **MyApplication**, která má veškeré vlastnosti a metody statické. Tudíž se jedná o statickou třídu. V této třídě jsou uloženy informace o přihlášeném uživateli a informace, které musí být dostupné z kteréhokoliv místa v aplikaci bez závislosti na nějakém modulu v aplikaci, např. data mappery, adresa serveru pro volání API a metody pro ukládání a načítání dat nastavení aplikace z lokálního úložiště.

2.2.2.3 Prezenční vrstva (PresentationLayer)

Poslední vrstva je vrstva, která je nejblíže uživateli. Jedná se o vrstvu, která s uživatelem komunikuje. Předává mu informace a zpracovává požadavky od uživatele. Prezenční vrstva komunikuje pouze s aplikační vrstvou, tudíž je oddělena od práce související se zpracováním dat a dostává hotové výsledky. V datové a následně aplikační vrstvy mohou přicházet výjimky a prezenční vrstva je poslední vrstvou, která by tyto výjimky již měla zpracovat kompletně a v případě chyb informovat uživatele pomocí chybové hlášky. Prezenčí vrstva se skládá z několika částí

2.2.2.3.1 Aktivita

Aktivita je základním prvkem aplikace. Pomocí aktivity můžeme zobrazovat získané data z aplikační vrstvy na layoutu (rozložení komponent, přehled grafického rozhraní), který aktivitě předáme.



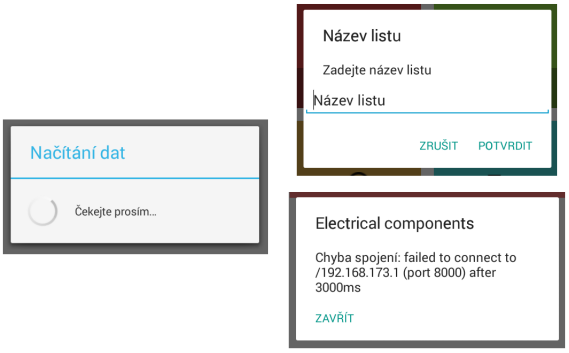
Obrázek 6: Ukázka layoutu v systému Android

2.2.2.3.2 Adaptéry a Views

Jsou nejčastěji spojovány se zobrazováním dat v listech/seznamech (ListView nebo jeho novější odnože (RecycleView)). Každá položka v seznamu má definovaný layout na základě adaptéru/-view, který se zároveň postará o naplnění dat do tohoto layoutu. Následně vyplněné layouty se předají na kreslení do seznamů.

2.2.2.3.3 Dialogy

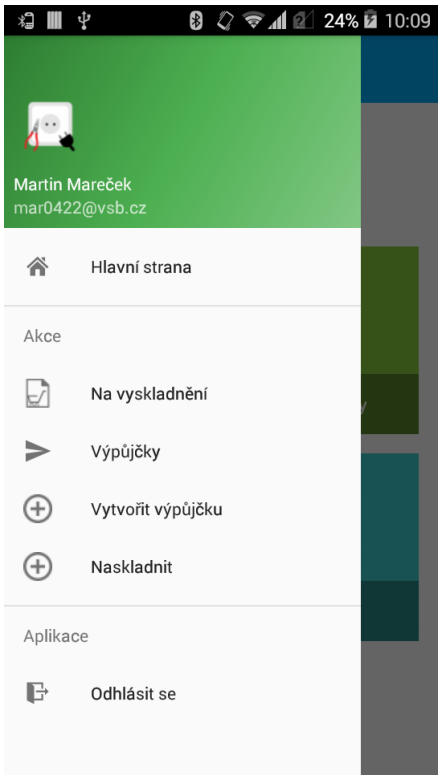
Klasické dialogy, jako je známe z různých systémů. Slouží pro rychlé oznamování informací, u kterých je skoro 100% jisté, že si je uživatel přečte a bude s jistotou informován. Mohou mít nastavenou vlastnost, že se daný dialog nezavře do doby, než na něj bude uživatel reagovat (např. kliknutí tlačítka). Další funkčností dialogů je dotazování pro informace případně rozhodování průběhu zpracování příkazů.



