FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

# ZADÁNÍ DIPLOMOVÉ PRÁCE

| | |
|---|---|
| **Název:** | Návrh a implementace zpracování formátu JSON ve frameworku x-definice |
| **Student:** | Bc. Gabriela Melingerová |
| **Vedoucí:** | Mgr. Václav Trojan |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Webové a softwarové inženýrství |
| **Katedra:** | Katedra softwarového inženýrství |
| **Platnost zadání:** | Do konce zimního semestru 2020/21 |

## Pokyny pro vypracování

X-definice je rozsáhlý framework určený k validaci a transformaci strukturovaných dat, jehož součástí může být dynamický přístup k externím zdrojům. Cílem této práce je návrh a implementace zpracování formátu JSON v tomto frameworku.

Postupujte v těchto krocích:

1) seznamte se se stávající implementací frameworku x-definice, zejména s prototypem, který již JSON formát v omezené míře zpracovává
2) seznamte se se stávajícím návrhem specifikace JSON Schema
3) navrhněte rozšíření frameworku X-definice o možnost zpracování formátu JSON s ohledem na existující návrh specifikace JSON Schema
4) navrhněte transformaci mezi formáty JSON a XML
5) návrh implementujte
6) implementaci řádně zdokumentujte a otestujte

## Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 18. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

# Návrh a implementace zpracování formátu JSON ve frameworku X-definice

***Bc. Gabriela Melingerová***

Katedra softwarového inženýrství
Vedoucí práce: Mgr. Václav Trojan

5. května 2019

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 5. května 2019 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Gabriela Melingerová. Všechna práva vyhrazena.
*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.*

### Odkaz na tuto práci

Melingerová, Gabriela. *Návrh a implementace zpracování formátu JSON ve frameworku X-definice*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

# Abstrakt

Tato diplomová práce se zabývá stávající implementací frameworku X-definice se zaměřením na dosavadní prototyp pro zpracování formátu JSON. Dále se práce zaobírá současným návrhem specifikace JSON Schema. Dalším bodem diplomové práce je navržení transformace mezi formáty JSON a XML, navržení rozšíření frameworku X-definice o možnost zpracování formátu JSON s ohledem na existující návrh specifikace JSON Schema a následná implementace v jazyce Java. Výsledkem této práce je rozšíření frameworku X-definice, který je schopný zpracovávat a transformovat formát JSON.

**Klíčová slova** X-definice, JSON, XML, framework, Java

# Abstract

This master's thesis deals with current implementation of the X-definition framework. It focuses on current prototype for JSON processing. Furthermore, the thesis deals with the current design of JSON Schema. Other goals of this thesis are to propose a transformation between JSON and XML formats, to propose an extension of the X-definition framework with the possibility to process JSON with respect to the existing JSON Schema specification and to implement all this in Java. The result of this work is the extension of the X-definition framework, which is able to process and transform JSON.

**Keywords** X-definition, JSON, XML, framework, Java

# Obsah

# Úvod

V dnešní době velké množství firem i osob potřebuje vytvářet, validovat a zpracovávat velká množství XML dokumentů. K tomuto účelu slouží framework X-definice, který je schopen výše zmíněné potřeby naplnit a dokonce umí zpracovávat XML dokumenty o velikosti jednotek až desítek GB [1].

V současnosti existují různé formáty pro popis a serializaci strukturovaných dat například již zmíněné XML nebo JSON, YAML atd.. Proto se jeví jako vhodné řešení rozšířit framework X-definice o zpracování formátu JSON, který se těší velké oblibě.

Téma diplomové práce jsem si vybrala proto, že už jsem se v minulosti setkala s formátem JSON, konkrétně v bakalářské práci. A technologie X-definice mě zaujala tím, že je výtvorem české firmy Syntea a popis XML dokumentu pomocí šablony X-definice je velmi intuitivní.

V této práci se seznamuji se stávající implementací frameworku X-definice, zejména s prototypem, který již JSON formát v omezené míře zpracovává. Dále pak pokračuji s analýzou návrhu specifikace JSON Schema. V návrhu se zaměřím na transformaci mezi formáty JSON a XML a rozšíření frameworku X-definice o možnost zpracování formátu JSON s ohledem na existující návrh specifikace JSON Schema. V implementační části se snažím tyto návrhy zrealizovat, aby výsledkem byl framework rozšířený o zpracování formátu JSON.

Tato diplomová práce je vypracována na základě spolupráce s firmou Syntea, která má registrovanou technologii frameworku X-definice.

Kapitola 1

# X-definice

Než začneme s analýzou stávající implementace frameworku X-definic, měli bychom si říci, co to vlastně ta X-definice je.

