Univerzita Pardubice
Fakulta elektrotechniky a informatiky

# Nástroj pro efektivní práci s JSON

Karel Andres

Bakalářská práce
2021

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2019/2020

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

| | |
|---|---|
| Jméno a příjmení: | **Karel Andres** |
| Osobní číslo: | **I17089** |
| Studijní program: | **B2646 Informační technologie** |
| Studijní obor: | **Informační technologie** |
| Téma práce: | **Nástroj pro efektivní práci s JSON** |
| Zadávající katedra: | **Katedra informačních technologií** |

## Zásady pro vypracování

Cílem bakalářské práce je vytvořit a otestovat paměťově i výpočetně efektivní nástroj pro práci s rozsáhlými JSON soubory.

Nástroj musí umožňovat přehlednou reprezentaci dat a realizovat efektivní operace nad rozsáhlými daty – např. validaci dat, filtraci dat dle klíčů, filtraci dat dle hodnot, případně další užitečné operace.

Text bakalářské práce musí obsahovat popis použitých datových struktur k realizaci jednotlivých operací s důrazem na efektivitu použitých řešení.

Předpokládaným programovacím jazykem je Java 8.

Rozsah pracovní zprávy: **30-40 normostran**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

BASSETT L. Introduction to JavaScript Object Notation: A To-the-Point Guide to JSON. O'Reilly Media; 1 edition (August 20, 2015), 126 pp. ISBN-13: 978-1491929483.

CUTAJAR J. Beginning Java Data Structures and Algorithms: Sharpen your problem solving skills by learning core computer science concepts in a pain-free manner. Packt Publishing (July 30, 2018), 202 pp. ISBN 978-1789537178.

Vedoucí bakalářské práce: **doc. Ing. Michael Bažant, Ph.D.**
Katedra softwarových technologií

Datum zadání bakalářské práce: **15. listopadu 2019**
Termín odevzdání bakalářské práce: **7. května 2020**

Univerzita Pardubice Česká republika L.S.

**Ing. Zdeněk Němec, Ph.D.**
děkan

**Ing. Lukáš Čegan, Ph.D.**
pověřený vedením katedry

V Pardubicích dne 17. prosince 2019

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že 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, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 11. 5. 2021

Karel Andres

# PODĚKOVÁNÍ

Mé díky patří doc. Ing. Michealu Bažantovi, PhD., vedoucímu práce, za poskytnuté konzultace a pomoc při psaní této bakalářské práce. Děkuji Bc. Pavlu Křivdovi, který mi svými radami a dodanou motivací velmi pomohl při psaní této práce. Dále bych chtěl moc poděkovat své rodině a přítelkyni za nejen psychickou podporu při tvorbě této práce i po celou dobu mých studií.

**ANOTACE**

Cílem této práce je vytvořit a otestovat paměťově i výpočetně efektivní nástroj pro práci s rozsáhlými JSON soubory.

Nástroj by měl umožňovat přehlednou reprezentaci dat a efektivně realizovat některé z operací nad jeho daty (validace, vyhledávání, filtrace dat dle klíčů, filtrace dat dle hodnot či další užitečné operace).

Při realizaci tohoto projektu je předpokládaným využitým programovacím jazykem Java.

**KLÍČOVÁ SLOVA**

JSON, Java, JavaFX, datové struktury

**TITLE**

Tool for effective work with JSON

**ANNOTATION**

The aim of this work is to create and test memory and computing efficient tool to working with JSON files.

The tool should allow well-arranged representation of data and effectively to implement some of the operations over his data (validation, finding, data filtering by keys or values or other useful operations).

In implementation of this project, Java is assumed to be used.

**KEYWORDS**

JSON, Java, JavaFX, data structures

# OBSAH

# SEZNAM OBRÁZKŮ

# SEZNAM TABULEK

# SEZNAM UKÁZEK ZDROJOVÉHO KÓDU

# SEZNAM ZKRATEK

| | |
|---|---|
| API | Application Pogramming Interface |
| CSS | Cascading Style Sheets |
| EBNF | Extended Backus Naur Form |
| GUI | Graphics User Interface |
| IDE | Integrated Develpment Environemt |
| IT | Information Technology |
| JDK | Java Develpment Kit |
| JRE | Java Runtime Environment |
| JSON | Java Script Object Notation |
| SHA1 | Secure Hash Algorithm 1 |
| XML | Extensible Markup Language |

# ÚVOD

Ve chvíli, kdy se využíval pro přenos dat na webu formát XML, který z pohledu vývojářů v JavaScriptu měl hodně nedostatků, práce s XML byla komplikovaná, tak se na světě objevil formát JSON. JSON sice nemůže konkurovat XML při zápisu celých dokumentů, ale při výměně dat mezi webovými aplikacemi jednoznačně boj vyhrál. Možná i proto se stal JSON jedním z nejpoužívanjších formátů ve webových aplikacích. [1]

Tato bakalářské práce je zaměřena na vývoj efektivního nástroje pro prási s JSON soubory. Nástroje pro práci s JSON soubory již samozřejmě existují, ale žádný z nich nedokázal spojit více funkcionalit do jednoho, a když už spojuje více funkcionalit, tak nefungoval moc efektivně. Proto je pro mě výzvou pracovat na této bakalářské práci, jejímž cílem je vytvořit efektivní nástroj, který spojí více funkcionalit. Před zpracováním této práce je nutné si nastudovat potřebné informace k jejímu zpracování, zvolit vhodnou strategii pro psaní, zanalyzovat již existující nástroje, navrhnout nástroj nový, který bude něčím jiný.

Myslím si, že JSON soubory se v dnešním světě informatiky používají hojně. Například pro přenosy dat z jedné databáze do druhé, hromadné importování/exportování dat do/z databáze, jejich využití můžeme nalézt také ve webových aplikacích, kde se používají pro přenos dat z backendu do frontendu a obráceně.

# 1. JSON SOUBORY

Kapitola je věnována JSON souborům, jejich stručnému popisu, jak se validují a obsahuje ukázky souborů s jejich popisem.

## 1.1. Popis JSON souborů

Zkratka JSON vychází ze slov Java Script Object Notation, v překladu zápis Java Scriptového objektu. JSON je totiž vytvořen na základě jazyka JavaScript. Jedná se o textový formát, který není závislý na jazyce. Jsou v něm využity konvence z již známých jazyků. Proto je pro programátory dobře čitelný, jednoduše se dá vytvářet i upravovat člověkem a zároveň i strojově. JSON je vhodný pro přenos dat i mezi různými platformami. [2], [3]

JSON je založen na dvou strukturách:

a) Struktura párů (jméno a obsah, resp. název a hodnota) - tato struktura je reprezentována jako jednoduchý objekt, hashovací tabulka nebo seznam s klíči.

b) Utříděný seznam hodnot - může být realizován jako pole, seznam, vektor nebo sekvence.

Všechny výše zmíněné datové struktury jsou univerzální a ve většině dnešních programovacích jazyků jsou podporovány, proto na nich byl založen tento formát, který je díky tomu velmi jednoduše přenositelný. [2]

Objekt ve formátu JSON obsahuje neuspořádanou množinu párů. Objekt je uvozen složenými závorkami, { levou na začátku, } pravou na konci. Mezi těmito závorkami pak nalezneme několik párů název/hodnota, ve formátu název: hodnota. Hodnota může být číslo, řetězec, znak, pole, v podstatě vše, co známe z programování. Číslo je pak zapisováno bez zvláštního zápisu. Řetězec nalezneme v uvozovkách, pole je uvozeno hranatými závorkami a prvky v něm jsou pak odděleny čárkou, stejně jako jednotlivé páry název/hodnota v celém objektu. [2]

## 1.2. Validace JSON souborů

JSON soubory mají předepsaná pravidla, která se musí při jejich tvorbě dodržovat. Tato pravidla lze kontrolovat několika způsoby, které budou zmíněny níže. V této práci jsem využil validaci dle EBNF gramatiky, s níž jsem byl seznámen již při vytváření seminární práce v předmětu Teorie jazyků, bylo tedy pro mě rychlejší pochopit a zrealizovat tuto možnost.

### 1.2.1. Validace pomocí JSON schéma

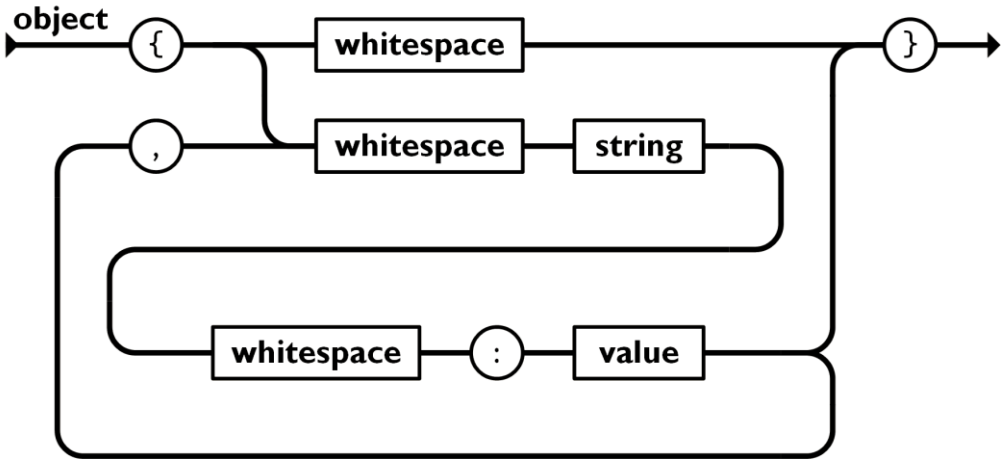
JSON schéma definuje formát JSONu pro popis struktury dat. Využívá klíčová slova k omezení instancí JSON objektů, případně k přidání dalších informací JSON objektu. Jiná klíčová slova se pak používají k uplatnění tvrzení a přidávání dalších informací na složité datové struktury JSON objektu. JSON schéma definuje, jak má JSON vypadat, způsob jak extrahovat data a jak se k jeho datům dostat. JSON schéma umožnuje definovat klíčová slova, která autorům slouží k popsání dat několika způsoby. Využívá klíčová slova k uplatnění omezení na instance či anotací na složitější objekt. Tato klíčová slova jsou pak uspořádána do slovníku. Soubor je pak validován dle těchto klíčových slov, je v něm vyhledáváno a určováno, zda hodnoty odpovídají klíčovým slovům, či nikoliv. [4]

### 1.2.2. Validace dle EBNF

EBNF (Extended Backuc Naur Form) se používá pro definici gramatiky programovacích jazyků, specifikuje daná pravidla, sekvence symbolů, klíčových slov, čísel, struktur a rezervovaných znaků. Pro JSON soubory jsou definované EBNF diagramy, které jednoduše popisují strukturu souboru a lze dle nich soubory převádět na JSON objekt. EBNF definuje pravidla, která se dodržují při rozebírání JSON souboru na jednotlivé objekty, hodnoty a pole. [5]

Každý diagram odpovídá jednomu pravidlu EBNF. Terminál[1] je znázorněn ve tvaru elipsy, neterminál[2] ve tvaru obdélníku. Směr vyhodnocování pravidla určují šipky, které jsou spojeny s terminály a neterminály. Opakování průchodu části pravidla je znázorněno smyčkou. Je zde umožňeno větvení, kde z terminálu může šipka jít ruzným směrem, k vyhovujícím neterminálu. [5]

[1] Symbol, který nelze měnit pomocí pravidla gramatiky
[2] Symbol, který lze měnit pomocí pravidla gramatiky




*Obrázek 1 - ukázka EBNF diagramu (JSON objekt) [2]*

## 1.3.Jednoduchá ukázka JSON souboru

```
{
  "cislo": 5,
  "retezec": "slovo",
  "pole_cisel": [
    5,
    1,
    3,
    15,
    33
  ],
  "pole_slov": [
    "slovo1",
    "slovo2"
],
  "boolean": true,
  "prazdno": null,
  "vnoreny_objekt": {
    "parametr1": 1,
    "parametr2": "nazev"
  }
}
```

*Ukázka zdrojového kódu 1 - jednoduchá ukázka JSON souboru*

Na této ukázce je vidět jednoduchá struktura JSON souboru. Jsou zde uvedeny příklady číselných hodnot, logických hodnot, null hodnot či řetězce. Dále je tam ukázka jednoduchého pole a jednoduchého vnořeného objektu. Na této ukázce s jednoduchými názvy hodnot je demonstrována základní struktura JSON souboru.