Obrázek 7: Ukázka dialogů v systému Android

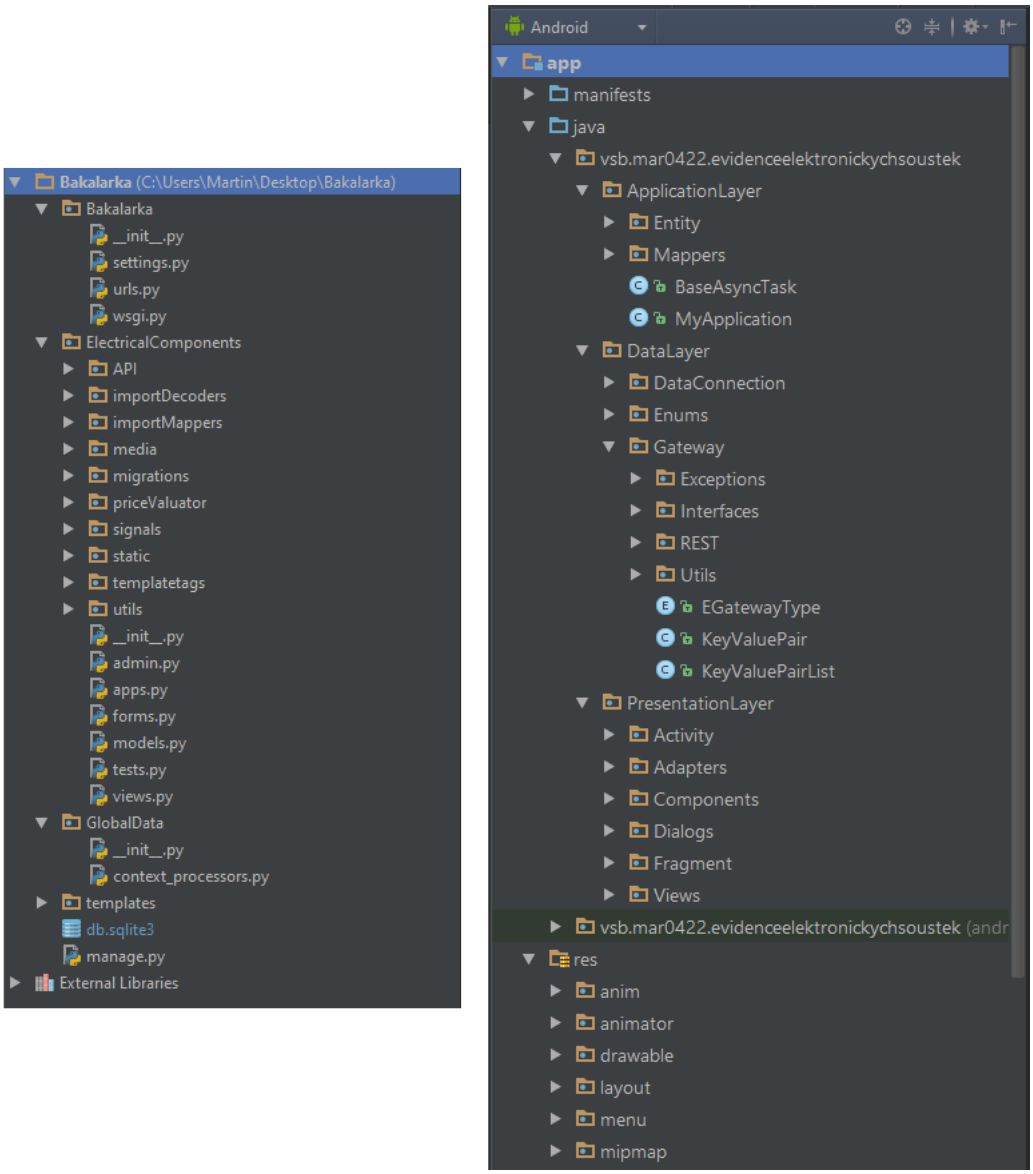
2.2.2.3.4 Fragmenty

Pokud potřebujeme uzavřít určitou funkčnost do nějakého bloku aplikace, používá se k těmto účelům v systémech android tzv. Fragment. Jde o komponetu, která obsahuje chování, data a zobrazení dat. Používání Fragmentu je dostupné až od verze Androidu 3.0 (Honeycomb), kde byla tato forma rozšířena díky zavedení výroby tabletů. Ty měli na tehdejší dobu obrovské displaye a tak se aplikace mohla rozložit do více bloků. Nevýhodou byl speciální kód pro mobilní zařízení a zvláštní kód pro tablety. Vývojáři si uvědomily této chyby zavčas a ve verzi Androidu 4.0 jsou již kódy pro použití fragmentů stejné na obou typech zařízení. Výhodou skládání aplikace do takových bloků je opakovatelnost použití v aplikaci na kterémkoliv místě. Každý fragment musí být svázaný s nějakou aktivitou. Bez aktivity fragment neví, na jakého předka je vázán a neví např., jak se má kreslit, jakými rozměry. Použití fragmentu ve fragmentu (blok v bloku) je zakázáno a při běhu aplikace je vyvolána výjimka. Od verze Androidu 4.2 je tento problém vyřešen zavedením tzv. Nested Fragmentů, které podporují mít v sobě další fragmenty. Typickým užití fragmentů je např. uživatelský panel, který má stejnou funkčnost kdekoliv v aplikaci a musí být dostupný všude nebo typický postranní navigační panel (NavigationDrawer), viz. obrázek 2.2.2.3.4



Obrázek 8: Ukázka fragmentu v systému Android

2.2.3 Kompletní přehled architektury



Obrázek 9: V levo je uvedena architektura systému Django, vpravo systém Android

2.3 Specifikace elektronické součástky

Prvním z požadavků pro vypracování bakalářské práce je evidence součástek s dostatečnými parametry. Každá elektronická součástka je specifická svými parametry a vlastnostmi, které mohou, a zároveň nemusí, být jedinečné. Ovšem některé vlastnosti jsou pevně dané a musí je splňovat každá součástka. Jde o vlastnosti jako je název, počet vývodů/pinů a package/obal/-pouzdro. Pro splnění dalšího úkolu z této práce, možnost importu souborů z návrhových softwarů na elektronická schémata, je přidána k elektronické součástce ještě jedna povinná vlastnost, a to je označení součástky ve schématu vzhledem k použitému softwaru. Proč je tomuto tak učiněno se vrátím později. Nepovinnými vlastnostmi jsou popis, označení (BC337 atp), hlavní hodnota (např. h_{21e}=160), parametry, dokumentace, náhled na součástku a schématické znázornění. Těmito parametry a vlastnostmi je každá elektronická součástka identifikovatelná a jedinečná.

2.3.1 Pouzdro elektronické součástky

Každá elektronická součástka obsahuje informace o tom, v jakém je provedení, neboli jaké má pouzdro. Každé pouzdro je specifické svým označením, jako např. TO-220 a TO-92 pro tranzistorové pouzdra a DIL/DIP (Dual in-line package) pro integrované obvody. Tyto pouzdra se řadí nejpoužívanější. Pouzdra mají daný pevný rozměr. Tyto parametry jsou uchovávány v jednotkách mm pro hodnoty X, Y a Z. Mohlo by se zdát, že pouzdrem je definován i počet vývodů dané součástky. Není tomu tak. Typickým příkladem je pouzdro TO-220, kde se může lišit počet pinů u výkonových tranzistorů. Typický počet vývodů u tohoto pouzdra je 3. Existují ovšem více emitorové tranzistory, tím pádem má toto pouzdro o vývody více. Proto nemůže být počet pinů uveden u součástky a ne u pouzdra.

2.3.2 Hlavní hodnota elektronické součástky

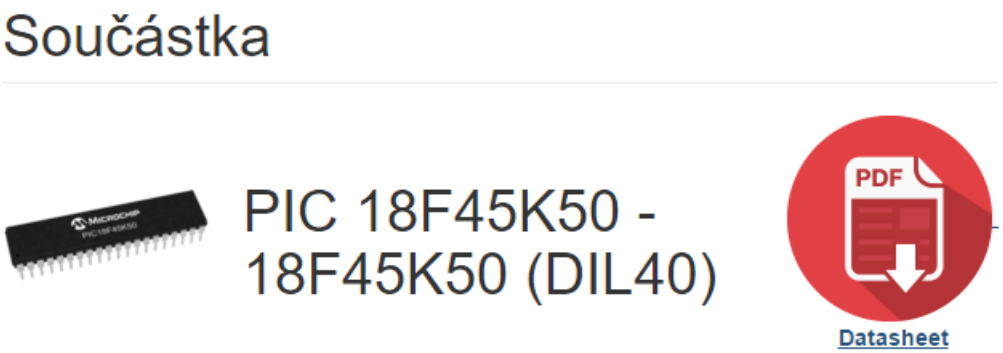
Součástka může obsahovat i takzvanou hlavní hodnotu neboli její hlavní prezenční hodnotu. Typicky jsou tyto hodnoty u pasivních součástek jako rezistory, kondenzátory a indukčnosti (tlumivky, ferity) nebo i u aktivních součástek jako bipolární tranzistory, kde se uvádí hodnota proudového zesilovacího součinitele β (h_{21e}). Tato vlastnost je evidována v několika parametrech. Prvním parametrem je hodnota, kde je uvedena v číselné podobě hodnota. Další z parametrů je jednotka, ve které je zadána hodnota, např. nF a pod nebo může být i bez jednotky (proudový zesilovací součinitel β). Následuje schématické značení pro větší přehlednost hodnot. Obsahuje textové hodnoty typu R,C,L a další. Poslední parametr je zápis hodnoty, jak je zapsána v návrhových softwarech. Tento parametr je povinný a používá se k rozpoznávání součástek při importu do systému z externích systémů. Pro ukázku může obsahovat hodnotu např. **100n**.

2.3.3 Parametry elektronické součástky

Součástka může mít i specifické parametry a vlastnosti jako napájecí napětí, příkon, tolerance hodnoty a další. Každý parametr se skládá ze dvou povinných hodnot. První hodnotou je název parametru. Další vlastností je hodnota. Poslední nepovinnou vlastností je jednotka, protože některé parametry mohou být uvedeny bez jednotky, např. počet AD převodníků nebo USB modulů u mikroprocesorů. Parametry nejsou pevně svázány k jedné součástce. To znamená, že parametr mohou sdílet více součástek současně, např. příkon u rezistorů. Čím více parametrizovaná součástka je, tím je lepší možnost hledat případné alternativy.

2.3.4 Dokumentace, náhled, schéma elektronické součástky

Tyto vlastnosti jsou nepovinné. Ovšem umožňují grafický přehled, schématické zapojení a přístup k dokumentaci dané součástky.



Obrázek 10: Náhled na zobrazení grafického náhledu a možnost zobrazení dokumentace

2.3.5 Označení ve schématu

Jde o seznam, ve kterém je uvedeno označení součástky pro daný návrhový software. Těchto vlastností se využívá při importování seznamu součástek do systému. Dle těchto vlastností se určuje součástka vedená v systému. Toto označení je proto velice důležité.

3 Řešení požadavků

3.1 Možnost skenování součástek

Pro možnost vedení počtů u elektronických součástek je nutné zavést další entitu, která se o uchovávání těchto informací bude starat a zároveň nebude zasahovat do struktury dat. Pro tuto možnost je v systému zavedena entita **Produkt**. Tato entita uchovává již zmíněnou vlastnost evidenci počtů součástek, odkaz na součástku, kterou eviduje a poslední vlastností je unikátní kód pro vygenerování čárového kódu typu EAN-13, který se následně používá pro skenování mobilním zařízením. Takže pomocí čárového kódu je mobilní aplikace se schopna dostat k elektronické součástce a jejich počtu na skladě. V případě změn, co se týče naskladnění součástek, není třeba zasahovat do elektronických součástek, ale je úprava entity produkt.