## 1.1 X-definice

X-definice, jak už zde bylo zmíněno, je opensource projekt spravovaný firmou Syntea software group a.s., který pomáhá validovat, zpracovávat a vytvářet XML dokumenty 1.1. X-definice 1.2 je i jazyk, díky kterému můžeme popsat strukturu, obsah, a i zpracování obsahu XML objektů nebo jejich konstrukci. Hlavní výhodou tohoto jazyka je, že je velmi intuitivní, protože se jedná o XML dokument, který struktuou velmi připomíná popisovaná data [2].

```
<Clovek>
    <Jmeno>Adam</Jmeno>
    <Prijmeni>Novotny</Prijmeni>
    <Plat>30000</Plat>
</Clovek>
```

Listing 1.1: Jednoduché XML

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/3.2"
 root="Clovek" >
        <Clovek>
                <Jmeno>string()</Jmeno>
                <Prijmeni>string()</Prijmeni>
                <Plat>int(10,100000)</Plat>
        </Clovek>
</xd:def>
```

Listing 1.2: Jednoduchá X-definice

### 1.1.1 Režimy práce procesoru X-definic

#### 1.1.1.1 Režim validace

Tento režim zpracovává a kontroluje výskyt objektů podle X-definice. „Proces je v tomto režimu řízen zpracováním vstupního zdrojového tvaru XML objektu, k němuž se vyhledají odpovídající části X-definice.“[3]. Režim je vhodný ke kontrole dat.

#### 1.1.1.2 Režim konstrukce

„Režim je řízen X-definicí (nikoliv vstupními daty jako v prvním případě). Proces je tomto případě řízen zpracovávanou X-definicí, která slouží jako předpis, podle kterého se tvoří výsledek.“[3]. Tento režim je vhodný k vytváření objektů či transformaci vstupních objektů do jiného tvaru. Tam, kde se u režimu validace kontroluje a zpracovávají vstupní data, tak v tomto režimu se vytváří objekt podle návodu v X-definici - nejčastěji akcí **create** 1.

### 1.1.2 Model elementu

Model elementu je základem X-Definic. Díky němu můžeme popsat strukturu XML Elementu a jeho další vlastnosti. V modelu elementu se může objevit **kontrola typu dat**, např. string(), kde obsah musí být typu string, aby prošel validací bez chyb - toto tvoří **validační sekci** [2]. Příklady možných forem kontroly typu dat:

1. **string()**, **int()**, **email()** atd.
2. **int(10, 100000)** - hodnota musí být číslo v rozsahu 10 až 100000, **string(2,30)** - hodnota musí být řetězec, který má minimálně 2 a maximálně 30 znaků.

Příkladu modelu elementu z XML 1.1 můžeme vidět zde 1.3.