## 1.4. Složitější ukázka JSON souboru

```
{
  "jmeno": "Petr",
  "prijmeni": "Novák",
  "bydliste": {
    "ulice": "Karlova",
    "cp": 22,
    "psc": "55111",
    "mesto": "Petrov"
  },
  "deti": [
    {
      "jmeno": "Lenka",
      "prijmeni": "Nováková",
      "bydliste": {
        "ulice": "Karlova",
        "cp": 22,
        "psc": "55111",
        "mesto": "Petrov"
      }
    },
    {
      "jmeno": "Lukáš",
      "prijmeni": "Novák",
      "bydliste": {
        "ulice": "Karlova",
        "cp": 22,
        "psc": "55111",
        "mesto": "Petrov"
      }
    }
  ],
  "prezdivky": [
    "Novas",
    "Pěťas"
  ]
}
```

*Ukázka zdrojového kódu 2 - složitější ukázka JSON souboru*

Na této ukázce je již složitější struktura, která je demonstrována na zcela jednoduchém příkladu. Jako příklad jsem zde zvolil osobu. O každé osobě jsou známy nějaké informace, tyto údaje nejsou nikomu cizí, proto je jednoduché tento příklad pochopit. Na tomto objektu jsou demonstrovány struktury JSON souboru, jako jsou vnořený objekt, pole, objekt v poli. Vnořený objekt nám představuje bydliště. Je jasné, že bydliště každého z nás má několik parametrů, podle nichž jasně určíme přesnou polohu místa. Další část ukazuje pole s vnořenými objekty, které v tomto příkladu představují děti (dítě je osoba). Zde je vidět, že v ukázce máme dvě děti. Poslední částí je jednoduché pole, obsahující přezdívky dané osoby.

Tímto způsobem lze vytvářet JSON soubory. Tyto ukázky jsou vcelku logické a jednoduše uspořádané, ale pokud máme objekty složitější, určitě se nám pro čtení těchto souborů hodí vhodný editor.

# 2. ANALÝZA POŽADAVKŮ NÁSTROJE

Níže uvedené požadavky byly navrženy s ohledem na plánované nasazení aplikace a byly zpracovány v souladu s [6].

## 2.1. Funkční požadavky

- JSON editor bude umožňovat otevření jakéhokoliv JSON souboru
- JSON editor bude umožňovat snadnou editaci JSON souborů
- JSON editor bude validovat JSON soubory vůči EBNF diagramům
- JSON editor bude umožňovat uložení JSON souborů
- JSON editor bude umožňovat vyhledávání v JSON souboru
- Editor bude zobrazovat JSON soubor jako text
- Editor bude zobrazovat JSON soubor ve stromové struktuře

## 2.2. Nefunkční požadavky

- JSON editor bude napsán v jazyce Java
- Grafické uživatelské rozhraní editoru bude implementováno pomocí JavaFX
- Grafické uživatelské rozhraní bude obsahovat komponenty pro snadné čtení JSON souboru
- Kód v jazyce Java bude optimální a efektivní

# 3. DŮVODY PRO VYTVOŘENÍ NÁSTROJE

V této kapitole jsou důvody, které mě vedly k vytvoření nového nástroje pro editaci JSON souborů.

## 3.1. Velký potenciál využití JSON souborů v programování

JSON soubory hýbou světem programování. Jejich využití je čím dál větší, každý správný program by měl využívat pro přenos dat JSON soubory. Využívají se například pro přenos celých datových struktur, objektů, dají se využít při serializaci objektů. Serializace objektu je uložení aktuálního stavu celého objektu. Jako další příklad využití zde musím uvést import/export dat do/z databází, je to jednoduchý způsob, jak do databáze dostat dat. Spousta programátorů, a nejen programátorů, se bude s JSON soubory setkávat častěji a častěji, proto je vhodné mít efektivní nástroj pro editaci těchto souborů. [1]

## 3.2. Snadnější čtení a editace JSON souborů

Pro někoho nemusí být jednoduché se v JSON souboru zorientovat, proto by měl tento nástroj umožnit jednoduchou a rychlou editaci těchto souborů. Pro znalého člověka je JSON soubor jednoduché přečíst, ale když se k takovému souboru dostane někdo, kdo nemá ponětí, jaká je struktura tohoto souboru a jak se v něm data ukládají, není to už tak snadné. Takovému uživateli poté stačí spustit si editor vytvořený v této práci a v něm otevřít JSON soubor, který potřebuje přečíst a dekódovat, a hned jasně vidí, co potřebuje.

Pro větší přehlednost bylo v práci použito dvojí zobrazení souboru. Soubor je zobrazen ve stromové struktuře, kde je přehledně vidět, která hodnota patří ke které, a je zde dobře vidět strukturu samotného souboru. V druhém zobrazení je čistý text JSON souboru (tak jak vypadá soubor), v tomto režimu je možné upravovat soubor (editovat, přidávat, mazat hodnoty).

## 3.3. Kontrola syntaxe (validace)

JSON soubory mají daná pravidla, která se při ukládání do nich musí striktně dodržovat. Dodržování těchto pravidel hlídá i tento nástroj. Mezi tato pravidla patří způsob zápisu názvů, přiřazování hodnot, způsob zápisů hodnot (číslo, řetězec, logická hodnota, null, pole či vnořený objekt) a samotná celá struktura souboru (je uvozen složenými závorkami).

Nástroj kontroluje tato pravidla při provedení změny v něm a po vypršení daného časového úseku. Tato kontrola je realizována pomocí jiného vlákna, které je nezávislé na hlavním vláknu, v němž běží samotný hlavní program.

## 3.4. Více funkcí v jednom nástroji

Nástroj slučuje více funkcí do jednoho programu. Mezi tyto funkce patří validace v reálném čase, přívětivé zobrazení pro uživatele (snadné čtení a editace) a v neposlední řadě rychlá a efektivní práce s JSON soubory (rychlé a plynulé načtení/uložení souboru).

Současné nástroje sice více funkcí v jednom nástroje mají, ale vždy zvládají efektivně jen jednu z těchto funkcí. Například umí optimálně a vcelku rychle editovat JSON, ale neumí validaci v reálném čase; nebo naopak zvládají velmi dobře validaci dat v reálném čase, ale práce v tomto nástroji trvá dost dlouho. Proto jedním z hlavních cílů této práce je vytvořit nástroj, který by sdružil všechny tyto funkce do jedné, tedy mít nástroj, který bude rychlý a bude umožňovat validaci dat v reálném čase.

# 4. EXISTUJÍCÍ NÁSTROJE PRO PRÁCI S JSON SOUBORY

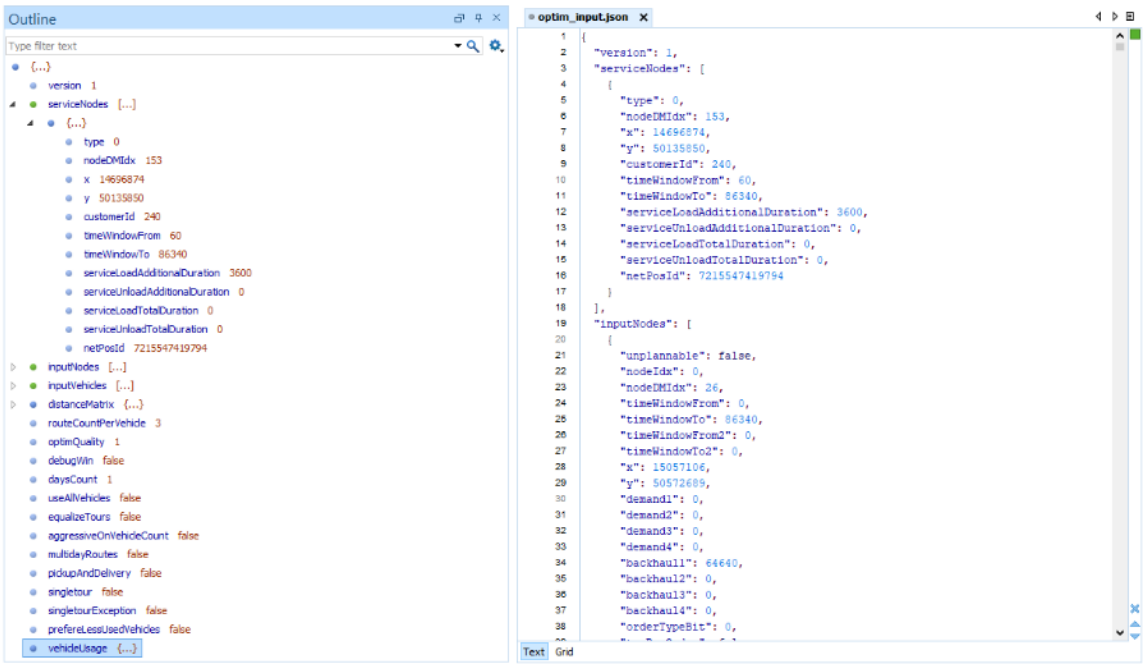
Tato kapitola se věnuje již existujícím vybraným nástrojům pro práci s JSON soubory. Je zde jednoduše popsána práce v nich, jejich funkce a zda je vhodné volit tento nástroj.

## 4.1.<oXygen/> XML editor

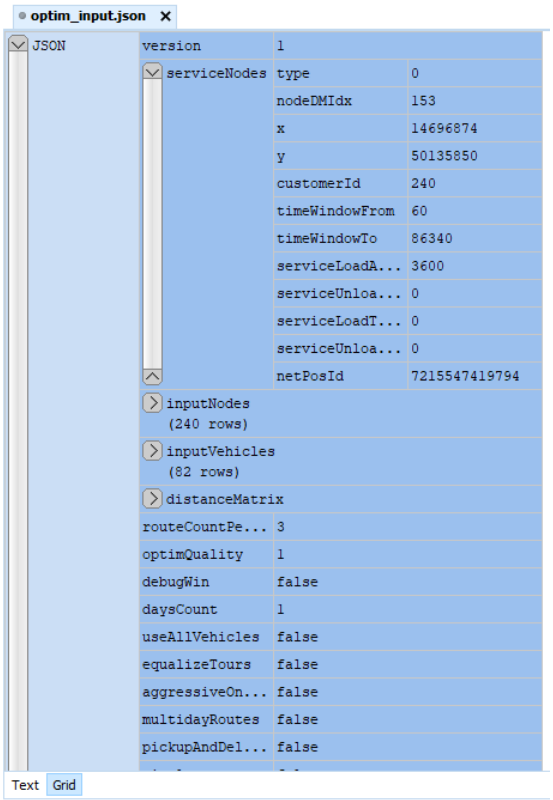
Nástroj <oXygen/> XML editor slouží primárně pro čtení/editaci XML souborů, ale hravě zvládá otevřít a editovat i JSON soubory. Už asi jen proto, že tento nástroj není primárně určen pro JSON soubory, nefunguje moc plynule a optimálně.

Z uživatelského pohledu je v tomto nástroji obrazovka rozdělena na dvě části. V první části je JSON soubor v textové podobě. V druhé části je JSON soubor rozdělený do stromové struktury. Dále lze textovou podobu přepnout do režimu „Grid“. Data se naskládají do tabulky, kde jsou přehledně vidět a lze je i editovat. Oproti tomu ve stromové struktuře data editovat nelze, ta slouží pouze ke zobrazení dat uložených v souboru. V textovém režimu lze soubor také editovat.

Jednou z největších nevýhod tohoto softwaru je, že není zdarma, pro jeho plné využívání jsme nuceni zaplatit nemalou částku. Nejnižší verze je určená pro studenty, učitele, knihovny, muzea, nemocnice a zdravotnická zařízení. Tato nejnižší verze stojí 99 amerických dolarů *(cena k 28. 2. 2021)* na rok. Nutno však podotknout, že s nástrojem <oXygen/> XML editor získáme za tuto částku i spoustu jiných nástrojů určených k práci s XML soubory. Další možností jak získat tento nástroj zdarma je verze Trial, kterou získáte po registraci na měsíc zdarma. [7]




*Obrázek 2 - snímek obrazovky z prostředí <oXygen/> XML editor, textový režim [zdroj vlastní]*



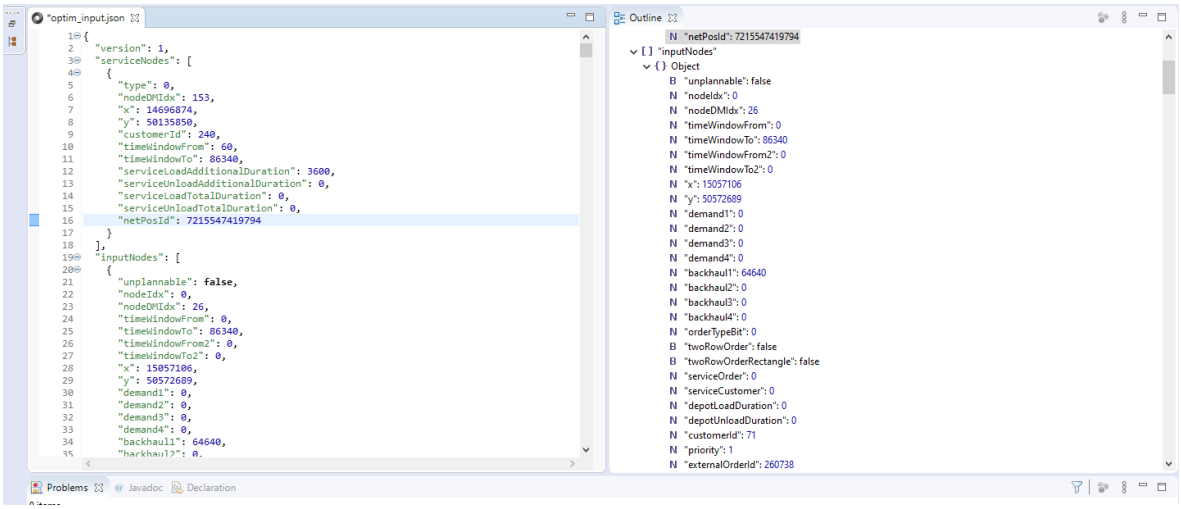

*Obrázek 3 - snímek obrazovky z prostředí <oXygen/> XML editor, režim grid [zdroj vlastní]*

## 4.2. Plugin pro Eclipse

Eclipse je vývojové prostředí sloužící primárně pro vývoj v jazyce Java, ovšem není to jediná funkcionalita tohoto programu. Mimo jazyka Java si poradí i s jinými programovacími jazyky. Eclipse umožňuje doinstalování pluginu, který zvládá otevření a editování JSON souborů. Zobrazení JSON souboru pomocí tohoto pluginu je velmi podobné jako u prvního zmiňovaného nástroje. Plugin je určen primárně pro JSON soubory. Editace je vcelku rychlá, ovšem pokud máme rozsáhlejší soubor, jeho načtení trvá poměrně dlouho a někdy se dokonce stane, že se objeví nápis „Neodpovídá“.