3.1.1 Čárový kód EAN

Čárový kód se skládá z bílého podkladu a černých pruhů, které mají různou šířku. Snímaná šířka bílého místa signalizuje počet logických 1 a obráceně, černá místo počet 0. Kód se skládá z několika číslic. Poslední číslicí, která do samotného kódu nepatří je číslo checksum, pomocí kterého se validují nasnímaná data. Tento checksum je počítán jako: **kód** *modulo* 10

Pro generování čárových kódů na platformě Django se využívá knihovna graphics, která je zahrnuta v interpretu Python. V systému je pevně zadána velikost generování tisknutelné verze čárového kódu. Systém by proto mohl být rozšířen o parametry při generování tisku jako rozměry, formát, zobrazení číslic pod kódem atp.

```
barcode = createBarcodeDrawing('EAN13', value=barcode_value, barHeight=30 * mm,
    barWidth=0.7 * mm, humanReadable=True)
Drawing.__init__(self, barcode.width, barcode.height, *args, **kw)
self.add(barcode, name='barcode')
```

Výpis 16: Generování čárového kódu na platformě Django

Na snímání kódu na platformě Android je použita externí knihovna ZXing [ZXing], protože systém Android neobsahuje žádné knihovny pro snímání kódů.

```
integrator = new IntentIntegrator(this);
integrator.setDesiredBarcodeFormats(Collections.singletonList(BarcodeFormat.
    EAN_13.toString()));
integrator.setCameraId(0);
integrator.setBeepEnabled(true);
```

Výpis 17: Příprava snímače čárového kódu na platformě Android

3.2 Integrace s návrhovým systémem

Požadavkem pro integraci je vytvoření projektu v návrhovém softwaru, následný export použitých součástek. Systém by měl podporovat nahrání a zpracování tohoto souboru a porovnání dat v systému. Prvním návrhem řešení bylo vytvoření speciální knihovny součástek pro návrhový systém, aby bylo možné následně pracovat s daty vyexportovanými z návrhu. Vyřešení problému tímto způsobem bylo provozuschopné, ale nedostatečné, co se týče modularity, která je také uvedena v zadání práce. Proto byl tento způsob zpracování zrušen a nahrazen novým. K zaručení modularity nejen na architektonické vrstvě, ale i v tomto případě, je v systému Djnago vytvořen balíček **importDecoders**. Tento balíček obsahuje třídy pro dekódovaní dat z importovaných souborů a předpis pro interface těchto tříd. Jak je zmíněno v architektuře systému Django [2.1.1], u jazyka Python [2.1.1.3], jazyk Python nepotřebuje interfacy pro svůj provoz, proto jsou přepisy interfaců uchovávány alespoň v textové podobě u tříd, které je využívají. Každá třída reprezentuje jeden návrhový systém, tedy obsahuje chování, jak se bude daný import součástek provádět. Pro použití v systému, aby se vědělo, jaký soubor s třídou se bude používat, je vytvořen v balíčku **importMappers** soubor *iDecoderMapper.py*, ve kterém je definována funkce pro výběr importMapperu. Jako parametr je jí předávám návrhový software, který je uveden v databázi systému.

```
from ElectricalComponents.importDecoders import iDecoder_EAGLE, iDecoder_PADS

def getDecoder(designsoftware):
    if(designsoftware.title == "EAGLE"):
        return iDecoder_EAGLE.importDecoder()
    if(designsoftware.title == "PADS"):
        return iDecoder_PADS.importDecoder()
```

Výpis 18: Mapper funkce pro výběr importovacího dekodéru

Každý import dekodér obsahuje dvě funkce. První, *validInputFile*, zkontroluje, zda je daný soubor správného formátu, který je přístupný pro následující import dat. Další funkcí je funkce *getItems*, která ze zadaného a předem zkontrolovaného souboru získává entity dočasných elektronických součástek, **CheckPartlistItem**. Tato entita obsahuje důležité parametry pro možnost vyhledání skutečné elektronické součástky v systému. V případě shody a nálezu součástky, je této entitě předána i entita elektronické součástky. Pokud není nalezena schoda součástky, hodnoty součáskty, pouzdra nebo počtu naskladněných součástek vzhledem k počtu požadovaných, je tato informace o varování případně chybě zanesena do entity checkPartlistItem, která je předána prezentační vrstvě pro informování uživatele o těchto chybách. Importovaný soubor je uložen do databáze pro opakované použití

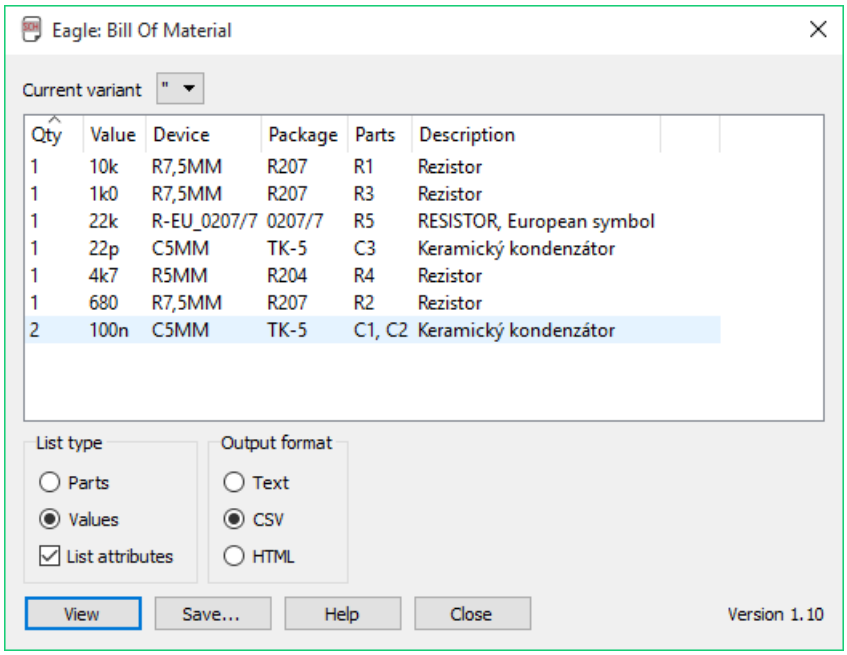
```
try:
      pcg = EC_Package.objects.get(name__exact=p_package)

      designation = DesignSoftwareDesignations.objects.filter(
          designation__exact=p_device).first()

      if len(p_value) > 0:
            mv = EC_MainValue.objects.get(mainValue__exact=p_value)
            ec = ElectricalComponent.objects.get(deviceDesignations=designation,
                mainValue=mv, package=pcg)
      else: #to ze soucastka nema hodnotu neznamena, ze nemuze existovat
            ec = ElectricalComponent.objects.get(deviceDesignations=designation,
                package=pcg)

      pD = Product.objects.get(eComp=ec)

      if pD.amount >= p_qty:
            validProductCount = True
            if pD.amount == p_qty:
                  anyError = True
      else:
            validProductCount = False
            anyError = True

      .
      .
      .

      vPli = CheckPartlistItem(product=pD, description=p_description,
          schematicChar=p_parts, schematicValue=p_value, amount=p_qty, package=
          p_package, device=p_device)
      productsForValidate.append({'errorMessage': error_findMsg, 'errorMessage2
          ': error_findMsg2, 'vPli': vPli, 'validProductCount':
          validProductCount})
```

Výpis 19: Výňatek validace parametrů z importovaného souboru pro software EAGLE

Obrázek 11: Ukázka kontroly naskladněných součástek z importovaného souboru

3.2.1 Návrhový software EAGLE

Pro vytvoření funkčnosti a testování byl použit návrhový software společnosti EAGLE [EAGLE]. Po vytvoření návrhu umožňuje software EAGLE export tzv. BOM souboru. Zkratka BOM má v informační technice více významů. V tomto případě se jedná o zkratku **Bill of materials**, v doslovném překladu **účet za materiál**. V tomto seznamu jsou uloženy veškeré použité součástky z daného návrhu, jejich počty a hodnoty. Export podporuje výstup do různých formátů, jako obyčejný textový soubor (TXT), HTML soubor a stěžejní **CSV**. Typ listu (*List Type*), aby byl tento seznam potvrzen jako validní pro import do systému, musí mít zvolen parametr **Values**, viz. obrázek kapitoly 3.2.1. V tabulce 3.2.1 je vzor vyexportovaného souboru CSV.