```
<Clovek>
        <Jmeno>string()</Jmeno>
        <Prijmeni>string()</Prijmeni>
        <Plat>int(10,100000)</Plat>
</Clovek>
```

Listing 1.3: Model elementu Clovek

### 1.1.3 Skript

„Skript X-Definic je jazyk, který se zapisuje jako textová hodnota do atributů nebo textových uzlů. Pro zápis skriptu elementu je zaveden speciální pomocný atribut „xd:script“. Pomocí skriptu se popisují vlastnosti dat a akce, spojené s jejich zpracováním při různých událostech. Skript má volný formát a je složen z několika částí. Pořadí jednotlivých částí skriptu není závazné a žádná část skriptu není povinná (podle typu události může být doplněna přednastavenou hodnotou). Jako oddělovač jednotlivých částí slouží znak „;“ (středník).“[3]. Nyní se podíváme, co všechno může být ve skriptu.

#### 1.1.3.1 Validační sekce

Validační sekce kontroluje **výskyt elementu**. Pokud chybí doplní se počet výskytů na 1 - required. To znamená, že element se musí objevit ve vstupním XML dokumentu, který chceme kontrolovat. Za specifikací výskytu se může objevit zápis pro **kontrolu typů**. Chybí-li typová kontrola doplní se string() [3].

##### 1.1.3.1.1 Výskyt elementů

1. **?** - element se nemusí nebo se může vyskytnou pouze jednou (stejné jako 0..1 nebo optional).
2. **optional** - element se nemusí nebo se může vyskytnou pouze jednou (stejné jako 0..1 nebo ?).
3. **+** - element se musí vyskytnou jednou nebo vícekrát (stejné jako 1..* nebo required).
4. **required** - element se musí vyskytnou jednou nebo vícekrát (stejné jako 1..* nebo required).
5. ***** - element se může nebo nemusí vyskytnout, počet výskytů není omezen (stejné jako 0..*).
6. **m** - element se musí vyskytnout právě m-krát.
7. **m..n** - element se musí vyskytnout m-krát a maximálně n-krát.
8. **n..*** - element se musí vyskytnout n-krát a smí se vyskytnout neomezeně.
9. **ignore** - element se sice může vyskytnout, ale jeho výskyt je v dalším zpracování ignorován.
10. **illegal** - element se nesmí vyskytnout.

#### 1.1.3.1.2 Kontrola typů

**1.1.3.1.2.1 Metody pro kontrolu typů shodných s typy v XML schématech** Zde se seznámíme s metodami pro kontrolu typů, které se shodují s typy v XML Schématech. Představíme si jich jen pár, o dalších se můžeme dočíst v dokumentaci.

1. **string()** - hodnota musí být znakový řetězec.
2. **double**() - hodnota musí být číslo s pohyblivou řádovou čárkou.
3. **int()** - hodnota musí být celé číslo.
4. **time()** - hodnota musí být čas.
5. **date()** - hodnota musí být datum.

**1.1.3.1.2.2 Metody pro kontrolu typů, které nejsou v XML schématech** Dále tu máme metody pro kontrolu typů, které nejsou v XML schématech. Opět si ukážeme jenom některé, o dalších si můžeme přečíst opět v dokumentaci.

1. **email()** - hodnota musí být emailová adresa.
2. **eq(s)** - hodnota se musí rovnat hodnotě řetězce „s“.
3. **num()** - hodnota je posloupnost číslic.
4. **ends(s)** - hodnota musí končit hodnotou řetězce v parametru „s“.
5. **uri()** - hodnota musí být formálně správný zápis URI, tak jak je implementován v Java.

### 1.1.3.2 Akce

Akce nám říkají, co se má provést při různých událostech ve fázích zpracování objektu. Akce se skládá ze jména události, za ním následuje skript s příkazem, který se provede v příslušné události [3]. Příklady událostí:

1. **create** - „Událost nastává jen v režimu kompozice v okamžiku, když se zahajuje zpracování nového objektu při průchodu X-definicí (ještě před událostí „init“). Příkaz této akce je výraz, jehož výsledkem je odpovídající objekt (element, atribut nebo textová hodnota - obvykle se jedná o vyhledání objektu ve vstupních datech nebo o vytvoření nového objektu). U atributů a textových uzlů je jako výsledek příkazu očekáván textový řetězec a u elementů se očekává objekt typu element (jméno a předci tohoto elementu jsou pro další zpracování nevýznamné). Není-li akce specifikována, provede se převzetí odpovídajícího objektu ze

vstupního objektu (podle odpovídající pozice právě zpracovávaného objektu v X-definici).“[3].

2. **init** - akce, která inicializuje, se provede před dalším zpracováním objektu.

3. **onTrue** - akce události se provede, je-li výsledek vyhodnocení typu (výsledku validační akce) „true“. Není-li akce specifikována, uloží se kopie objektu do výsledku.

4. **onAbsence** - tato událost nastane, chybí-li požadovaný objekt nebo není splněna podmínka minimálního výskytu.

5. **match** - událost pro akci „match“ nastává před dalším zpracováním elementu nebo atributu. Akce vrací hodnotu „true“ nebo „false“. Při hodnotě „true“ zpracování dál pokračuje. Je-li hodnota „false“ pak aktuální element není zpracováván podle modelu elementu.

6. **default** - tato událost se týká pouze atributů a textových hodnot. Při absenci atributu nebo textové hodnoty se vyvolá akce, kde se hodnota za akcí „default“ objeví v příslušném atributu či textové hodnotě.

#### 1.1.3.3 Volby - options

Tato část skriptu je již nepovinná. Chybí-li ve skriptu možnost voleb, převezmou se volby, které jsou přednastaveny v X-definici. Volby se zapisují pomocí slova „options“ a následuje seznam jmen voleb, které jsou mezi sebou odděleny čárkou [3].

1. **ignoreComments** - ignoruje komentáře.

2. **setTextLowerCase** - písmena hodnot textových uzlů jsou nastavena na malá.

3. **trimText** - odstraní se mezery, tabulátory, nové řádky na začátku a na konci hodnot textových uzlů elementů. Toto je defaultní nastavení.

4. **noTrimText** - opak volby „trimText“, mezery, tabulátory a nové řádky jsou ponechány.

5. **ignoreTextWhiteSpaces** - mezery v textových uzlech, které jsou navíc, jsou před zpracováním odstraněny.

#### 1.1.3.4 Odkaz

Odkazy slouží k zjednodušení a zpřehlednění modelu elementu. Syntaxe je jednoduchá, stačí ve skriptu napsat: ref a jméno-modelu-elementu. Chceme-li odkazovat na model z jiné X-definice, syntaxe vypadá takto: ref jméno-x-definice#jméno-modelu-elementu [3]. Použití a syntaxi vidíme zde 1.4.

```
<Pivovar jmeno = "required" >
        <Adresa xd:script = "ref Adresa" />
</Pivovar>

<Adresa ulice = "required "
         cislo = "required int()"
         mesto = "required string()"
         PSC = "required num(5)" /
```

Listing 1.4: Příklad odkazu

### 1.1.4 Skupina objektů

Může se nám stát, že při popisování struktury potřebujeme popsat určitou skupinu objektů. Skupinou rozumíme posloupnost elementů. V X-definicích můžeme narazit na tři kategorie skupin: neuspořádaná, výběrová a uspořádaná skupina [3].

#### 1.1.4.1 Neuspořádaná skupina - mixed

Tato skupina popisuje seznam prvků, které mohou mít libovolné pořadí. V neuspořádané skupině se nesmí objevovat prvek se stejným jménem. Seznam elementů se uzavírá do speciálního elementu s názvem „xd:mixed“. Příklad použití můžeme vidět zde 1.5.