Obrazovka je v tomto nástroji rozdělena na dvě části. První část obsahuje JSON v textové podobě, kde lze soubor zároveň editovat. V druhé části je JSON v stromové struktuře, v této části nelze soubor editovat, lze si ho pouze prohlížet. [8]

Samotný plugin nekontroluje validitu hodnot. Ale pro kontrolu validity lze využít kupříkladu plugin Json Tools, který soubor formátuje, zvýrazňuje syntaxi a hlásí problémy v souboru. [9]



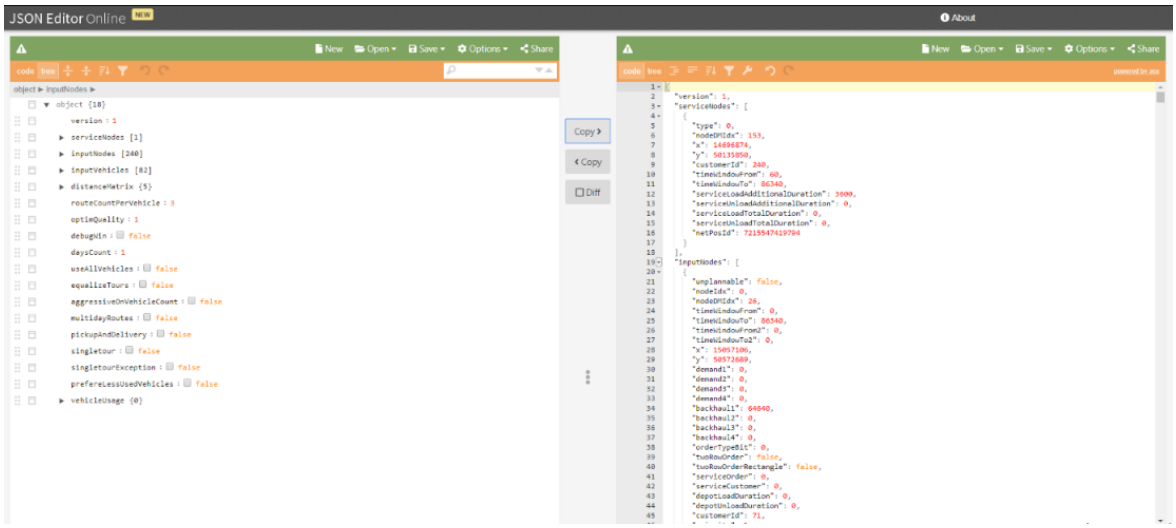

*Obrázek 4 - snímek obrazovky z programu Eclipse [zdroj vlastní]*

## 4.3.JSON editor online

JSON editor online je velmi jednoduchý nástroj. K jeho použití není třeba instalace. Stačí jednoduše otevřít danou webovou stránku a můžeme využívat funkce tohoto nástroje.

Nástroj má dva režimy zobrazení JSON souboru. Prvním je klasický textový režim, druhým je zobrazení ve stromové struktuře. V obou režimech je možná úprava souboru (přidávání, editace a mazání parametrů). Přidávání hodnot je pak možné pouze v textovém režimu. Soubor je kontrolován v reálném čase. Jestli se jedná o validaci syntaxe či validaci dle JSON schématu se nedá dle fungování přesně říct. Validace se provádí v reálném čase, jakmile do souboru něco dopíšeme, něco upravíme, tak se soubor kontroluje. Při objevení chyby se v textovém režimu u čísla řádku, kde je problém, objeví červený křížek. Při najetí kurzoru na tento křížek se pak zobrazí chyba, která nastala. Při přepínání mezi režimy probíhá taktéž validace, při objevení chyby se zobrazí chybová hláška.

Při otevírání rozsáhlejšího souboru sice chvíli čekáme, ale není nijak velká doba čekání, stejně tak při přepínání režimů u rozsáhlejšího souboru. Tento nástroj funguje vcelku jednoduše a efektivně. [10]




*Obrázek 5 - snímek obrazovky z webového JSON editoru [zdroj vlastní]*

## 4.4. Porovnání

| Nástroj | Rychlé | Zdarma | Validace | Efektivní | Více režimů zobrazení |
|---|---|---|---|---|---|
| **<oXygen/> XML editor** | ANO | NE | ANO | NE | ANO |
| **Plugin pro Eclipse** | ANO | ANO | NE | NE | ANO |
| **JSON Editor Online** | NE | ANO | ANO | ANO | ANO |

*Tabulka 1 - porovnání nástrojů pro editaci JSON souborů*

Na základě této tabulky je tedy vcelku zřejmé, že nástroj, jemuž je věnována tato práce, nebude mít jednoduché se prosadit. Má na trhu již obstojné konkurenty. Můj nástroj by se měl asi nejvíce podobat JSON Editoru Online s tím, že by měl pracovat rychleji. [11]

# 5. VYUŽITÉ TECHNOLOGIE PŘI VÝVOJI

Kapitola seznamuje s využitými technologiemi při vývoji nástroje pro editaci JSON souborů.

## 5.1. Java

Java je programovací jazyk, který je třídně založený, objektový a souběžný. Java je navržena tak, aby bylo možné dosáhnout plynulosti jazyka dostatečně jednoduše. Java je založena na C a C++, ale její organizace je odlišná. Některé aspekty jazyků C a C++ jsou vynechány a naopak jsou v jazyce Java zahrnuty aspekty z jiných jazyků. [12]

Jedná se o staticky silně napsaný programovací jazyk, což umožňuje rozlišení chyb kompilace:

- chyby, které se detekují při kompilaci,
- chyby, které se vyskytují za běhu programu.

Čas kompilace je závislý na předkládání programu do strojově nezávislé reprezentace bajt kódu. Postup při běhu programu:

1. načítání a propojování tříd potřebných k provedení programu,
2. volitelné generování strojového kódu,
3. dynamická optimalizace programu,
4. skutečné provedení programu.

Jazyk Java je vysoké úrovně, vzhledem k těmto detailům. Disponuje automatickou správou úložiště (garbage collector). V jazyce Java nenajdeme nebezpečné konstrukce, například při přístupu do pole je vždy kontrolován index. Nebezpečné konstrukce by způsobily nespecifické chování programu. [12]

Kompilace jazyka Java probíhá zpravidla do sady instrukcí bajtového kódu a do binárního formátu, který je definován v Java Virtual Machine Specification. [12]

Jednoduchá ukázka konstrukce jazyka:

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello World!!");
    }
}
```

*Ukázka zdrojového kódu 3 - jednoduchá ukázka kontrukce jazyka Java [13]*

## 5.2.JavaFX

JavaFX slouží k výoji bohatých klientských aplikací, které jsou konzistentní na různých platformách. Je sadou balíčků umožňující testovat, nasazovat, navrhovat a vytvářet aplikace. Knihovna JavaFX je vytvořena jako rozhraní Java API. Odděluje vývoj GUI od samotné implementace aplikace. Grafické části se může věnovat jeden programátor (jeden tým) a samotné implementaci druhý programátor (druhý tým). Lze stylovat vzhled GUI pomocí CSS. Samotný vzhled je pak možné vyvíjet ve skriptovacím jazyce FXML, ale lze využít možnosti vývoje vzhledu bez nutnosti psát kód, to nám umožní JavaFX Scene Builder. [14]

API JavaFX je dostupné jako integrované prostředí JRE i JDK. JDK lze spouštět na všech desktopových platformách, proto také aplikace napsané v JavaFX lze spouštět na všech těchto platformách. [14]

Některé zásadní vlastnosti:

- **FXML a nástroj pro tvorbu scén** – značkovací jazyk založený na XML, popřípadě JavaFX Scene Builder
- **Integrované ovládací prvky GUI a CSS** – umožňuje přidávat všechny ovládací prvky, které jsou nutné pro plnohodnoté využití aplikace; prvky lze stylovat pomocí CSS
- **Tisk API** - od verze Java SE 8 balíček JavaFX obsahují taktéž API umožňující tisk
- **Funkce 3D grafiky** – umožňuje vytvářet 3D tvary, kterými jsou například válec, kvádr nebo koule

### 5.2.1. Scene Builder

Scene Builder je nástroj, který slouží pro vytváření GUI v JavaFX. Scene Builder umožňuje GUI vytvářet bez nutnosti psát kód. Jen jednoduše umisťujeme prvky do prostoru. Můžeme prvkům přiřazovat, co se má stát po kliknutí na prvek, po přejetí myší, po změně textu, nebo třeba po stisku různých kláves či klávesových zkratek. V Scene Builderu si vytvoříme vzhled takový, jaký chceme, který je následně převeden do FXML automaticky. Scene Builder je součastí IDE IntelliJ IDEA, ve kterém jsem aplikaci vyvíjel. Případně si tento nástroj lze stáhnout samostatntě. [15]

### 5.2.2. RichTextFX

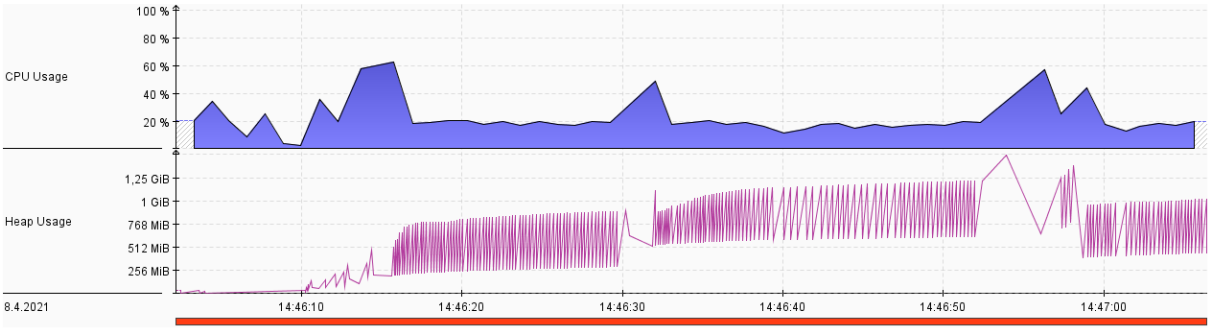
Aplikace měla problém při načítání obsáhlejších souborů, zasekávala se a nebyla plynulá. Proto jsem se rozhodl hledat, co ono zasekávání způsobuje. Zkoušel jsem aplikaci spustit bez GUI, kdy byl problém odstraněn, vše probíhalo v pořádku. Můj problém tím byl zúžen na GUI a prvky, ve kterém je JSON soubor zobrazován. Dále jsem tedy zkoušel upravit GUI následovně:

- Zobrazení JSONu pouze v TextArea – problém přetrvával
- Zobrazení JSONu pouze v TreeView – problém byl odstraněn

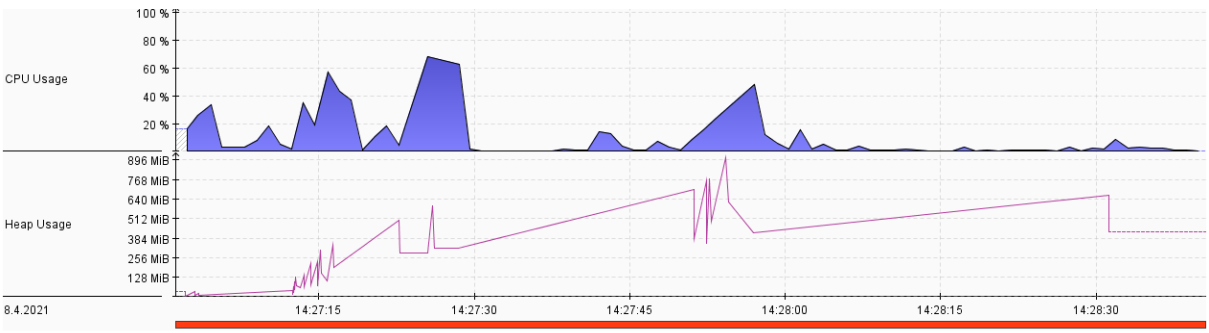
Na základě již zmiňovaného zkoušení jsem usoudil, že by mohl být problém s prvkem TextArea. Mým cílem bylo najít nějakou alternativu pro tento prvek. Narazil jsem na knihovnu RichTextFX, která slibovala paměťově efektivní prvky. Nabízela se tedy možnost otestovat rozdíly mezi RichTextFX a TextArea, což jsem také udělal.

Při zjišťování toho, co zpomaluje mou aplikaci, jsem využil Java Flight Recorder z produktu IntelliJ IDEA ve verzi 2021.1, který sleduje činnost aplikace. Data, která byla výsledkem analýzy Java Flight Recorderu, jsem otevřel v aplikaci JDK Mission Control, kde jsem si zobrazil přehledný graf využití paměti a procesoru.

Průběh záznamu: spuštění aplikace, otevření rozsáhlého JSON souboru s cca 23 000 řádků.




*Obrázek 6 - výsledky dat z Java Flight Recorderu při využití TextArea [zdroj vlastní]*




*Obrázek 7 - výsledky dat z Java Flight Recorderu při využití RichTextFX [zdroj vlastní]*

| | **JSON editor s využitím TextArea** | **JSON Editor s využitím RichTextFX** |
|---|---|---|
| **Maximální využití paměti** | 1, 49 GiB | 486 MiB |
| **Maximální zatížení procesoru** | 62, 9 % | 64,5 % |

*Tabulka 2 - porovnání výsledků dat z Java Flight Recorder*

Ze získaných hodnot jsem zjistil, že při použití prvku TextArea aplikace využívá velké množství operační paměti, což mě přesvědčilo, že bude vhodnější ve své práci využít zniňovanou knihovnu RichTextFX. Již na první pohled byla aplikace plynulejší a nesekala se. Další faktorem pro výběr této knihovny byla možnost přidání čísel řádků, zvýraznění aktuálního řádku, popřípadě přidání formátování (zvýraznění názvů, hodnot, atp.).

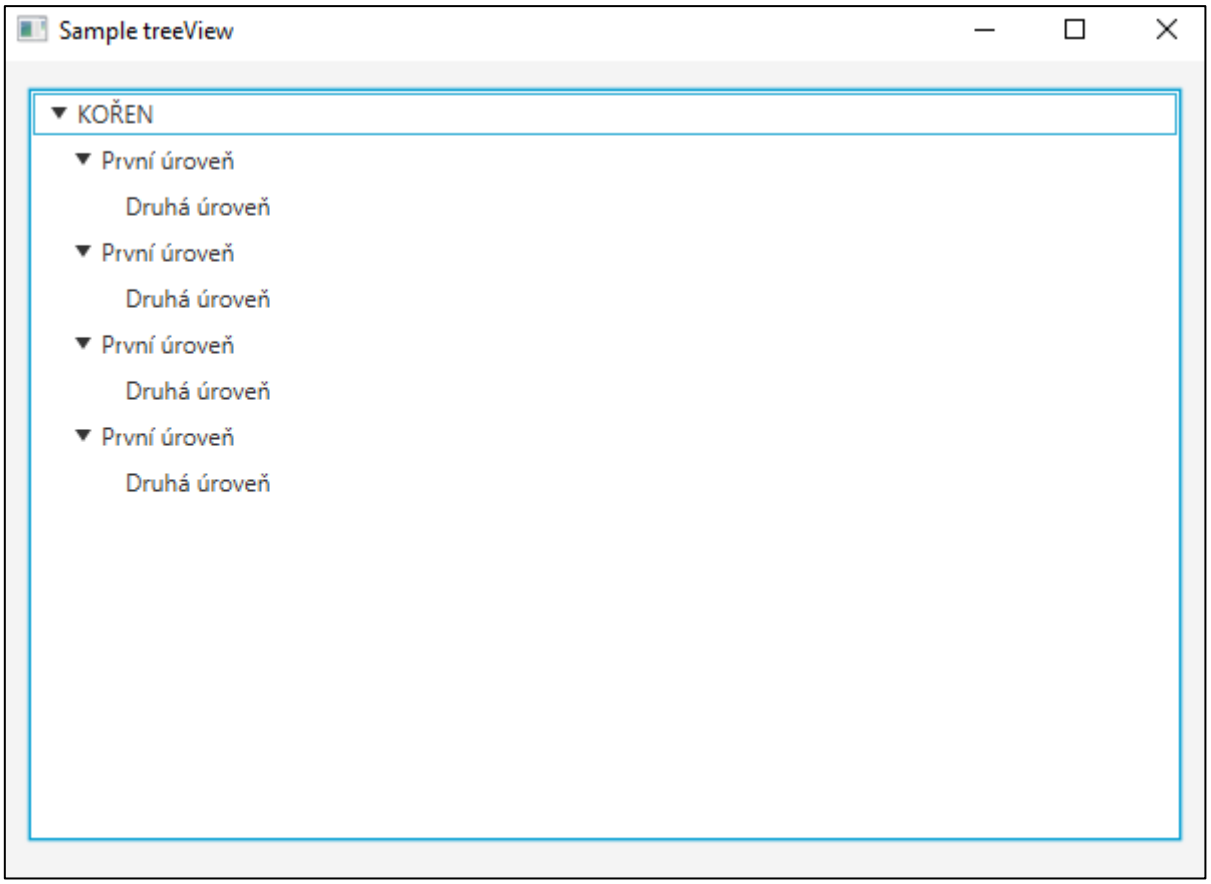
RichTextFX je knihovna obsahující paměťově efektivní textová pole pro JavaFX. Knihovnu RichTextFX lze využít pod licencí BSD 2-Clause, příslušný licenční soubor je součástí projektu. V knihovně nalezneme například tyto komponenty:

- **CodeArea** – využito v této práci, určeno pro zdrojové kódy
- **GenericStyledArea** – umožňuje vložení vlastních objektů do oblasti vedle textu

Díky RichTextFX je možné získat paměťově efektivní textové pole pro JavaFX, navíc tato komponenta umožňuje stylovat text pomocí CSS. Je určen pro editory zdrojových kódů se zvýrazňováním syntaxe. Umožňuje programátorům upravovat rozsahy textu, zobrazovat vlastní objekty a přepisovat výchozí chování. [16]

### 5.2.3. TreeView

TreeView poskytuje pohled na hierarchicky uspořádané struktury. V každém stromu je nejvyšší objekt zvaný kořen, který obsahuje další podřízené položky. JSON objekt má hierarchicky uspořádané hodnoty, proto bylo vhodné využít právě tuto komponentu pro jednu z možností zobrazení JSON souboru. [17]




*Obrázek 8 - ukázka TreeView [zdroj vlastní]*

*TreeView* má vždy přiřazen kořen neboli *RootItem*, do kterého jsou přidávány další položky. Úrovní může mít několik, v ukázce jsou dvě. Pomocí šipky na začátku řádku lze snadno rozbalit a zabalit jednotlivé úrovně. V *TreeView* je možné taktéž prvky editovat, editace je možné naprogramovat pomocí *TreeCellFactory*, která definuje akce před začátkem editace a po ukončení editace. V následující ukázce kódu je příklad inicializace *TreeView*.

```
public class Controller {
    @FXML
    TreeView<String> treeView;
    @FXML
    private void initialize(){
        TreeItem<String> root = new TreeItem<>("KOŘEN");
        TreeItem<String> prvniUroven = new TreeItem<> ("První úroveň");
        TreeItem<String> druhaUroven = new TreeItem<> ("Druhá úroveň");
        prvniUroven.getChildren().add(druhaUroven);
        root.getChildren().add(prvniUroven);
        root.getChildren().add(prvniUroven);
        root.getChildren().add(prvniUroven);
        root.getChildren().add(prvniUroven);
        treeView.setRoot(root);
    }
}
```

*Ukázka zdrojového kódu 4 - inicializace ukázky TreeView*

## 5.3. Git

Před tím, než bude řečeno, co je to Git, je nutné definovat pojem verzování.

Verzování je proces přiřazování jedinečných názvů verzí nebo jedinečných čísel verzí jedinečným stavům počítačového softwaru. Převzato z IT-slovnik.cz. [18]

Git je systém pro správů verzí, neboli tedy verzovací systém. Je nejpoužívanější a velmi moderní systém používaný v dnešním světě. Jedná se o aktivně udržovaný open source projekt, který původně vytvořil slavný tvůrce jádra operačního systému Linux Torvalds. Při vývoji Gitu byl kladen důraz na flexibilitu, výkon a zabezpečení. [19]

S mnoha alternativami jsou surové výkonostní charakteristiky Gitu velmi silné. Umožňuje zavádění nových změn, porovnávání minulých verzí, slučování či větvění, a to vše s optimalizací na výkon. Git uvnitř využívá algoritmy, které využivají hlubokých znalostí o zdrojových kódech a o tom, jaké jsou přístupové vzory a jak se obvykle zdrojové kódy mění. Git se zaměřuje přímo na obsah souboru, soubory bývají často rozděleny, přeskupeny nebo přejmenovány. Jako formát ukládání objektů souborů Git využívá kombinaci ukládání rozdílu v obsahu, kompresi a explicitně ukládá objekty metadat verze a obsah adresáře. [19]

Některé další verzovací systémy nemají ochranu proti utajeným změnám, což může způsobovat závažné chyby v zabezpečení informací v organizaci. V úložišti Git jsou obsahy souborů, verze, značky, skutečné vztahy mezi soubory a adresáři a revize těchto objektů zabezpečeny kryptografickým hashovacím algoritmem SHA1. Toto hashování zajišťuje plnou sledovanost historie a chrání před nechtěnými změnami. [19]

Značná část absolventů vysokých škol i velké množství vývojářů již zkušenosti s Gitem má. Kromě toho, že značná část programátorů nástroj Git zná, se zde dá zmínit jedna další výhoda. Existuje spousta nástrojů třetích stran, které mají integrovaný Git. Mezi ně například patří samotné IDE. Na seznemu znalostí vývojáře či programátora by Git neměl chybět. [19]

Git je navržen, aby podporoval označování a větvení, jako jsou například operace, které mají vliv na značky a větvení. Toto vše je součástí historie změn. Tuto úroveň sledování nemají všechny systémy pro správu verzí. [19]

Git má funkce, které velké množství týmů a vývojářů potřebuje. Při porovnávání různých alternativ bylo zjištěno nejedním týmem, že je Git pro ně velmi příznivý. [19]

Při vývoji editoru jsem využíval server GitHub.com, mimo tento server existijí také tyto servery: GitLab.com nebo BitBucket.org.

# 6. VYUŽITÉ DATOVÉ STRUKTURY PŘI VÝVOJI

V této práci bylo nutné uchovávat samotné hodnoty JSON souborů a pomocných objektů při rozebírání JSON souboru. K tomuto účelu posloužily datové struktury. Konkrétní datové struktury, které byly využity, jsou popsány níže.

Všechny příklady jsou uváděny v jazyce Java

## 6.1. Pole

Pole je datová struktura, kterou nalezneme téměř ve všech programovacích jazycích. Vzniklo z důvodu touhy využívat v programování nejen čísla, ale i vektory. Jedná se o statickou datovou strukturu, do které můžeme uložit téměř cokoliv (čísla, řetězce, znaky, …). Statická znamená, že má předem definovanou velikost. Jako příklady využití pole zde lze uvést datový typ String, nebo celý adresový prostor. Proto lze String i adresový prostor jednoduše indexovat. Pole by se dalo popsat jednoduše jako spousta krabiček uložených vedle sebe. Tyto „krabičky" jsou očíslovány přirozenými čísly. Bývá zvykem začít číslování od 0, původně to bylo z důvodu šetření paměti, v dnešní době 2 bajty moc velká úspora není, ale zpřehledňuje to kód. [20]

| **Hodnota** | 55 | 33 | 20 | 105 | 653 | 200 |
|---|---|---|---|---|---|---|
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 |

*Tabulka 3 - ukázka pole*

Pole v této práci bylo využito pouze na jednom místě. Využívá se při tvorbě lexémů. Nejdříve je celý soubor převeden na jeden textový řetězec a tento řetězec následně na pole znaků, které jsou samostatně analyzovány.

### 6.1.1. Deklarace a inicializace

Jak již bylo zmíněno, pole je statická datová struktura. Při inicializaci tedy vlastně pevně určujeme počet prvků, kolik v něm může být uloženo. Dále při inicializaci určujeme, jakého datového typu budou prvky uložené v daném poli. [20]

Příklady:

```
//pole celých čísel o velikosti 20
int [] poleCelychCisel = new int [20];
//pole řetězců o velikosti 10
String [] poleRetezcu = new String [10];
```

*Ukázka zdrojového kódu 5 - příklady inicializace pole*

### 6.1.2. Příklady prováděných operací:

- výpis pole

```
for(int i = 0; i<pole.lenght; i++){
        System.out.println(pole[i]);
}
```

*Ukázka zdrojového kódu 6 - příklad výpisu pole*

- vložení/úprava prvku

```
poleCisel[5] = 10;
poleRetezcu[2] = "retezec";
```

*Ukázka zdrojového kódu 7 - příklad vložení prvků do pole*

- smazání prvku

```
for(int i = indexMazanehoPrvku; i<pole.lenght-1; i++){
        pole[i] = pole[i+1];
}
```

*Ukázka zdrojového kódu 8 - příklad smazání prvku v poli*

- nalezení prvku

```
int hledanyPrvek = nejakyPrvek;
int poziceHledaneho = -1;
for(int i 0; i<pole.lenght; i++){
    if(pole[i] == hledanyPrvek){
        System.out.println(„Hledany prvek " + pole[i] + „ byl nalezen na
pozici " + i);
        poziceHledaneho = i;
        break;
    }
}
if(poziceHledaneho == -1){
    Systém.out.println(„Prvek "+ hledanyPrvek +" nebyl nalezen.");
}
```

*Ukázka zdrojového kódu 9 - příklad hledání prvku v poli*

- setřídění prvků (BublleSort) – od nejnižšího po nejvyšší

```
for(int i 0; i<pole.lenght - 1; i++){
        for(int j = 0; j<pole.length - 1; j++){
            if(pole[j] > pole[j+1]){
                int temp = pole[j];
                pole[j] = pole[j+1];
                pole[j+1] = temp;
            }
        }
}
```

*Ukázka zdrojového kódu* 10 *- příklad seřazení prvků v poli*

## 6.2. Seznam na poli

Seznam je dynamickou datovou strukturou. Dynamika znamená, že můžeme datovou strukturu měnit i po spuštění bez problému. Lze bez problému přidávat, ubírat, zjišťovat velikost, či se ptát, který prvek obsahuje. Jedná se v podstatě o chytřejší dynamické pole. Ve vnitřní struktuře je implementován pomocí pole, tzn., že po přidání prvku, který již překračuje kapacitu vnitřního pole, je pole předimenzováno (vytvoří se nové pole o větší velikosti a jsou do něj uloženy prvku z původního pole). Seznam na poli v podstatě vznikl pro usnadnění práce programátorů. Při využívání pole bylo nutné neustále pole zvětšovat, kopírovat prvky atp., abychom získali pole dynamické. [21]

| **Hodnota** | 55 | 33 | 20 | 105 | 0 | 0 |
|---|---|---|---|---|---|---|
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 |

*Tabulka 4 - ukázka seznam na poli*

Seznam v ukázce má 4 prvky, ale pole, které je umístěné uvnitř seznamu má prvků 6. Dva prvky nejsou využívány a nejsou viditelné zvenku.

Nejčastější implementací seznamu na poli v jazyce Java je třída ArrayList, která implementuje rozhraní List. Mimo tuto implementaci je možné seznam realizovat také jinak. Lze ho realizovat jako spojový seznam, který má dvě varianty, jednosměrný a obousměrný. Princip spočívá v tom, že prvky tohoto seznamu pak ukazují na další prvek v pořadí, popřípadě na předchozí prvek ze seznamu (pro obousměrný). [22]

| Operace | Seznam na poli | Spojový seznam |
|---|---|---|
| **Vkládání** | O (n) | O (1) |
| **Mazání** | O (n)* | O (1) |
| **Získání (náhodný přístup)** | O (1) | O (n) |
| **Vyhledávání** | O (1) | O (n) |

*Tabulka 5 – porovnání vybraných operací z pohledu časové složitosti*

* bylo by O (1), ale prvky by byly zpřeházené, pokud tedy má být zachováno pořadí prvků, je složitost O (n)

V této práci jsem zvolil seznam na poli, pro potřebné operace je méně časově složitý než spojový seznam, seznam na poli byl zvolen z důvodu nižší časové složitosti náhodného přístupu a vyhledávání. [23]

Seznam v práci slouží k uchování hodnot JSON objektu. Nabízela se zde možnost využít Hashovací tabulku, ale bylo by to nevhodné z důvodu řazení prvků dle hashe. V tomto případě bylo potřeba, aby byly prvky řazeny ve stejném pořadí, v jakém byly vloženy.

### 6.2.1. Deklarace a inicializace

```
//obecně
List<DatovyTyp> list = new ArrayList<>();
//celá čísla
List<Integer> listCelychCisel = new ArrayList<>();
//řetězce
List<String> listRetezcu = new ArrayList<>();
```

*Ukázka zdrojového kódu 11 - příklad inicializace seznamu s polem*

### 6.2.2. Příklady prováděných operací

- Přidání prvku do seznamu

```
//obecně
list.add(pridavanyPrvek);
//celá čísla
listCelychCisel.add(1);
//řetězce
listRetezcu.add("retezec");
```

*Ukázka zdrojového kódu 12 - příklad přidání prvku do seznamu s polem*

- Přidání více prvku do seznamu

```
//obecně
list.addAll(pridavanyPrvek1,pridavanyPrvek2);
//celá čísla
listCelychCisel.addAll(1,5);
//řetězce
listRetezcu.addAll("retezec1", "retezec2");
```

*Ukázka zdrojového kódu 13 - příklad přidání více prvků do seznamu s polem*

- Vymazání všech prvků

```
//obecně
list.clear();
//celá čísla
listCelychCisel.clear();
//řetězce
listRetezcu.clear();
```

*Ukázka zdrojového kódu 14 - příklad vymazání všech prvků v seznamu s polem*

- Zjištění, zda obsahuje daný prvek

```
//obecně
list.contains(hledanyPrvek);
//celá čísla
listCelychCisel.contains(1);
//řetězce
listRetezcu.contains( "retezec");
```

*Ukázka zdrojového kódu 15 - příklad zjišťování, zda seznam na poli obsahuje prvek*

- Získání prvku na daném indexu

```
//obecně – prvek na prvním místě
DatovyTyp prvek = list.get(0);
//celá čísla – prvek na druhém místě
int celeCislo = listCelychCisel.get(1);
//řetězce – prvek na třetím místě
String retezec = listRetezcu.get(2);
```

*Ukázka zdrojového kódu 16 - příklad získání prvku na indexu v seznamu s polem*

[21]

## 6.3. Fronta

Jedná se o datovou strukturu, do které můžeme ukládat prvky a následně je vybírat ve stejném pořadí, v jakém jsme je tam vložili. Prvek, který byl přidán nejdříve, je odebrán nejdříve (FIFO – First in First Out). S frontou se člověk setkává v běžném životě, například fronta u pokladny v obchodě. Tato datová struktura kopíruje tu frontu z běžného lidského života. [24]

Jak tedy jednoduše popsat fungování fronty? Vezmeme několik prvků (čísel), které uložíme do této datové struktury v tomto pořadí (55, 21, 0, 3). Následně prvky chceme vybrat, dostáváme je v tomto pořadí (55, 21, 0, 3).

Na této datové struktuře se realizují dvě operace: enqueue (add – vložení prvku) a dequeue (get – vybrání prvku). [24]

Pokud je fronta správně implementována, její operace pak mají časovou náročnost O (1). Fronta může být implementována na poli, kde se definují hlava (HEAD) a ocas (TAIL), neboli začátek a konec fronty. Prvky se v poli musejí přesouvat. V práci byla využita fronta, která je součástí knihovny *Util* Java 8. Tato fronta je implementována pomocí spojového seznamu. [25]

Speciální případ fronty je prioritní fronta, ve které můžeme prvkům přiřadit prioritu a dle této priority se pak řadí do fronty, v praxi to tedy znamená, že se prvky ve frontě mohou předbíhat. [22]

Fronta je v jazyce Java implementována pomocí spojového seznamu. Spojový seznam je speciální druh seznamu, který obsahuje prvky ukazující na další prvek v seznamu, popřípadě i na předchozí prvek v seznamu (to je pak obousměrný spojový seznam). Pokud chceme vybrat prvek z tohoto seznamu, je nutné projít všechny prvky od prvního až k požadovanému indexu. Tato datová strukutra je vhodná v případě, kdy nepotřebujeme prvky indexovat. [22]

V mé práci jsem zvolil frontu implementovanou na spojovém seznamu, která je využívána při načítání JSON souboru a jeho rozkladu na lexémy a tokeny. Výsledné tokeny jsou uspořádány ve frontě, ze které jsou odebírány pro vytvoření samotného JSON objektu. Fronta se zde hodila, protože soubor je načítaný od začátku, jsou z něj vytvořeny tokeny a tyto tokeny je třeba následně získat ve stejném pořadí, v jakém jsou v souboru. Proč jsem tedy nevyužil prostý spojový seznam? Protože fronta nabízí pouze 3 jednoduché metody (vlož, dej první ve frontě a odeber a dej první z fronty), které byly pro tento případ dostačující.

V následující části podkapitoly se dozvíte, jak se fronta deklaruje, inicializuje a jak se provádějí dané operace na ní.

### 6.3.1. Deklarace a inicializace

```
//obecně
Queue<DatovyTyp> fronta = new LinkedList<>();
//celá čísla
Queue<Integer> frontaCelychCisel = new LinkedList<>();
//řetězce
Queue<String> frontaRetezcu = new LinkedList<>();
```

*Ukázka zdrojového kódu 17 - deklarace a inicializace fronty*

### 6.3.2. Příklady prováděných operací

- Vložení prvku

```
//obecně
fronta.add(prvek);
//celá čísla
frontaCelychCisel.add(55);
//řetězce
frontaRetezcu.add("retezec1");
```

*Ukázka zdrojového kódu* 18 *- přidání prvku do fronty*

- Vybrání prvku

```
//obecně
//zjištění, který prvek je další
DatovyTyp prvek = fronta.peek();
//odebrání dalšího prvku
DatovyTyp prvek = fronta.poll();
//celá čísla
//zjištění, který prvek je další
int celeCislo = frontaCelychCisel.peek();
//odebrání dalšího prvku
int celeCislo = frontaCelychCisel.poll();
//řetězce
//zjištění, který prvek je další
String retezec = frontaRetezcu.peek();
//odebrání dalšího prvku
String retezec = frontaRetezcu.poll();
```

*Ukázka zdrojového kódu 19 - vybrání prvku z fronty*

[24]

# 7. POPIS NÁSTROJE A IMPLEMENTACE

Kapitola je věnována popisu nástroje a samotné implementaci nástroje, který je předmětem této bakalářské práce.

Ukázky kódu jsou uváděny v jazyce Java

## 7.1. Popis nástroje

Nástroj, který je předmětem této bakalářské práce, slouží k otevírání, editaci, validaci, snadnému zobrazení a ukládání JSON souborů. Nástroj je napsán v jazyze Java, grafické uživatelské rozhraní v JavaFX. Nástroj provádí validaci JSON souborů, vždy při otevírání, ukládání, případně pak při jakékoliv změně v obsahu. Ovládání nástroje je velmi jednoduché a intuitivní, stačí nástroj spustit, pomocí menu otevřít vybraný JSON soubor a pak už jen provádět chtěné změny v souboru pomocí textového pole, ve kterém je zobrazen. Pro větší přehlednost je taktéž zobrazen JSON soubor v komponentě *TreeView*, kde je vidět jeho struktura. Následně lze upravený soubor uložit do původního souboru, či ho uložit do úplně nového souboru.

## 7.2. Struktura projektu

Projekt je rozdělen do několika balíčků, balíčky jsou pojmenovány dle toho, k čemu slouží třídy uvnitř něj. Projekt je rozdělen do těchto balíčků:

- **converting** – převod JSON objektu na text,
- **enums** – výčtový typ pro typy tokenů,
- **exceptions** – výjimky,
- **gui** – třídy realizující grafické uživatelské rozhraní,
- **lexing** – třídy realizující převod textu na lexémy,
- **parsing** – převod lexémů na tokeny a vytváření JSON objektu a jeho hodnot,
- **reading** – převod JSON souboru na text,
- **tokens** – třídy reprezentující tokeny,
- **validting** – třídy realizující validaci v reaálném čase,
- **values** – třídy reprezentující hodnoty JSON objektu,

- **writing** – ukládání JSON objektu do souboru.

## 7.3. Funkce nástroje a jejich realizace

### 7.3.1. Čtení a analyzování JSON souboru

Při čtení a analyzování JSON souborů jsem využil znalosti z předmětu Teorie jazyků a zkušeností z psaní semestrální práce v tomto předmětu. V předmětu Teorie jazyků jsme se učili číst a analyzovat soubor dle daných předepsaných pravidel, která byla předem určena. Taktéž JSON soubory mají jasně předepsaná pravidla, která musí být při jejich vytváření dodržována. Na základě těchto znalostí jsem využil EBNF diagram, který mi velmi usnadnil práci. Soubor vstupuje do aplikace, postupně prochází jednotlivými metodami.

- Převod souboru na text