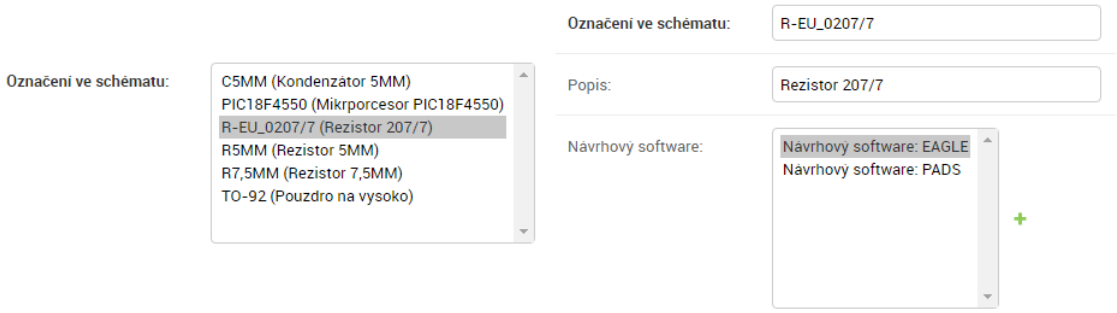
Obrázek 12: Vytvoření exportu partlistu z programu EAGLE

Qty	Value	Device	Package	Parts	Description
2	100n	C5MM	TK-5	C1, C2	Keramický kondenzátor
1	10k	R7,5MM	R207	R1	Rezistor
1	1k0	R7,5MM	R207	R3	Rezistor
1	22k	R-EU_0207/7	0207/7	R5	RESISTOR, European symbol
1	22p	C5MM	TK-5	C3	Keramický kondenzátor
1	4k7	R5MM	R204	R4	Rezistor
1	680	R7,5MM	R207	R2	Rezistor

Tabulka 4: Přehled exportovaných dat v CSV souboru

Dekodér musí znát, jakým způsobem jsou označeny elektronické součástky v návrhovém systému. Na vyřešení tohoto problému jsou v systému zavedeny entity **DesignSoftware** a **De-**

signSoftwareDesignations. Entity typu designSoftware obsahují názvy vývojových softwarů, které jsou použity v importMapperu. Následně entita designSoftwareDesignations obsahuje samotné označení součástky v návrhovém softwaru a následnou vazbu na tento software na entitu designSoftware. Jedno označení může naléhat více návrhovým softwarům. Např. označení C5MM může být uvedeno jak v programu EAGLE, tak v programu PADS [PADS]. Potom při dekódování jsou tyto informace použity pro validaci, zda je hledaná součástka z importovaného souboru obsažena i v systému.



Obrázek 13: Vlevo: Přiřazení označení k elektronické součástce. Vpravo: Označení elektronické součástky ve schématu

3.3 Komunikace s externími systémy

Další funkčností pro použitelnost systému byl požadavek na připojení k internetovým obchodům pro kontrolu cen a spočítání ceny pro daný návrh ze softwaru. Veškeré třídy pro tuto funkčnost jsou obsaženy v balíčku **priceValuator**.

3.3.1 Cenová nabídka

Ve výše uvedeném balíčku se nachází třída **priceOffer**. V této třídě se nacházejí informace o ceně jedné položky. Položky se nemusí prodávat po jednom kuse, ale například až po nějakém množstevním limitu. Proto je uveden jako další parametr počet kusů v balení. Posledním parametrem je cena celého balení, vzhledem k množstevnímu limitu.

3.3.2 Produktová nabídka

Další třída je **productOffer**. Obsahuje informace o nabídce produktu, jako název, symbol (unikátní označení v daném obchodu), odkaz na stránku produktu a také veškeré cenové nabídky [3.3.1] tohoto produktu.

3.3.3 Price valuator

Price valuator slouží pro získávání cenových nabídek. Třída **priceValuator** obsahuje metody, které dle zadaného textu vyhledají na daném internetovém obchodě produktové nabídky a v návaznosti na ně i jejich cenové nabídky. Každý priceValuator musí obsahovat metody, které jsou uvedené v souboru *interface*. **Nemusí** se jednat jen o internetové obchody, ale je to typický příklad napojení na cenovou politiku cizích zdrojů. Každá třída typu priceValuator musí obsahovat tyto metody:

- getID(self)
- getName(self)
- getProductOffers(self, symbol=None)
- searchProducts(self, searchText, searchPage=1)
- getPagerInfo(self, searchText)

Metoda **getID** musí vracet unikátní ID třídy priceValuator. Metoda **getName** slouží pro pojmenování priceValuatoru pro následné zobrazení v prezenční vrstvě (např. TME [TME API]). Další je metoda **searchProducts**, která vrací *symboly* produktů, což jsou unikátní čísla produktů, které slouží následně pro vyhledávání produktových nabídek. **GetProductOffers** získává produktové a cenové nabídky k předanému symbolu. Poslední metoda je **getPagerInfo**, která vrací počet stran s výsledky v případě vícero výsledků při vyhledávání symbolů metodou *searchProducts*.

```
def searchProducts(self, searchText, searchPage=1):
      res = []

      params = {
            'SearchPlain': searchText,
            'Country': self.country,
            'Language': self.lang
      }
      if(searchPage>1):
            params['SearchPage'] = searchPage

      response = self.api_call('Products/Search', params, self.token, self.
         app_secret)
      for product in response['Data']['ProductList']:
            symbol = product['Symbol']
            title = product['Description']
            page = product['ProductInformationPage']
```

```
            po = productOffer(self.ID, symbol, title, page)
            res.append(po)

    return res
```

Výpis 20: Ukázka komunikace s API rozraním TME

Těmito základními metodami lze dosáhnout nejzákladnější a nejobecnější komunikace s externím rozhraním. Pro ukázku byl zvolen pouze jeden internetový obchod jako zdroj dat, který je duplikován, aby byla možnost ukázky modularity

3.3.4 Price valuators

Soubor *priceValuators.py* obsahuje metodu **getValuators**, která vrací kolekci tříd dostupných pro cenovou validaci. Tato kolekce se využívá ve vrstvě kontroléru, kde se zpracovávají data od uživatele a následně vrací výsledky

Obrázek 14: Ukázka napojení cenových API na systém

3.4 Komunikace s mobilní platformou

Pro splnění požadavku multiplatformnosti a používání mobilních zařízení pro snímání součástek pomocí čárových kódů, musí systém nějakým způsobem poskytovat data externím systémům. Důležité při této akci je skrýt způsob uchování a zpracování dat, jde o tzv. know-how domény, pod kterou je software provozován. Pro tyto podmínky se využívá API rozhraní, které je třeba využito při napojení systému na internetové obchody, viz. kapitola 3.3. Pro poskytování dat má tedy systém implementováno vlastní rozhraní, díky kterému poskytuje data. Implementace API funkce je definována adresou, která reprezentuje danou funkci API rozhraní a kontroléry, které se starají o požadavky na tyto adresy, tedy zpracovávají, případně poskytují, data předané pomocí parametrů HTTP požadavku. Veškeré souvislosti s poskytováním API rozhraní se nachází v balíčku *API* a souboru *urls.py*

```
url(r'^api/v1/getToken\$', 'ElectricalComponents.API.v1.views.getToken'),
url(r'^api/v1/userExist\$', 'ElectricalComponents.API.v1.views.userExist'),
url(r'^api/v1/user\$', 'ElectricalComponents.API.v1.views.user'),
url(r'^api/v1/electricalComponent\$', 'ElectricalComponents.API.v1.views.
    electricalComponent'),
url(r'^api/v1/electricalComponent/(?P<eCompID>[0-9]+)\$', 'ElectricalComponents
    .API.v1.views.electricalComponent'),
```