```
<Zoo jmeno = "required" >
  <xd:mixed>
    <Hroch xd:script = "occurs ?"
           jmeno = "required string()"/>
    <Lev xd:script = "occurs *"
           jmeno = "required string()"/>
    <Surikata xd:script = "occurs ?"
           jmeno = "required string()"/>
   </xd:mixed>
</Zoo>
```

Listing 1.5: Příklad neuspořádané skupiny mixed

#### 1.1.4.2 Výběrová skupina - choice

Výběrová skupina nám poskytuje možnost popsat různé varianty. Jména elementů v jednotlivých variantách se musí lišit. Skupina se uzavírá do elementu „xd:choice“. V příkladu 1.6 vidíme použití této skupiny. Aby validace proběhla bez chyb, vstupní XML musí obsahovat alespoň jeden prvek z třech zmíněných, které jsou v této skupině.

```
<Zoo jmeno = "required" >
  <xd:choice>
    <Hroch xd:script = "occurs ?"
           jmeno = "required string()"/>
    <Lev xd:script = "occurs *"
           jmeno = "required string()"/>
    <Surikata xd:script = "occurs ?"
           jmeno = "required string()"/>
  </xd:choice>
</Zoo>
```

Listing 1.6: Příklad výběrové skupiny choice

#### 1.1.4.3 Uspořádaná skupina - sequence

U uspořádané skupiny musí prvky dodržovat pořadí, které je jim určeno. Skupina se uzavírá do elementu „xd:sequence“ viz 1.7. Vstupní XML 1.8 by validací neprošlo, neboť elementy „Lev“ a „Hroch“ jsou prohozené, ale prošlo by validací u této X-definice 1.5.

```
<Zoo jmeno = "required" >
  <xd:sequence>
    <Hroch xd:script = "occurs ?"
           jmeno = "required string()"/>
    <Lev xd:script = "occurs *"
           jmeno = "required string()"/>
    <Surikata xd:script = "occurs ?"
           jmeno = "required string()"/>
  </xd:sequence>
</Zoo>
```

Listing 1.7: Příklad uspořádané skupiny sequence

```
<Zoo jmeno = "Zoo Praha" >
    <Lev jmeno = "Karel" />
    <Hroch jmeno = "Pepa" />
    <Surikata jmeno = "Marie" />
</Zoo>
```

Listing 1.8: Příklad vstupního XML

### 1.1.5 Záhlaví X-definice

„Záhlaví X-definice obsahuje informace, popisující její vlastnosti. Tyto vlastnosti se zapisují pomocí atributů s pevně určeným jménem kořenového uzlu X-definice, jejíž jména jsou součástí jmenného prostoru „xdef“. Specifikace některých atributů je povinná a některé atributy jsou volitelné. Kromě atributů ze jmenného prostoru X-definice mohou být v záhlaví X-definice uvedeny uživatelské parametry, na jejichž hodnoty je možné se odvolávat ve skriptu uvnitř X-definice.“[3].

#### 1.1.5.1 Povinné atributy

**1.1.5.1.1 Specifikace jmenného prostoru - xmlns:xd** Povinné je uvádět: *xmlns:xd = "http://www.xdef.org/xdef/3.2"*, kde se poslední číslo mění podle aktuální verze.

**1.1.5.1.2 Jméno X-definice - xd:name** Dalším povinným atributem je jméno X-definice, které musí být jednoznačné v rámci všech použitých X-definic.

#### 1.1.5.2 Nepovinné atributy

**1.1.5.2.1 Specifikace seznamu přípustných kořenových elementů - xd:root** „Tato specifikace se zapíše pomocí atributu „xd:root“, který obsahuje seznam modelů elementů, které popisují elementy, které jsou v X-definici popsány. V jedné X-definici totiž může být popsáno více kořenových elementů, z nichž se ten odpovídající vybere podle jména. Jednotlivá jména jsou oddělena znakem „—“ (čteme jako „nebo“).“ [3]. Na příkladě 1.9 můžeme vidět, že kořenovým elementem může být model elementu „Zoo“ nebo model elementu „Cirkus“.