```
public String readJSONFromFile(File file) throws IOException {
    BufferedReader bufferedReader = new BufferedReader(
                new FileReader(file));
    String line = bufferedReader.readLine();
    StringBuilder stringBuilder = new StringBuilder();
    while (line != null) {
        stringBuilder.append(line).append("\n");
        line = bufferedReader.readLine();
    }
    bufferedReader.close();
    return stringBuilder.toString();
}
```

*Ukázka zdrojového kódu 20 - převod souboru na textový řetězec*

V této metodě dochází k převodu JSON souboru na textový řetězec. Do metody vstupuje již otevřený JSON soubor, který je načten a následně převeden na textový řetězec, který tato metoda vrací.

- Vytvoření lexémů

Lexémem se rozumí sekvence znaků odpovídající daným pravidlům předepsaných tokenů. V tomto případě se jedná o rozdělení na klíčové znaky a slova. [26]

```
public List<Lexem> createLexemsFromString(String jsonAsString) {
    List<Lexem> lexems = new ArrayList<>();
    String buffer = "";
    boolean isString = false;
    int row = 0;
    int column = 0;
    for (char ch : jsonAsString.toCharArray()) {
        if (isNotImportantChar(ch)) {
            row++;
            column = 0;
            continue;
        }
        if (isString && !isQuontatitonMarksChar(ch)) {
            buffer += ch;
            column++;
            continue;
        }
        if (isWhiteSpaceChar(ch)) {
            addLexem(buffer, row, column, lexems);
            buffer = "";
            continue;
        }
        if (isSeparatingChar(ch)) {
            addLexem(buffer, row, column, lexems);
            buffer = "";
            addLexem(String.valueOf(ch), row, column, lexems);
            continue;
        }
        if (isQuontatitonMarksChar(ch)) {
            //je to konec retezce - prida se i retezec
            addLexem(buffer, row, column, lexems);
            buffer = "";
            //pridaji se uvozovky
            addLexem(String.valueOf(ch), row, column, lexems);
            isString = !isString;
            continue;
        }
        buffer += ch;
        column++;
    }
    addLexem(buffer, row, column, lexems);
    return lexems;
}
```

*Ukázka zdrojového kódu 21 - vytvoření lexémů*

Jedná se o metodu, do které vstupuje JSON soubor jako text a je rozdělen na jednotlivé lexémy, které reprezentují již oddělení znaky, řetězce, klíčová slova nebo čísla. Následně pak vrací tato metoda seznam lexémů.

- Převod lexémů na tokeny

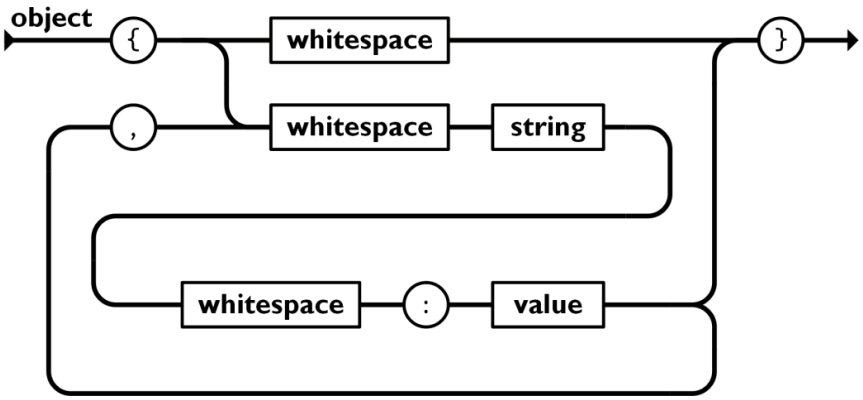
Token představuje objekt nějakého lexému, jednoduše token je lexém převedený na konkrétní hodnotu (číslo, slovo, speciální znak, klíčové slovo, atp.) [26]

```
//Metoda vytvářející ze seznamu lexému frontu tokenů určených k další
//analýze
public Queue<Token> createTokensFromLexems(List<Lexem> lexems) {
    Queue<Token> tokens = new LinkedList<>();
    for (Lexem lexem : lexems) {
        tokens.add(createToken(lexem));
    }
    return tokens;
}

//Metoda pro vytvoření tokenu z lexému
public Token createToken(Lexem lexem) {
    TokenType typeOfNewToken = getTokenTypeFromLexem(lexem);
    return new Token(typeOfNewToken, lexem.getValue(), lexem.getRow(),
lexem.getColumn());
}
```

*Ukázka zdrojového kódu 22 - převod lexémů na tokeny*

Metoda slouží k vytvoření tokenů ze seznamu lexémů. Seznam lexému vstupuje do metody jako parametr. Následně cyklem projdou všechny lexémy, které jsou přidány do fronty tokenů díky pomocné metodě, jež nalezne typ tokenu a vytvoří novou instanci třídy *Token* s danými parametry. Hlavní metoda poté vrací celou frontu tokenů. Je zde využita fronta z důvodu snadnějšího využití při analýze tokenů, ze kterých je následně vytvářen samotný JSON objekt.

- Uložení tokenů do struktury JSON objektu



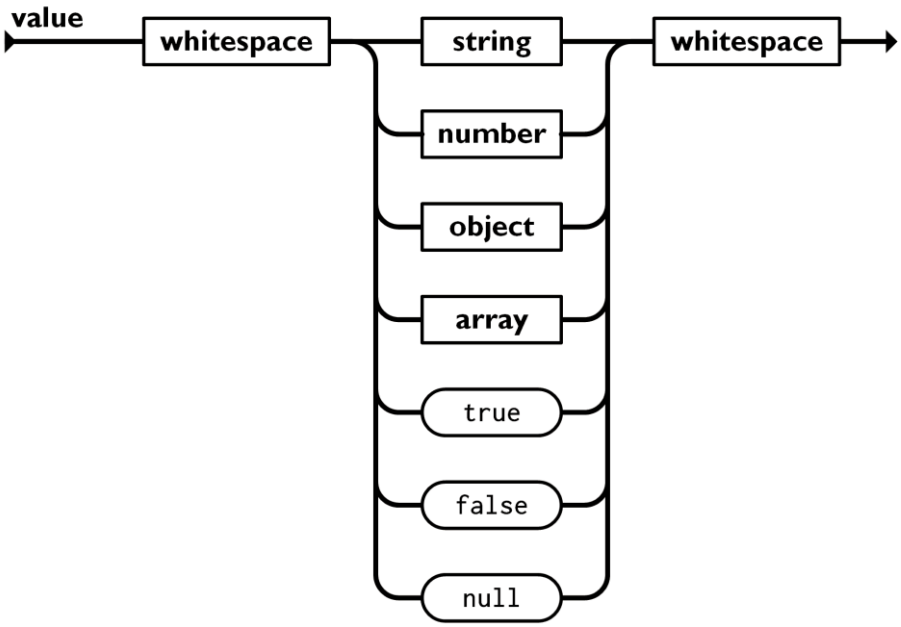

*Obrázek 9 - EBNF JSON object [2]*

Obrázek představuje jednoduchý nákres, jak probíhá načtení JSON objektu. Každý objekt začíná složenou závorkou, za kterou následuje bílý znak, název hodnoty, dvojtečka, která značí přiřazení hodnoty a poté buď čárka, po které následuje další hodnota, popřípadě ukončovací složená závorka, která značí konec JSON objektu.

```
//Metoda pro vytvoření JSON objektu z tokenů
public JSONObject parseJSObject(Queue<Token> tokens, String objectName)
throws JSONErrorException {
    if (tokens.isEmpty()) {
        return new JSONObject();
    }
    if (verifyTokensQueue(tokens) &&
!tokens.peek().getTypeOfToken().equals(TokenType.CURLY_BRACKET_START)) {
        throw new JSONErrorException("Curly bracket START expected at ("+
        tokens.peek().getRow() + ", " + tokens.peek().getColumn() + ")");
    }
    tokens.poll();
    JSONObject object;
    if (objectName != null && objectName.length() > 0) {
        object = new JSONObject(objectName);
    } else {
        object = new JSONObject();
    }
    parseJSObjectValues(tokens, object);
    //ocekava se ukoncovaci zavorka
    if (!verifyTokensQueue(tokens) &&
!tokens.peek().getTypeOfToken().equals(TokenType.CURLY_BRACKET_END)) {
        throw new JSONErrorException("Curly bracket END expected at (" +
        tokens.peek().getRow() + ", " + tokens.peek().getColumn() + ")");
    }
    tokens.poll();
    return object;
}
```

*Ukázka zdrojového kódu 23 - převod tokenů do struktury JSON objektu*

Tato metoda jen „kopíruje“ předchozí nákres. A takto je v projektu realizováno načtení JSON objektu.



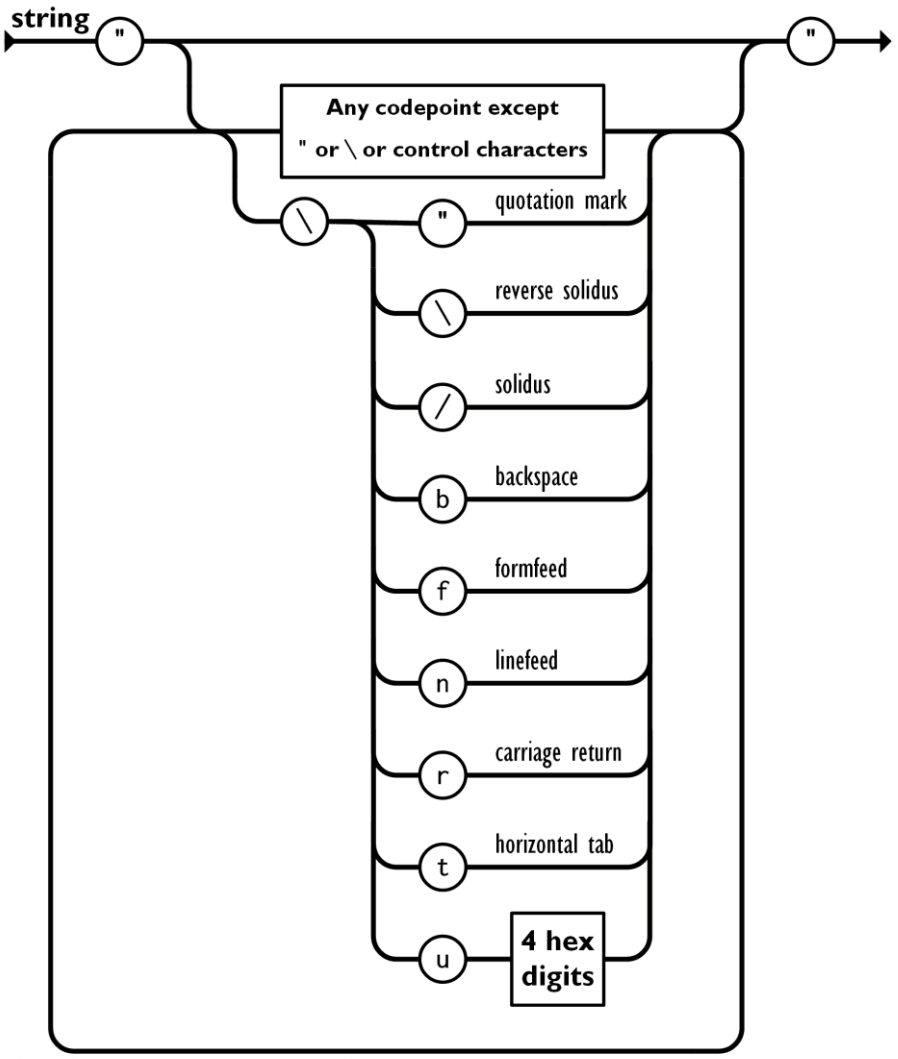

*Obrázek 10 - EBNF JSON value [2]*

EBNF diagram popisující načtení jednotlivých hodnot JSON souboru, hodnoty můžeme mít následující: řetězec, číslo, objekt, pole, logické hodnoty nebo null.