Výpis 21: Výňatek definic url adres a kotrolérů pro API rozhraní

3.4.1 Zabezpečení

Pro obranu proti útokům na poskytované rozhraní nebo pro autentizaci uživatelů používajících rozhraní je třeba zavést opatření. Proti útokům se dá použít základní opatření např. pomocí omezení počtů požadavků za časovou jednotku z jednoho zdroje. Data by měl získávat pouze uživatel, který k nim bude mít přidělený přístup, tedy požadavek na tedy bude doprovázen identifikací uživatele. Tento proces je nazývám autentizací uživatele, tedy za koho se daný požadavek vydává. Autentizace uživatele je rozpoznávána pomocí zaslané hodnoty **Token** v požadavku. Toto pole obsahuje unikátní identifikátor, díky kterému můžeme požadavek autentizovat. Tyto tokeny musí být uloženy a navázány na uživatele v systému. U tokenu může být problém se zabezpečením, a to tím způsobem, že útočník token uživatele získá a poté se může za tohoto uživatele vydávat. Nejjednodušší způsob ochrany je svázat tento token i s IP adresou sítě, kde budou vznikat požadavky. Generování tokenů se může provádět pomocí nějakého hashovacího algoritmu (např. MD5, SHA1), kde vstupním parametrem bývá např. název emailové schránky. Může se stát, že vznikne stejný hash pro 2 různé vstupy hashovací funkce. Tyto kolize se musí hlídat a vyřešit, aby se dva uživatelé nemohli vydávat pod jedním tokenem. V této práce je gene-

rování tokenů založena na náhodném generování řetězce a následného hashování tohoto řetězce. Pro tuto funkčnost slouží metoda *genToken* v entitě **Token**, která je vázaná na uživatele

```
def genToken():
      generated = None
      conflicts = 1
      while conflicts == 1:
            generated = binascii.b2a_hex(os.urandom(16))

            conflicts = Token.objects.filter(token=generated).count()

      return generated
```

Výpis 22: Funkce genToken pro generování Tokenu pro přístup k API rozhraní

Funkce vygeneruje náhodný hash, následně tento hash zkontroluje, zda se v datovém úložišti již tento hash nenachází. Pokud ne, hash se vytvoří pod daného uživatele. V opačném případě se generuje nový náhodný hash a postup se opakuje.

3.4.2 Zpracování požadavku

V souboru s adresami, *urls.py*, je ke každému požadavku na adresu přiřazen i kontrolér, který tento požadavek zpracuje. Před každým kontrolérem je uvedený dekorátor **REST_isValidRequest**, viz. sekce [2.2.1.3], který se stará o validaci zaslaného požadavku.

```
1  class REST_isValidRequest(object):
2      def __init__(self, authorization=True, methods=["GET"], requireQueryParams
           =[], requireHeaderParams=[]):
3          self.authorization = authorization
4          self.methods = methods
5          self.requireQueryParams = requireQueryParams
6          self.requireHeaderParams = requireHeaderParams
7
8          self.queryParams = []
9          self.queryParamKeys = []
10
11     def __call__(self, f):
12         @csrf_exempt
13         def wrap(request, *args, **kwargs):
14             error = None
15             code = 400
```

```
16
17              CONTENT_TYPE = request.META['CONTENT_TYPE']
18              QUERY_STRING = request.META['QUERY_STRING']
19              REQUEST_METHOD = request.META['REQUEST_METHOD']
20
21              # Snaha o vyparsovani QUERY stringu
22              if (QUERY_STRING is not "") and (QUERY_STRING is not None) and (
                    error is None):
23                  try:
24                      for qP in QUERY_STRING.split("&"):
25                          prm = qP.split("=")
26                          self.queryParams.append({prm[0]: prm[1]})
27                          self.queryParamKeys.append(prm[0])
28                  except:
29                      error = "Nepodarilo se spravne ziskat parametry QUERY stringu
                            "
30
31              # Kontrola QUERY string dat
32              if (len(self.requireQueryParams) > 0) and (error is None):
33                  missParams = []
34                  for reqKey in self.requireQueryParams:
35                      isIn = False
36                      for key in self.queryParamKeys:
37                          if key in reqKey:
38                              isIn = True
39                      if isIn is False:
40                          missParams.append(reqKey)
41
42                  if len(missParams) > 0:
43                      error = "Je pozadovan QUERY string parametr(y): %s" % str(
                            missParams)
44
45              #Kontrola HEADER dat
46              if (self.requireHeaderParams is not None) and (error is None):
47                  missParams = []
48                  for hP_key in self.requireHeaderParams:
49                      value = getHeaderData(request, hP_key)
50                      if value is None:
51                          missParams.append(hP_key)
```

```
52
53                  if len(missParams) > 0:
54                      error = "Je pozadovan HEADER parametr(y): %s" % str(
                            missParams)
55
56              #Overeni metody
57              if (REQUEST_METHOD not in self.methods) and (error is None):
58                  error = ("Metoda %s neni podporovana. Je treba: %s" % (
                        REQUEST_METHOD, self.methods))
59
60              #Overeni Autorizace
61              if (self.authorization is True) and (error is None):
62                  AUTHORIZATION = getHeaderData(request, "AUTHORIZATION")
63                  if AUTHORIZATION is not None:
64                      token = Token.objects.filter(token=AUTHORIZATION).first()
65                      if token is None:
66                          error = "Zaslany token se neshoduje s zadnum uzivatelem"
67                          code = 403
68                  else:
69                      error = "Je pozadovano pole autorizace: AUTHORIZATION"
70
71              if error is not None:
72                  return generateOutput_detail(error, code)
73              else:
74                  return f(request, *args, **kwargs)
75
76
77          wrap.__doc__ = f.__doc__
78          wrap.__name__ = f.__name__
79          return wrap
```

Výpis 23: Dekorátor REST_isValidRequest

Na řádku 2 jsou uvedeny výchozí parametry pro každý kontrolér pro zpracování HTTP požadavku. Jsou zde uvedeny parametry jako povinnost autorizace při volání daného kontroléru. Tento parametr je ve výchozím stavu nastaven, aby každý požadavek vyžadoval ověření token kódem. Toto ověření ale musí být u některých funkcí vypnuto, ale zároveň musí splňovat další parametry. Typickým příkladem je právě získání Token kódu, kde potřebujeme komunikovat s API, ale ještě daný Token nemáme a potřebujeme ho získat. Jako další parametr je uvedení metody/metod, jakým způsobem mohou být url adresy volány. Posledními parametry jsou pa-

rametry, kterými definujeme, jaké vlastnosti musíme předat danému kontroléru, aby byl validní a mohl začít provádět svůj běh. Pokud selže kterýkoliv krok validace, je navrácena volající straně chybová hláška s problémem. Od řádku 21 až 29 je snaha o dekódování parametrů z URL adresy. Na řádcích 31 až 42 je prováděna kontrola zadaných parametrů z URL adresy na základě vstupních parametrů dekorátoru. Mezi řádky 45 až 54 je prováděna stejná kontrola, akorát zdrojem kontrolovaných parametrů není URL adresa, ale hlavička HTTP, kde se mohou také přenášet parametry. Na řádku 56 začíná kontrola typu požadavku. Může nabývat hodnot dle definice HTTP protokolu, např. GET nebo POST, [HTTP metody]. Od řádku 60 následuje blok, který zkontroluje, zda kontrolér vyžaduje autorizaci a následně může zkontrolovat platnost zaslaného Tokenu. Pokud se vyskytne chyba během validace vstupních dat, je vygenerován chybový výstup, viz. řádek 72, v opačném případě je volána funkce za dekorátorem (řádek 74).

```
1 @REST_isValidRequest(methods=['GET'], authorization=False, requireQueryParams=[
      'username'])
2 def userExist(request):
3     exist = False
4     username = getQueryStringData(request, 'username')
5     if User.objects.filter(username=username).first() is not None:
6         exist = True
7 
8     return generateOutput_200({'result': str(exist)})
```