```
xd:root = "Zoo | Cirkus
```

Listing 1.9: Příklad xd:root

## 1.2 Implementace

### 1.2.1 Objekty

V této sekci si představíme objekty, které se vyskytují v X-Definicích.

#### 1.2.1.1 XDPool

„XDPool je binární objekt, který vznikne překladem X-Definic ze zdrojového tvaru. Objekt XDPool lze vytvořit např. pomocí statické metody **compileXD**. První parametr této metody je hodnota typu „java.util.Properties“, kde je možné nastavit některé parametry pro zpracování. Pokud je tento parametr „null“, použijí se hodnoty nastavené sytémem
(metodou System.getProperties()). Další parametry pak specifikují X-definice, které se přeloží.“[4].

#### 1.2.1.2 XDDocument

XDDocument se vytvoří z objektu XDPool pomocí metody **createXDDocument**, dále je XDDocument používán pro práci s požadovaným XML dokumentem. „Výsledkem procesu z objektu XDDocument je XML dokument (případně X-komponenta). Při zpracování X-definice vznikne také protokol (informace o chybách, varování atd). V objektu XDDocument jsou uloženy hodnoty proměnných, deklarovaných v X-definicích. Kromě výsledného XML dokumentu je pak možné případně získat z objektu XDDocument i hodnoty proměnných (pomocí metody **getVariable**). Protokol o chybách, varování atd. je uložen v tzv. **reportéru**, který je předán procesoru X-definic pomocí parametru při spuštění.“[4].

### 1.2.2 Režim validace

Tento režim validování a zpracovávání vstupního dokumentu se spouští metodou **xparse**, která je metodou instance třídy XDDocument. Prvním parametrem je vstupní dokument, druhým - nepovinným je reportér pro hlášení chyb. Program v X-Definici vyhledá příslušný model, který je deklarován v záhlaví X-Definice jako atribut **xd:root**. Dále je zpracovává podle tohoto modelu. Kontroluje výskyt jednotlivých prvků a u textové hodnoty validuje podle validačních metod [4].

### 1.2.3 Režim konstrukce

Tento režim konstrukce se spouští metodou **xcreate**, která je opět metodou instance třídy XDDocument. Prvním parametrem je model, podle kterého se vytvoří výsledný XML dokument. Druhým parametrem může být výše zmíněný reportér. „Jednotlivé části vytvářeného dokumentu se konstruují podle zadaného modelu na základě tzv. kontextu. Kontext jsou data, která

jsou v daném okamžiku tvorby použita pro tvorbu cílového XML objetu. Při tvorbě každé části vytvářeného dokumentu je (anebo není) k dispozici nějaký aktuální kontext. Pokud je ve Skriptu modelu XML objektu, který je konstruován zapsán příkaz sekce „create“, pak se pro tvorbu tohoto XML objektu stane kontextem hodnota, která je výsledkem příkazu sekce „create“ (tato hodnota musí být použitelná jako kontext). Tato hodnota se pak stane výchozí pro všechny vytvářené potomky. Požadovaný typ této hodnoty se liší podle druhu vytvářeného prvku. Často používanou metodou create sekce je XPath výraz, který pracuje s XML dokumentem, který byl přiřazen jako výchozí kontext pro tvorbu výsledku (tímto způsobem lze provést „transformaci“ XML dokumentu z kontextu do výsledné podoby).“[4].

Kapitola 2

# Stávající implementace frameworku X-Definice - prototyp JSON

V této kapitole se seznámíme s tím, jak vypadá stávající implementace X-definic, která zpracovává formát JSON 4.1.1 - jedná se o verzi 3.2.. Implementace je psaná v Javě, minimální verze Javy je 1.6.

## 2.1 Průběh - validační mód

Celý průběh X-definic, kde vstupem je JSON, můžeme pozorovat v složce **Other Sources** → **documentation** → **examples** - **ExampleXDef.java**, tato konkrétní ukázka ověřuje parsování JSON objektů pomocí X-definice.

```
Object json =  JSONUtil.parseJSON(source);
Element e1 = JSONUtil.jsonToXml(json);
String xml = KXmlUtils.nodeToString(e1);
String xdef =
    KXmlUtils.nodeToString(XJUtil.jsonToXDef(json), true);
ArrayReporter reporter = new ArrayReporter();

XDPool xp = XDFactory.compileXD(null, xdef);
XDDocument xd = xp.createXDDocument();
Element e2 = xd.xparse(xml, reporter);
```

Listing 2.1: Příklad - průběh X-definic, kde vstupem je JSON

Zde vidíme 2.1, že nejdříve se vytvoří Object „json“ pomocí metody **JSONUtil.parseJSON**, která má jako vstupní parametr typ String. Dále se Object „json“ stane vstupem do metody **jsonToXml**, která se opět nachází ve třídě **JSONUtil**. Tato metoda vytvoří Element, který se převede na typ

String pomocí **nodeToString** metody. Dále se vytvoří „xdef“ (X-definice), která se vytvoří převodem Objectu „json“ na Element X-definice pomocí metody **XJUtil.jsonToXDef**. Objekt „xp“ typu XDPool se vytvoří kompilací X-definice a Element „e2“ se vytvoří parsováním, kde vstupem je „xml“ a „reporter“, který zachytává chyby, a „xml“. Průběh, dá se říct, se skládá z transformací mezi jednotlivými formáty a pak zapojením X-definic.

## 2.2 Transformace

### 2.2.1 parseJSON

O této metodě si můžeme přečíst zde 4.5.1 a zde 4.4.1.1. Metoda volá další metodu **parseJSON**, která zparsuje JSON pomocí **StringParseru** a to tak, že zavolá **JParser(parser).parse()**, kde „parser“ je typu StringParser. Metoda **parse** volá metodu **readValue**, která postupně prochází vstup znak po znaku a kontroluje, zpracovává a vytváří dané objekty typu Object. Tato metoda je rekurzivní. Zde 2.2 můžeme vidět, jak metoda pracuje, když se dostane ke znaku „{“.