```
private void parseJSObjectValues(Queue<Token> tokens, JSONObject object)
throws JSONErrorException {
    //resi, zda objekt neni prazdny
    if
(tokens.peek().getTypeOfToken().equals(TokenType.CURLY_BRACKET_END)) {
        return;
    }
    String name = parseStringValue(tokens);
    if (!verifyTokensQueue(tokens) &&
        !tokens.peek().getTypeOfToken().equals(TokenType.COLON)) {
        throw new JSONErrorException("Colon expected at (" +
        tokens.peek().getRow() + ", " + tokens.peek().getColumn() + ")");
    }
    tokens.poll();
    object.addValue(parseValue(tokens, name));
    //dokud je tam carka (oddelenni hodnot, tak cti dalsi hodnoty)
    while (verifyTokensQueue(tokens) &&
        tokens.peek().getTypeOfToken().equals(TokenType.COMMA)) {
        tokens.poll();
        //nacteni nazvu
        name = parseStringValue(tokens);
        //ocekava se :
        if (!verifyTokensQueue(tokens) ||
!tokens.peek().getTypeOfToken().equals(TokenType.COLON)) {
            throw new JSONErrorException("Colon expected at (" +
        tokens.peek().getRow() + ", " + tokens.peek().getColumn() + ")");
        }
        tokens.poll();
        //pokud je slozena zavorka, cti vnoreny objekt
        if (verifyTokensQueue(tokens) &&
tokens.peek().getTypeOfToken().equals(TokenType.CURLY_BRACKET_START)) {
            object.addValue(parseJSObject(tokens, name));
            continue;
        }
        object.addValue(parseValue(tokens, name));
    }
}
```

*Ukázka zdrojového kódu 24 - vytvoření hodnot JSON objektu*

Ukázka realizuje předchozí diagram, jako vstupní parametry do metody vstupují fronta tokenů a objekt, do kterého se hodnoty přidávají. Vždy je načten název hodnoty, dále má následovat dvojtečka a samotné načtení hodnoty. V cyklu jsou pak načítány hodnoty, dokud není nalezen oddělovací znak, tedy čárka.



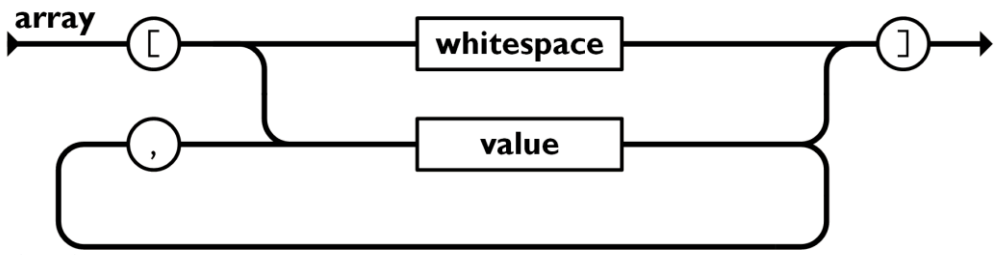

*Obrázek 11 - EBNF JSON String value [2]*

Diagram jednoduše popisuje, jaká pravidla platí pro řetězce uložené v JSON objektu. Řetězec musí být uvozen z obou stran uvozovkami a uvnitř těchto uvozovek pak může být defacto cokoliv.

```
private String parseStringValue(Queue<Token> tokens) throws
JSONErrorException {
    if (!verifyTokensQueue(tokens) ||
!tokens.peek().getTypeOfToken().equals(TokenType.QUONTATION_MARKS)) {
        throw new JSONErrorException("Quontation mark expected at ("+
        tokens.peek().getRow() + ", " + tokens.peek().getColumn() + ")");
    }
    tokens.poll();
    String value = tokens.peek().getValue();
    tokens.poll();
    if (!verifyTokensQueue(tokens) ||
!tokens.peek().getTypeOfToken().equals(TokenType.QUONTATION_MARKS)) {
        throw new JSONErrorException("Quontation mark expected at (" +
        tokens.peek().getRow() + ", " + tokens.peek().getColumn() + ")");
    }
    tokens.poll();
    return value;
}
```

*Ukázka zdrojového kódu 25 - vytvoření JSON string hodnoty*

Tato metoda opět jen kopíruje předchozí diagram. Vstupním parametrem je zde fronta tokenů, ze které je načtena hodnota String. Hledají se první uvozovky, poté se načte samotná hodnota, dále se očekávají opět uvozovky. Následně metoda vrací onu načtenou hodnotu.




*Obrázek 12 - EBNF JSON array [2]*

Digram popisující načtení pole v JSON objektu. Pole je uvozeno hranatými závorkami. Uvnitř pole pak jsou hodnoty, oddělené čárkou, tyto hodnoty jsou uváděny bez názvu.

```
private JSONArray parseJSArray(Queue<Token> tokens, String name) throws
JSONErrorException {
    if (!verifyTokensQueue(tokens) ||
!tokens.peek().getTypeOfToken().equals(TokenType.SQUARE_BRACKET_START)) {
        throw new JSONErrorException("SQUARE BRACKET START expected at ("
        +tokens.peek().getRow()+", "+tokens.peek().getColumn()+")");
    }
    tokens.poll();
    List<JSONValue> JSONValues = new ArrayList<>();
    while
(!tokens.peek().getTypeOfToken().equals(TokenType.SQUARE_BRACKET_END)) {
        JSONValues.add(parseValue(tokens, name));
        if (tokens.peek().getTypeOfToken().equals(TokenType.COMMA)) {
            tokens.poll();
        } else {
            break;
        }
    }
    if (!verifyTokensQueue(tokens)
||!tokens.peek().getTypeOfToken().equals(TokenType.SQUARE_BRACKET_END)) {
        throw new JSONErrorException("SQUARE BRACKET END expected at (" +
        tokens.peek().getRow() + ", " + tokens.peek().getColumn() + ")");
    }
    tokens.poll();
    return new JSONArray(name, JSONValues);
}
```

*Ukázka zdrojového kódu 26 - vytvoření JSON pole*

Metoda sloužící k vytvoření JSON pole. Jako parametry zde vstupují fronta tokenů a název pole. Na začátku pole je očekávána hranatá závorka. Následně pomocí cyklu jsou načítány hodnoty, dokud není nalezena ukončovací závorka. Poté je ověřeno, zda je pole správně ukončeno, a je vrácena nová instance třídy *JSONArray* s příslušnými hodnotami a názvem.

### 7.3.2. Validace

Validací souboru se rozumí kontrola, zda soubor odpovídá předepsaným pravidlům. Validace souboru je prováděna při analyzování souboru, kde je v případě chyby v syntaxi vyvolána příslušná chyba s odpovídající hláškou. Soubor je tedy validován již při jeho otevírání, v případě, že není validní, není pak vůbec otevřen. Dále validace probíhá při jakékoliv změně v textovém poli. To je realizováno pomocí nového vlákna, které zkouší text převést na JSON objekt a tím zjistit, zda je validní (v obou případech je pak zobrazena hláška, kde se vyskytla chyba). Validace při změně v textovém poli je realizována pomocí třídy *TextChangeChecking* implementující rozhraní *Runnable*.

```
public void run() {
    while
(lastTimeTextChanged.plusSeconds(5).compareTo(LocalDateTime.now()) > 0) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    List<Lexem> listOfLexems =
        Lexer.getInstance().createLexemsFromString(stringOfJSONFile);
    IJSONParser parser = new JSONParser();
    Queue<Token> listOfTokens =
        parser.createTokensFromLexems(listOfLexems);
    try {
        parser.parseJSObject(listOfTokens, "");
    } catch (JSONErrorException ex) {
        Platform.runLater(() -> {
            Alerts.getInstance().showAlert("JSON parsing error", "JSON
        File is not valid", ex.getMessage(), Alert.AlertType.ERROR);
        });
    }

}
```

*Ukázka zdrojového kódu 27 - třída TextChangeChecking metoda run*

Metoda run třídy *TextChangeChecking* uspává vlákno, dokud od posledního času změny neuplynulo 5 sekund. 5 sekund je proto, že by se jinak vykonávala metoda při každém stisku klávesy. Je tedy předpokládáno, že uživatel bude chvíli psát, a jakmile přestane psát, proběhne validace. Pokud již 5 sekund uplynulo, tak se „pokouší“ vytvořit JSON objekt z řetězce. Pokud nastane při vytváření objektu chyba, není text validní a je zobrazena hláška o typu chyby a o tom, ve kterém místě chyba nastala.

```
@FXML
private void initialize() {

    textChangeChecking = new TextChangeChecking(LocalDateTime.now(), "");
    threadToTextCheck = new Thread(textChangeChecking);
    threadToTextCheck.start();
    textAreaJSON.textProperty().addListener((observable, oldValue,
    newValue) ->
        {
        textChangeChecking.setLastTimeTextChanged(LocalDateTime.now());
        textChangeChecking.setStringOfJSONFile(textAreaJSON.getText());
        if (!threadToTextCheck.isAlive()) {
            threadToTextCheck = new Thread(textChangeChecking);
            threadToTextCheck.start();
        }
    });
}
```

*Ukázka zdrojového kódu 28 - inicializace GUI*

Při inicializaci GUI je vytvořeno nové vlákno s instancí této vytvořené třídy. Třída *TextChangeChecking* má jako parametr čas poslední změny. CodeArea, v němž je JSON soubor jako text, má pak přiřazen listener, který nastaví instanci třídy aktuální čas a předá jí text z CodeArea. Pokud je současné vlákno ukončeno, vytvoří se nové a spustí.

### 7.3.3. Zápis

Zápis JSON objektu do souboru je realizován pomocí dvou tříd. Pomocí třídy *JSONWriter* a třídy *JSONConverter*. *JSONConverter* realizuje převod JSON objektu na řetězec a *JSONWriter* už poté realizuje samotný zápis řetězce do souboru.

```
public void writeJSONToFile(JSONObject JSONObject, File fileToWrite)
throws IOException, JSONErrorException {
    if(fileToWrite!=null) {
        IJSONConverter JSONConverter = new JSONConverter();
        String JSONinString = JSONConverter.convertJSON(JSONObject);
        BufferedWriter writer = new BufferedWriter(
                new FileWriter(fileToWrite));
        writer.write(JSONinString);
        writer.close();
    }
}
```

*Ukázka zdrojového kódu 29 - třída JSONWriter*

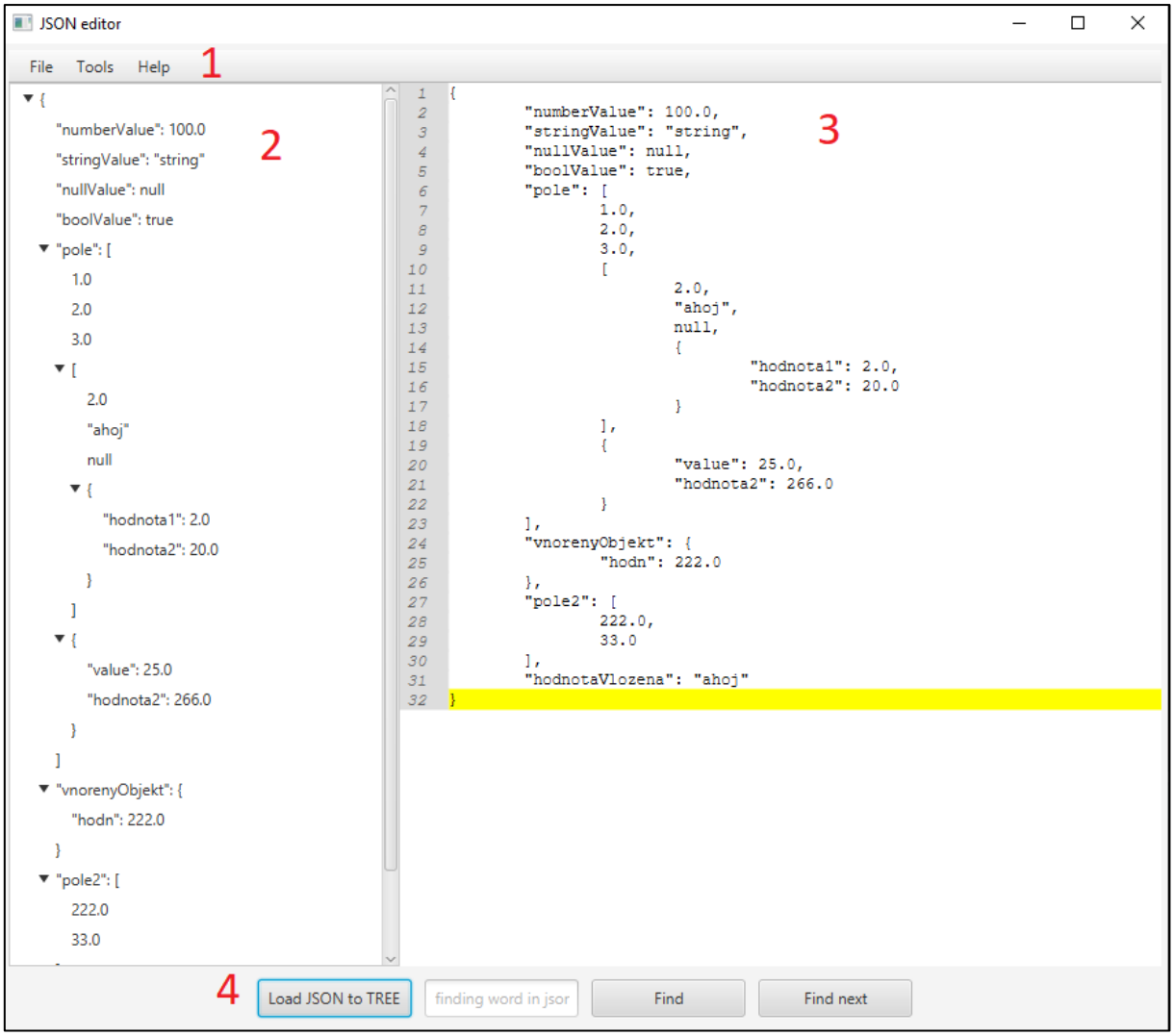
Zde je pomocí třídy *JSONConverter* převeden JSON objekt na řetězec, který je následně pomocí *BufferedWriter*u zapsán do již předem otevřeného souboru.