Výpis 24: Kontrolér API rozhraní s použitým dekorátorem

Zpracování kontroléru je následující. Provede se dekorátor na řádku 1. V případě úspěchu se volá kontrolér *userExist* na řádku 2. Na řádku 4 získáváme z požadavku (parametr **request**) query hodnotu **username**. Nemusíme se obávat chyby nebo chybové hlášky. Jelikož v parametru dekorátoru je tento parametr vyžadován a kontrolér byl spuštěn, znamená to, že tento parametr skutečně požadavek obsahoval. Následně na řádku 5 je provedena kontrola, zda se v systému uživatel s tímto jménem nachází. Výsledek funkce je navrácen jako úspěšný na řádku 8.

3.4.3 Formát přenášených dat

Výstupní formát dat musí být univerzální pro všechny platformy a jednoduše čitelný pro strojové zpracování. Dříve se pro přenos dat hojně využívalo značkovacího jazyku XML, ale v dnešní době je na ústupu díky modernějším způsobům zápisu dat. Jedním z hlavním, který se v dnešní době hojně využívá právě na webové platformě je způsob zápisu JSON. Výhodou JSON je např. úspora přenášených dat, a to formou, že JSON neobsahuje značky jako XML a následné uzavírací značky jako dlouhé řetězce, ale pouze jako složené závorky případně hranaté závorky. Úspora JSON se hodí právě na webové aplikace pro mobilní telefony, kde se bere ohled na stahování

co nejmenšího počtu dat kvůli tarifikaci mobilních operátorů atp. JSON může obsahovat pouze data typu klíč:hodnota. Výhoda XML formátu spočívá v možnosti, že každý záznam může nést svůj název, atributy a vlastněnou hodnotu. V JSON se musí tyto vlastnosti obcházet např. pomocí vytvoření pomocného objektu, který tyto vlastnosti zahrne. Pro formát dat v této práci je zvolen formát dat JSON.

3.4.4 Serializéry

Z rozhraní může přijít požadavek pro navrácení entity ze systému. Toto je důležitý krok v poskytování dat, zároveň i nebezpečný. V chybě při návrhu by mohla nastat situace, kdy poskytneme navenek systému citlivá uživatelská data, která mají být jen a jen pouze uvedená v systému, jako je např. přihlašovací jméno a heslo. Proto se výstupní entity musí omezit v tom, jaká data budou poskytovat. Z tohoto důvodu jsou zavedeny serializéry, které jsou uvedeny v souboru *serializers.py*. Jde o třídy, které seskupují informace. V serializéru se definují atributy, které budou poskytovány. Tímto se zamezí odesílání citlivých dat. Některé atributy můžeme chtít poskytovat u všech entit, např. atribut ID. Aby se tento atribut nemuset uvádět u každého serializéru, tak každý serializér volá metodu **serialize** z třídy *BaseSerializer*, která zajistí serializaci povinných atributů pro všechny entity. Serializéry vrací data ve formátu JSON.

```
class UserSerializer():
    @staticmethod
    def serialize(entity):
        return BaseSerializer.serialize(entity, params=['username', 'email', '
            first_name', 'last_name'])
```

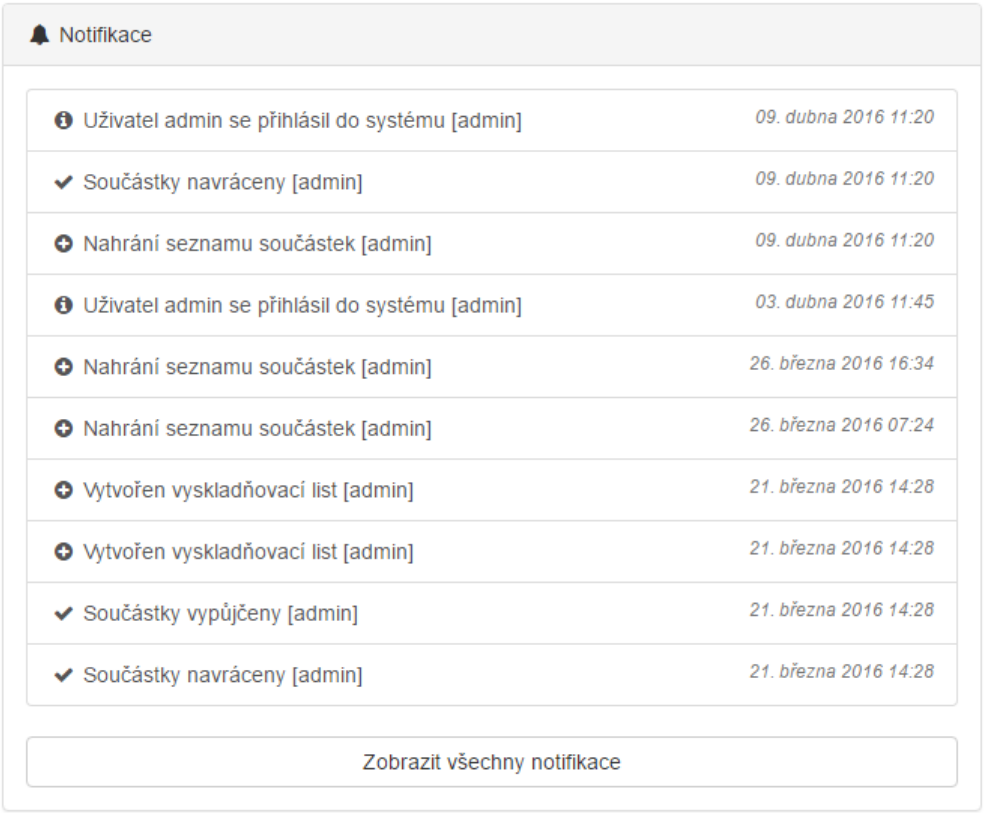
Výpis 25: Serializér pro entitu User

3.4.5 Verzování API

S postupem vývoje se architektura systému může měnit. Hlavně struktury entit, na které následně nebude sedět aplikační logika. Ovšem externí systémy o této změně nic neví a musí být zachována kompatibilita mezi předchozí a novou verzí. Proto je zavedeno verzování API rozhraní. Každá verze API rozhraní má svůj balíček nazvaný dle verze rozhraní. Jaké verze jsou použity se uvádějí v souboru *urls.py*, kde jsou uvedeny veškeré adresy pro zpracování požadavků. Můžeme tedy provozovat více verzí API současně či dokonce propojit používání verzí rozhraní (např. požadavek na získání uživatele z API verze 2 bude ve skutečnosti volat stále starší funkci z verze 1).