```
if (_p.isChar('{')) { // Map
  Map<String, Object> result =
     new TreeMap<String, Object>();
  _p.isSpaces();
  if (_p.isChar('}')) {
    return result;
  }
  for (;;) {
    Object o = readValue();
    if (o != null && o instanceof String) {
          // parse JSON named pair
          String name = (String) o;
          _p.isSpaces();
          if (!_p.isChar(':')) {
            // ":" expected
          throw new SRuntimeException(JSON.JSON002, ":",
          genPosMod());
      }
      _p.isSpaces();
      result.put(name, readValue());
      _p.isSpaces();
      if (_p.isChar('}')) {
        _p.isSpaces();
        return result;
      }
```

```
        if (_p.isChar(',')) {
          _p.isSpaces();
        }
    } else {
      // String with name of item expected
      throw new SRuntimeException(JSON.JSON004, genPosMod());
      }
}
```

Listing 2.2: Příklad, jak metoda readValue zpracovává JSON objekt

### 2.2.2 jsonToXml

Tato metoda je popsána zde 4.4.1.2 a zde 4.5.2. Metoda volá metodu **json2xml**, která na vstupu dostane Object „json“ a Document „doc“. V této metodě se na základě toho, jaké instance je Object „json“, rozhodne, která další metoda se zavolá. Pokud je „json“ instancí Mapy zavolá se metoda **mapToXml**, pokud je instancí listu zavolá se metoda **listToXml**. V metodě **mapToXml** se dále volá metoda **namedItemToXml**, která převádí jednotlivé prvky mapy na Elementy. V metodě **listToXml** se projíždějí jednotlivé prvky a podle instancí se volají příslušné metody 2.3. Výsledek převodu je k vidění zde 4.28.

```
private void listToXml(final List<?> list ,
                       final Node parent) {
  Element e = appendJSONElem(parent, J_ARRAY);
  for (Object x: list) {
    if (x == null) {
      addValue(e, null);
    } else if (x instanceof Map) {
      mapToXml((Map) x, e);
    } else if (x instanceof List) {
          listToXml((List)x,  e);
        } else {
      addValue(e, x);
    }
  }
  popContext();
}
```

Listing 2.3: Metoda listToXml

### 2.2.3 xmlToJson

Metoda je zmíněna zde 4.4.2.1 a zde 4.5.3. Metoda převádí XML element do Objectu JSON a to tak, že pokud vstupní XML obsahuje URI pro JSON/XML

konverzi „http://www.syntea.cz/json/1.0“ (to znamená, že obsahuje vlastní pojmenované elementy), tak se na základě lokálního jména elementu rozhodne, jaká metoda se bude volat. Je-li lokální jméno rovno „array“, volá se metoda **createList** 2.4, je-li rovno „map“, volá se metoda **createMap**. V metodách se dále procházejí childElementy, které se dále zpracovávají.

```
private static List<Object> createList(final NodeList nl) {
  List<Object> list = new ArrayList<Object>();
  for (int i = 0; i < nl.getLength(); i++) {
    Node n = nl.item(i);
    if (n.getNodeType() == Node.TEXT_NODE
        || n.getNodeType() == Node.CDATA_SECTION_NODE) {
      String s = n.getNodeValue();
      if (s == null) {
        s = "";
      }
    while (i+1 < nl.getLength()
     && ((n = nl.item(i+1)).getNodeType() == Node.TEXT_NODE
      || n.getNodeType() == Node.CDATA_SECTION_NODE)) {
            s += n.getNodeValue();
               i++;
      }
      valuesToList(list, s);
    } else {
        list.add(createItem(n));
      }
  }
  return list;
}
```

Listing 2.4: Metoda createList

### 2.2.4 jsonToXDef

Tato metoda bere jako vstup „json“ typu Object a vrací XML element, který reprezentuje X-definici. Tato metoda volá **json2xd** metodu, která podle instance Objectu volá další metody. Je-li instancí Mapy, zavolá se metoda **mapToXD**, je-li objekt instancí Listu, zavolá se **listToXD**. Metoda je velmi podobná metodě **jsonToXML**, liší se pouze hodnotou mezi tagy, kde při převodu z JSONu do X-definic, je skutečná hodnota nahrazena typovou kontrolou, např.: jstring(), jnumber(). Výslednou X-definici vyrobenou z JSONu 4.6 si můžeme prohlédnout zde 2.5.