```
public String convertJSON(JSONObject object) throws JSONErrorException {
    StringBuilder JSONStringBuilder = new StringBuilder();
    numberOfTabs = 0;
    JSONStringBuilder.append("{");
    numberOfTabs++;
    convertJSONObjectValues(object, JSONStringBuilder);
    numberOfTabs--;
    JSONStringBuilder.append("\n}");
    return JSONStringBuilder.toString();
}

private void convertJSONObjectValues(JSONObject value,
        StringBuilder JSONStringBuilder) throws JSONErrorException {
    for (JSONValue val : value.getValue()) {
        convertJSONValueWithName(val, JSONStringBuilder);
        JSONStringBuilder.append(",");
    }
    if (!value.getValue().isEmpty()) {

JSONStringBuilder.deleteCharAt(JSONStringBuilder.lastIndexOf(","));
    }
}

private void convertJSONValue(JSONValue JSONValue, StringBuilder
JSONStringBuilder) throws JSONErrorException {
    if (isJSArray(JSONValue)) {
        convertJSONArray((JSONArray) JSONValue, JSONStringBuilder);
    } else if (isJSObject(JSONValue)) {
        convertNestedJSONObject((JSONObject) JSONValue,
        JSONStringBuilder);
    } else if (isNumberValue(JSONValue)) {
        convertJSONNumberValue(((JSONNumberValue) JSONValue).getValue(),
        JSONStringBuilder);
    } else if (isStringValue(JSONValue)) {
        convertJSONStringValue(((JSONStringValue) JSONValue).getValue(),
        JSONStringBuilder);
    } else if (isNullValue(JSONValue)) {
        convertJSONNullValue(JSONStringBuilder);
    } else if (isBoolValue(JSONValue)) {
        convertJSONBoolValue(((JSONBoolValue) JSONValue).getValue(),
        JSONStringBuilder);
    } else {
        throw new JSONErrorException("Writing error, Unexpected type of"
        +" value in JSON file.");
}
```

*Ukázka zdrojového kódu 30 - třída JSONConverter*

Třída *JSONConverter* realizuje převod JSON objektu na řetězec. Veřejně přístupná je pouze metoda *convertJSON*, ostatní metody jsou pouze pomocné, které převádí hodnoty objektu. Metoda *covertJSONObjectValues* realizuje převod všech hodnot, které objekt obsahuje, tato metoda v cyklu volá metodu, jež převede název hodnoty a následně samotnou hodnotu. Metoda *convertJSONValue* realizuje převod hodnoty, prochází několika podmínkami a ověřuje se, který typ hodnoty to právě je. Dle toho zavolá příslušnou metodu, která převede danou hodnotu na řetězec.

## 7.4. Popis uživatelského rozhraní




*Obrázek 13 - grafické uživatelské rozhraní [zdroj vlastní]*

1 – **Menu** slouží pro ovládání aplikace, je rozděleno na tří části:

- **File** – slouží pro manipulaci se soubory
  - **Open** – zobrazí dialog pro vybrání souboru, který je následně otevřen
  - **Save** – uloží změny do právě otevřeného souboru
  - **Save as** – zobrazí se dialog pro vybrání souboru, do kterého jsou následně uloženy změny
  - **Close** – zavře celou aplikaci, po zeptání se, zda chce uživatel uložit změny
- **Tools** – obsahuje nástroje pro práci se soubory
  - **Find** – realizuje vyhledávání v JSON souboru
  - **Find next** – vyhledá další

- **About** – obsahuje nápovědu a jednoduchý návod, jak aplikaci používat

2 – **TreeView**

- Komponenta, která zobrazuje JSON soubor v přehledné stromové struktuře

3 – **CodeArea**

- Komponenta, která zobrazuje JSON soubor jako text, ve kterém lze soubor editovat

4 – Panel pro operace s JSON souborem

- Tlačítko **Load JSON to TREE**
    - Převádí JSON z CodeArea do TreeView
- TextField „**finding word in JSON**“
    - Pole pro zadání hodnoty k vyhledávání
- Tlačítko **Find**
    - Tlačítko realizuje vyhledávání v JSON souboru
- Tlačítko **Find next**
    - Tlačítko realizuje vyhledávání dalšího výskytu v JSON souboru

# ZÁVĚR

Cílem této bakalářské práce byl vznik optimálního a efektivního editoru JSON souborů napsaného v jazyce Java. Teoretická část je věnována popisu JSON souborů, analýze existujících nástrojů, využitým technologiím a v neposlední řadě využitým datovým strukturám při vývoji.

První kapitola je věnována JSON souborům. Kapitola obsahuje popisu JSON souborů a možnostem jejich využití. Dále jsou zde uvedeny ukázky strukutry JSON souborů

Druhá kapitola je věnována analýze požadavků. Obsahuje funkční a nefunkční požadavky na editor.

Třetí kapitola se věnuje důvodům pro vytvoření nástroje. Jsou zde vybrány hlavní důvody, proč je dobré mít efektivní nástroj pro editaci JSON souborů.

Čtvrtá kapitola obsahuje analýzu již existujících nástrojů, které slouží pro práci s JSON soubory. Všechny tyto nástroje byly otestovány na rozsáhlejších souborech a jsou zde vyzdviženy jejich kladné vlastnosti a samozřejmě také záporné vlastnosti. V závěru této kapitoly je porovnání existujících nástrojů.

Pátá kapitola obsahuje popis využitých technologií při vývoji aplikace. Je zde popsána daná technologie, její základní syntaxe, případně využité prvky z této technologie.

Šestá kapitola popisuje využité datové struktury při vývoji. Vždy je popsána daná strukutra, průběh její inicializace a příklady prováděných operací. Popřípadě i to, k čemu je možné tuto strukturu využít.

Sedmá kapitola je věnována samotné implementaci nástroje a jeho popisu. Stručně popisuje nástroj, který je výsledkem této práce, jednoduché uspořádání strukutry projektu.

Problémem této aplikace je grafické uživatelské rozhraní, které bylo vytvořeno za pomocí JavaFX. JavaFX není úplně plynulá a nedá se říci, že by byla optimální pro takovou to aplikaci. Proto by byl mnohem vhodnější vývoj této aplikace jako webové, popřípadě zvolit úplně jiný programovací jazyk, což si myslím, že by přineslo ještě větší efektivitu nástroje.

Toto téma jsem si zvolil kvůli mému zájmu o JSON soubory, o kterých moc dobře vím, že hýbou světem programování a celkově IT. Při psaní této práce jsem se naučil hodně o fungování JSON souborů a využití správných datových struktur. Doufám, že tato práce mi bude přínosem do budoucna.

# POUŽITÁ LITERATURA


[1] HASSMAN, Martin. JSON: jednotný formát pro výměnu dat. *Zdroják.cz* [online]. [cit. 2021-04-22]. Dostupné z: https://zdrojak.cz/clanky/json-jednotny-format-pro-vymenu-dat/

[2] Úvod do JSON. *JSON.org* [online]. [cit. 2021-04-22]. Dostupné z: https://www.json.org/json-cz.html

[3] BASSETT, Lindsay. *Introduction to JavaScript Object Notation: A To-the-Point Guide to JSON*. 1st. O'Reilly Media, 2015. ISBN 978-1491929483.

[4] HUTTON, B. a SPOL. *JSON Schema: A Media Type for Describing JSON Documents* [online]. [cit. 2021-04-23]. Dostupné z: https://json-schema.org/draft/2020-12/json-schema-core.html

[5] SCHOERGENHUMER, Stefan a Markus DOPLER. *EBNF Visualizer - Manual* [online]. [cit. 2021-04-23]. Dostupné z: https://people.cs.vt.edu/~kafura/ComputationalThinking/Class-Notes/Manual/manual.html

[6] ARLOW, Jim a Ila NEUSTADT. *UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky*. 2., aktualiz. a dopl. vyd. Brno: Computer Press, 2007. ISBN 978-80-251-1503-9.

[7] XML Editor: Oxygen XML Editor 23.1. *SyncRO Soft SRL* [online]. [cit. 2021-04-22]. Dostupné z: https://www.oxygenxml.com/xml_editor.html

[8] JSON Editor Plugin. *Eclipse Foundation: marketplace* [online]. [cit. 2021-04-22]. Dostupné z: https://marketplace.eclipse.org/content/json-editor-plugin

[9] Json Tools. *Eclipse market place* [online]. [cit. 2021-04-23]. Dostupné z: https://marketplace.eclipse.org/content/json-tools

[10] DE JONG, Jos. *JSON Editor online* [online]. [cit. 2021-04-22]. Dostupné z: https://jsoneditoronline.org/

[11] 15+ JSON EDITORS – FREE, ONLINE, WINDOWS, MAC, BROWSER. *Butler Analytics* [online]. [cit. 2021-04-22]. Dostupné z: http://www.butleranalytics.com/15-json-editors-free-online-windows-mac-browser/

[12] GOSLING, James, Bill JOY, Guy STEELE, Gilard BRACHA a Alex BUCKLEY. The Java® Language Specification: Java SE 8 Edition. *Oracle: Documentation* [online]. [cit. 2021-04-22]. Dostupné z: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

[13] SCHILDT, Herbert. *Java 8: výukový kurs*. 1st. Brno: Computer Press, 2016. ISBN 978-80-251-4665-1.

[14] JavaFX: Getting Started with JavaFX. *Oracle: Java documentation* [online]. [cit. 2021-04-22]. Dostupné z: https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784

[15] JavaFX Scene Builder: Getting Started with JavaFX Scene Builder. *Oracle: Java documentation* [online]. [cit. 2021-04-22]. Dostupné z: https://docs.oracle.com/javase/8/scene-builder-2/get-started-tutorial/overview.htm#JSBGS164

[16] TOMASMIKULA, a A SPOL. RichTextFX. *GitHub* [online]. [cit. 2021-04-22]. Dostupné z: https://github.com/FXMisc/RichTextFX

[17] Class TreeView<T>. *Oracle: Java documentation* [online]. [cit. 2021-04-22]. Dostupné z: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/TreeView.html

[18] IT-SLOVNIK.CZ TEAM. Co je to verzování?. *IT slovnik.cz* [online]. [cit. 2021-04-23]. Dostupné z: https://it-slovnik.cz/pojem/verzovani

[19] What is Git. *Atlassian: BitBucket* [online]. [cit. 2021-04-22]. Dostupné z: https://www.atlassian.com/git/tutorials/what-is-git

[20] MICHÁLEK, Ondřej. Lekce 2 - Datové struktury pole a list. *ITnetwork.cz* [online]. [cit. 2021-04-22]. Dostupné z: https://www.itnetwork.cz/navrh/algoritmy/algoritmy-datove-struktury/datove-struktury-pole-a-list

[21] Java ArrayList. *W3schools.com* [online]. [cit. 2021-04-22]. Dostupné z: https://www.w3schools.com/java/java_arraylist.asp

[22] CUTAJAR, James. *Beginning Java Data Structures and Algorithms: Sharpen your problem solving skills by learning core computer science concepts in a pain-free manne*. Packt Publishing, 2018. ISBN 978-1491929483.

[23] BEALDUNG. Time Complexity of Java: Collections. *Baeldung* [online]. [cit. 2021-05-07]. Dostupné z: https://www.baeldung.com/java-collections-complexity

[24] MICHÁLEK, Ondřej. Lekce 4 - Fronta a zásobník. *ITnetwork.cz* [online]. [cit. 2021-04-22]. Dostupné z: https://www.itnetwork.cz/navrh/algoritmy/algoritmy-datove-struktury/fronta-a-zasobnik

[25] DOC. MGR. DVORSKÝ, Jiří, Ph.D. *Lineární datové struktury* [online]. [cit. 2021-05-09]. Dostupné z: http://www.cs.vsb.cz/dvorsky/Download/Algorithms2018/AlgorithmsII/Slides/Lecture01.pdf

[26] SMUDA, Jan. Syntaktická analýza – Lexikální analyzátor. *Smuda.cz* [online]. [cit. 2021-04-22]. Dostupné z: http://www.smuda.cz/syntakticka-analyza-lexikalni-analyzator/