3.5 Auditování v systému

Auditování slouží pro sledování změn v systému. Nejčastěji se používá u systémů, kde k systému přistupuje více uživatelů současně a manipulují s daty. V případě problémů je třeba zpětně dohledat, kdo tento problém způsobil případně jaká akce tento problém vyvolala. Jde o logování akcí v systému. Logování je nejčastěji používáno při změně dat. Aby se tyto změny daly sledovat, je v architektuře Django možnost tzv. **signalizací**. Tyto signály jsou zapsány v balíčku *signals* v třídě **signals.py**. Každý signál je definován akcí, kdy se má spouštět a modelem, nad kterým se akce bude spouštět. Typ akce se skládá ze 2 částí. V první části se nachází čas, kdy se bude akce spouštět, zda-li **pre**(před) nebo **post**(po) operaci. Druhá část je akce samotná, jako **Save**(uložení) a **Delete**(smazání) modelu z datového úložiště. Pokud je akce validní, je provedena metoda, která je pod touto akcí definována. Aby byl uživatel o těchto akcích informován, tak mu systém poskytuje tyto informace pomocí **notifikací**. Notifikace obsahuje informace o uživateli, ke kterému se notifikace vztahuje, data o původu, tedy proč byla notifikace vyvolána a datum vzniku notifikace. Následně jsou tyto notifikace ukládány a zobrazovány v uživatelském rozhraní. Tudíž uživatel vidí audity v systému



Obrázek 15: Zobrazování auditování v systému

3.6 Třídní diagram

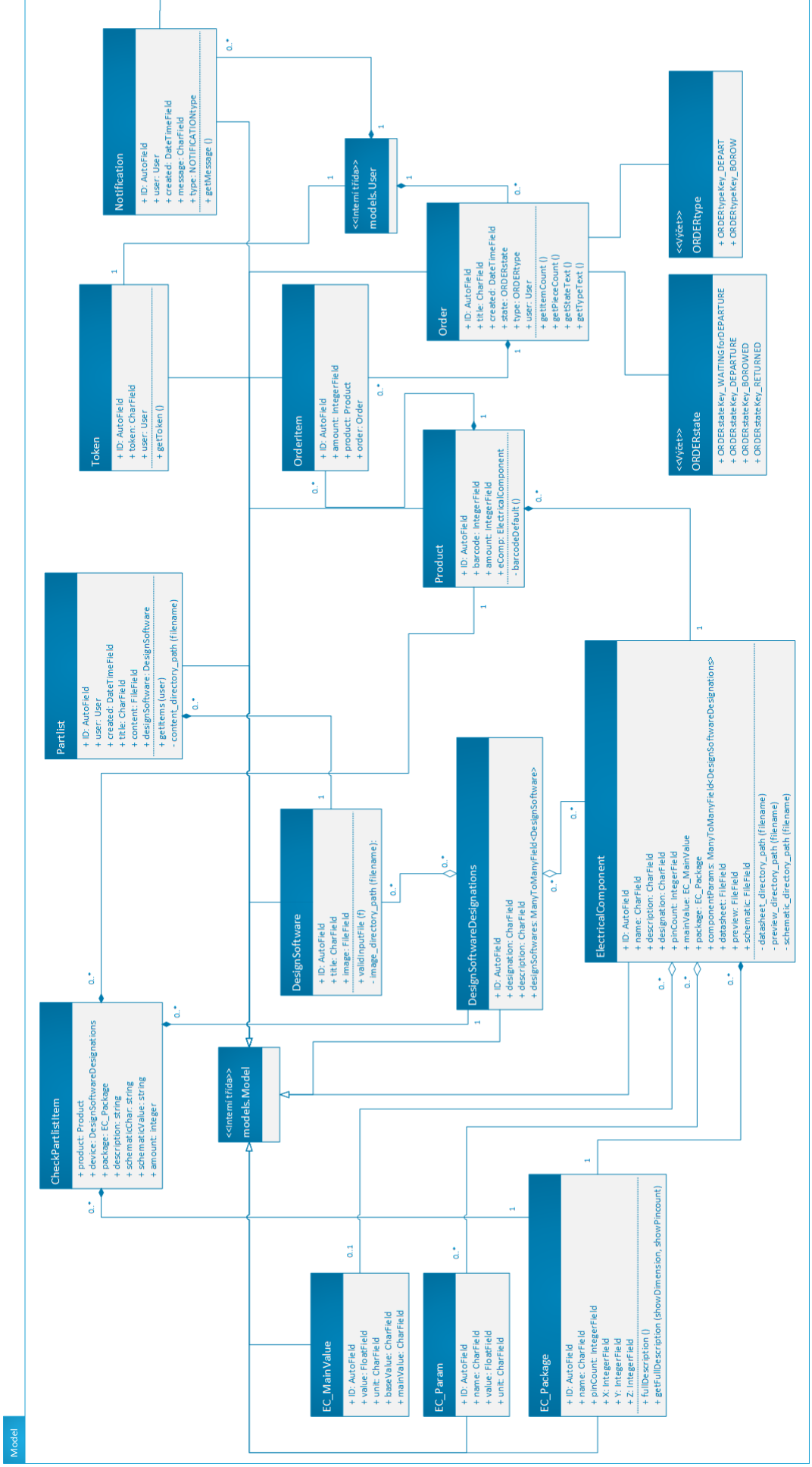
Pro lepší náhled na architekturu systému jsou níže uvedeny třídní diagramy obou architektur. Podrobnější přehled naleznete na přiloženém CD. Ukázku třídního diagramu naleznete na straně 54 a 53

3.7 Zajištění modularity

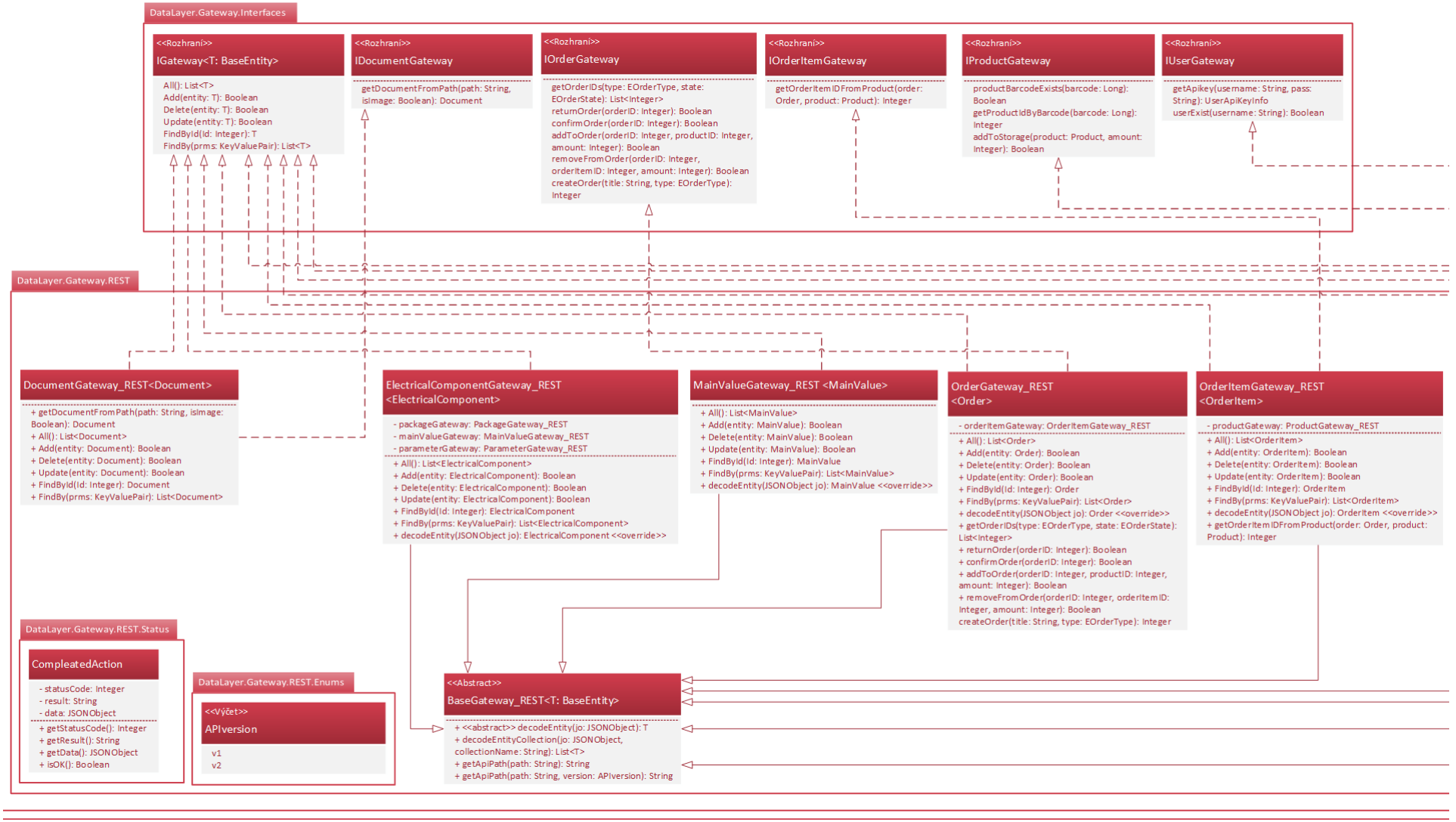
Modularitou je myšleno rozložení funkčnosti systému do bloků, které mohou být zaměnitelné. Pro používání modularity jsou v práci použity balíčky. V případě napojení na internetové obchody a importu součástek z externích systémů se ještě využívá jednotný interface. Výhodou modularity systému je zaměnitelnost bloků. Tudíž v případě rozšíření či opravě funkčnosti systému není třeba zasahovat do celé architektury, ale pouze jen do určité části (bloku), ve které se řešený problém nachází. Důležitá je také závislost modulů na sobě. Pokud budou modely navázány přes více vrstev, mohou nastat při úpravách problémy, kdy oprava v jednom modulu bude mít za následek způsobení chybovosti závislé vrstvy. V architektuře Django je modularita zařízena modelem MTC. Na architektuře Android je zřízena třívrstvá architektura. Jedno z nejpoužívanějších řešení je rozdělení funkčností do samostatných knihoven. Pro systém Windows jsou nejtypičtější DLL knihovny.