```
<xd:def root="js:map"
    xmlns:js="http://www.syntea.cz/json/1.0"
```

```
        xmlns:xd="http://www.xdef.org/3.2">
<js:map>
        <dite/>
        <jmeno>jstring()</jmeno>
        <kocky>
                <js:array>
                        <js:item>jstring()</js:item>
                        <js:item>jstring()</js:item>
                </js:array>
        </kocky>
        <manzel jmeno="string()" prijmeni="string()"></manzel>
        <muz>boolean()</muz>
        <povolani>jstring()</povolani>
        <prijmeni>jstring()</prijmeni>
        <vek>jnumber()</vek>
</js:map>
</xd:def>
```

Listing 2.5: Příklad X-definice vytvořené z JSONu

## 2.3 X-definice

### 2.3.1 compileXD

Metoda **compileXD** má jako vstupní parametr X-definici, ze které se vytváří objekt XDPool 1.2.1.1. X-definici je nutné přeložit do vnitřního tvaru, který je uložen do objektu - XDPool, obsahujícího množinu X-definic [3].

### 2.3.2 createXDDocument

Metoda **createXDDocument**, která je metodou XDPoolu, vytvoří objekt XDDocument 1.2.1.2.

### 2.3.3 xparse

Metoda **xparse** je metodou XDDocumentu a dostane, v tomto konkrétním případě 2.1, na vstup String s XML a ReportWriter. Mohou být různé varianty typů vstupů: XML formát může být typu Node atd.. Tato metoda spustí zpracování vstupních dat podle X-definice [4]. Případné chyby se zapíší do protokolu v konkrétním příkladě do „reporter“ objektu, který je instancí ReportWriter. Metodu můžeme vidět zde 2.6.

```
public final Element xparse(final String xmlData,
  final String sourceId,
  final ReportWriter reporter) throws SRuntimeException {
```

```
  if(xmlData == null || xmlData.length() == 0) {
    // Input XML source is empty or doesn't exist.
    throw new SRuntimeException(Report.error(XDEF.XDEF578));
  }
  ChkParser parser = new ChkParser(reporter, xmlData);
  if (sourceId != null) {
    parser._sysId = sourceId;
  }
  return xparse(parser, reporter);
}
```

Listing 2.6: Metoda xparse

Kapitola 3

# Stávající návrh specifikace JSON Schema

Před tím, než se seznámíme se stávajícím návrhem specifikace JSON Schema, měli bychom si ujasnit, co to takové JSON Schema vůbec je. Ještě před tím bychom si měli přečíst základy o formátu JSON 4.1.1.

## 3.1 JSON Schema

JSON Schema je schéma, které nám umožňuje popsat strukturu a typy, které by JSON data měla mít. JSON Schema je psáno v JSONu [5]. Na příkladu 3.1 můžeme vidět JSON Schema, které popisuje strukturu tohoto JSONu 4.6.

```
{
  "type": "object",
  "properties": {
    "jmeno": {"type": "string"},
    "prijmeni": {"type": "string"},
    "vek": {"type": "integer"},
    "povolani": {"type": "string"},
    "muz": {"type": "boolean"},
    "dite": {"type": "null"},
    "kocky": {"type": "array",
        "items": [{"type": "string"},
                  {"type": "string"}]
                  },
    "manzel": {"type": "object",
                  "properties": {
                          "jmeno": {"type": "string"},
                          "prijmeni": {"type": "string"}
                                  }
```

```
                    }
            }
}
```

Listing 3.1: Příklad JSON Schema

### 3.1.1 Základy JSON Schema

Nejjednodušším schématem, které přijme všechna JSON data je tento prázdný objekt 3.2. Vstupem ale musí být validní JSON například:

1. ”Pivo”,
2. 48,
3. { ”pivo”: ”Pilsner”, ”obal”: [ ”plechovka”, ”lahev”] }

```
{}
```

Listing 3.2: Příklad nejjednodušší JSON Schema

JSON Schema definuje **klíčová slova**:

1. **$schema** - uvádí, že toto schéma je psáno na základě specifického draftu standardu,
2. **$id** - definuje URI pro schéma a základní URI,
3. **title**, description - slouží pouze k popisu, nepřidávají omezení k validaci dat,
4. **type** - validační klíčové slovo, definuje první omezení na naše JSON data,
5. **properties** - validační klíčové slovo, kde hodnota musí být objekt a každá hodnota objektu musí být validní JSON Schema,
6. **required** - validační klíčové slovo, kde hodnota musí být pole a každý element pole, pokud je, musí být unikátní a string.

#### 3.1.1.1 Klíčové slovo type

Naším cílem není použití nejjednoduššího schématu, které přijme jakékoli validní JSON. Tím bychom nic nezkontrolovali. Proto JSON Schema zavádí klíčové slovo **type**, které nám pomůže vymezit typ JSON dat.

Základní typy jsou:

1. string,
2. numerické typy,

3. objekt
4. pole
5. boolean
6. null

Klíčové slovo **type** může mít hodnotu buď string nebo pole stringů, kde každý string má jméno ze základních typů. Validní JSON je pak ten, který se shoduje v jakémkoli uvedeném typu v poli. Schéma 3.3 připouští jen validní JSON data, která jsou typu string, např.: „pivo“, ale číslo 5 by neakceptovalo. Naproti tomu schéma 3.4 přijme jak „pivo“ tak i číslo 5.