3.8 Obsah přiloženého CD

- **Adresář Architektura_Django** - Architekturu Django
- **Adresář Architektura_Android** - Architekturu Android
- **Adresář UML diagramy** - Obsahuje třídní diagramy obou architektur
- **Adresář Vzorové vstupní import soubory** - CSV soubory pro otestování importovací funkce systému



Obrázek 16: Část architektury Django



Obrázek 17: Část datové vrstvy architektury Android

4 Závěr

Cílem práce bylo vyvinou informační systém, což obsah práce splnil. Všechny požadavky dle zadání byly splněny, vypracovány a označeny jako funkční. Přidány byly i funkčnosti navíc jako např. komunikace se systémy třetích stran (internetové obchody). Systém je provozuschopný. Problém uchovávání dat byl úspěšně vyřešen pomocí částečného převzetí modelu datového skladu. Co se týče elektronické součástky, tak veškeré parametry a vyhledatelnost součástky vázané na návrhový software, tak v prvopočátcích práce se směr vydával vytvářením vlastních knihoven pro tyto softwary. Tento problém byl nakonec vyřešen a nyní je systém univerzální. Ovšem při vytváření nových součástek se musí definovat jejich označení v těchto návrhových softwarech. Následné vyhledávání součástek pomocí hlavní hodnoty závisí na přesně zadaných hodnotách, dle celosvětové normy zápisu elektrických hodnot. Celý systém je provozován na platformách, které jsou dostupné zdarma. Proto je provozování systému jednoduché, díky komunitě lehce rozšířitelné a levné.

Pro vypracování byly důležité hlavně předměty během studia, díky kterým se dal systém, alespoň v základní podobě, vyvinout a byl provozuschopný. Hlavní předmětem byl vývoj informačních systémů (VIS), kde byly předány důležité informace pro návrh architektury. Tedy architektonické vzory, jako je zde použitá třívrstvá architektura nebo MVC model a návrhové vzory pro přístup k datům a organizaci objektů v aplikaci, DataMapper pro komunikaci s uloženými daty a Gateway pro umožnění přepínání datových zdrojů. Dalším předmětem byl předmět softwarové inženýrství (SWI), odkud jsem čerpal metodiky vývoje. Tedy specifikaci požadavků (zadání práce), analýzu problému, následnou implementaci a testování. Každý z požadavků zadání prošel tímto zpracováním. Předmět skriptovací a programovací jazyky (SPJA), ve kterém jsou se dozvěděl právě o frameworku Django, který se používá na velkých projektech jako např. Youtube firmy Google. Posledními předměty jsou programovací jazyky 1 (PJ I) a tvorba aplikací pro mobilní zařízení 2 (TAMZ II) pro architekturu Android.

Systém může být do budoucna rozšířen o cenovou politiku. To znamená, že by se dali počítat ceny součástek při importování z externích softwarů. Problém vzniká při nákupu součástek a následného naskladnění. Pokud budu mít ve skladu 20 součástek o jedné ceně a přikoupím 10 dalších těchto součástek za rozdílnou cenu než je v systému, jakým způsobem tyto záznamy evidovat? Mají se ceny průměrovat či počítat jednotlivě? To už záleží na doméně, ve které bude systém provozován. Tato funkčnost by musela být opět modulární a volitelná přes administraci. Dalším rozšířením může být např. automatické objednávání součástek při vyskladnění. Tato funkčnost ovšem závisí na možnosti poskytovaných API rozhraní internetových obchodů. Dalším rozšířením může být vyhledávání alternativ součástek na skladu pomocí parametrů součástek. Nikoliv jen pomocí hlavních hodnot.Pro řešení tohoto problému by např. mohl být použit vyhledávací algoritmus na podobnost pomocí vektorů.

Literatura

[Microsoft] Oficiální domovská stránka Microsoft, https://www.microsoft.com/cze/, 1.4 2016

[Linux] České stránky systému GNU/Linux, http://www.linux.cz/, 1.4 2016

[Apple] Apple (Česká republika), www.apple.com/cz/, 1.4 2016

[Python] Python.org, https://www.python.org/, 1.4 2016

[Jinja] Jinja engine http://jinja.pocoo.org/, 1.4 2016

[Open Handset Alliance] Open Handset Alliance http://www.openhandsetalliance.com/, 1.4 2016

[Google] Společnost Google https://www.google.cz/intl/cs/about/company/, 1.4 2016

[Xamarin] Xamarin Framework https://www.xamarin.com/, 1.4 2016

[Top programovací jazyky] Nejlépe hodnocené programovací jazyky http://www.tiobe.com/tiobe_index, 1.4 2016

[Hodnocení vývoje mobilních platforem] Hodnocení vývoje mobilních platforem za období 2012-2015 http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 3.4 2016

[Architektura systému Android] Grafické znázornění architektury systému Android https://en.wikipedia.org/wiki/File:Android-System-Architecture.svg, 3.4 2016

[MTV architektura] MTV architektura http://www.thomaswhitton.com/django-presentation/#/4, 3.4 2016

[Bootstrap] Bootstrap framework http://getbootstrap.com/, 3.4 2016

[Design Pattern - Factory] Návrhový vzor továrna http://www.tutorialspoint.com/design_pattern/factory_pattern.htm, 5.4 2016

[Design Pattern - Data Mapper] Návrhový data mapper http://martinfowler.com/eaaCatalog/dataMapper.html, 5.4 2016

[ZXing] Komponenta pro snímání čárových kódů na platformě Android https://github.com/zxing/zxing, 9.4 2016

[EAGLE] Návrhový software EAGLE http://www.cadsoftusa.com/, 9.4 2016

[PADS] Návrhový software PADS https://www.pads.com/, 9.4 2016

[TME API] Developer stránka pro možnost využívání API rozhraní firmy TME https://developers.tme.eu/en/, 9.4 2016

[HTTP metody] HTTP methods https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html, 10.4 2016