```
{ "type": "string" }
```

Listing 3.3: Příklad JSON Schema - string

```
{ "type": ["string", "number"] }
```

Listing 3.4: Příklad JSON Schema - pole

Dále můžeme mít různá omezení u různých typů v rámci properties. Například můžeme nastavit minimální věk ve schématu 3.1 a to přidání páru: "minimum": 3 k věku 3.5. Existuje jich celá řada, dozvíme se o nich později.

```
"vek": {"type": "integer",
                "minimum": 3},
```

Listing 3.5: Příklad JSON Schema - omezení

## 3.2 Stávající specifikace JSON Schema

Aktuálním draftem jsou **draft-handrews-json-schema*-01** dokumenty, které korespodují s **draft-07** meta-schématy. Specifikace je rozdělena do tří sekcí a jedné specifikace, která spolu souvisejí:

1. **Core** - definuje základy JSON Schema.
2. **Validace** - definuje validační klíčová slova JSON Schema.
3. **Hyper-schéma** - definuje hyper-media klíčová slova JSON Schema.
4. **Relativní JSON ukazatelé** - související specifikace, rozšiřuje syntaxi JSON ukazatelů o realitivní ukazatele.

Meta-schémata jsou schémata, proti kterým mohou být ostatní schémata validována. Jsou samopopisující: JSON Schema meta-schématu se validuje samo. Zatímco JSON Hyper-schéma meta-schématu se také validuje samo a ještě definuje svůj „vlastní link“ [6].

### 3.2.1 Core

Sekci o základu JSON Schema jsme si hrubě popsali zde 3.1.1. Ve všech sekcích se používají určité výrazy, které si definujeme.

#### 3.2.1.1 JSON Dokument

JSON dokument je informativní zdroj. Jedná se o obyčejný JSON.

#### 3.2.1.2 Instance

Instance je JSON dokument, na který je aplikováno schéma [7].

### 3.2.2 Validace

Nejvíce zajímavá sekce je právě tato. Zde se dozvíme, co všechno můžeme použít, abychom ověřili, zda je daný JSON (JSON Dokument) validní. Zásadní pro validaci jsou validační klíčová slova, která se liší podle typu dané instance. Klíčových slov je mnoho, zmíníme jen ta, které se používají nejvíce.

#### 3.2.2.1 Jakýkoliv typ

Validační klíčová slova pro **jakýkoli typ** instance:

1. **type** - viz. 3.1.1.1.
2. **enum** - hodnotou je pole alespoň s jedním prvkem, prvky v poli by měly být unikátní, validace je úspěšná, pokud se hodnota instance rovná jednomu z elementů v tomto klíčovém poli hodnot.

3. **const** - hodnotou může být jakýkoli typ včetně null, validace je úspěšná, pokud se hodnota instance rovná hodnotě klíčového slova.

#### 3.2.2.2 Numerická validace

Validační klíčová slova pro **numerické** instance:

1. **multipleOf** - hodnotou musí být číslo větší než 0, numerická instance je validní jen tehdy, když násobení hodnotou klíčového slova je integer.
2. **maximum** - hodnotou je číslo, jestliže instance je číslo, tak toto klíčové slovo ji validuje jen tehdy, když instance je menší nebo rovna maximu.
3. **exclusiveMaximum** - hodnotou je číslo, jestliže instance je číslo, tak toto klíčové slovo ji validuje jen tehdy, když instance je menší než maximum.
4. **minimum** - hodnotou je číslo, jestliže instance je číslo, tak toto klíčové slovo ji validuje jen tehdy, když instance je větší nebo rovna minimu.
5. **exclusiveMinimum** - hodnotou je číslo, jestliže instance je číslo, tak toto klíčové slovo ji validuje jen tehdy, když instance je větší než minimum.

#### 3.2.2.3 String validace

Validační klíčová slova pro **Stringy**:

1. **maxLength** - hodnotou musí být kladný integer, validací projde string, jehož délka hodnoty je menší nebo rovna maxLength.
2. **minLength** - hodnotou musí být kladný integer, validací projde string, jehož délka hodnoty je větší nebo rovna maxLength.
3. **pattern** - hodnotou je string, validací projde string, jehož hodnota vyhovuje zadanému regulárnímu výrazu.

#### 3.2.2.4 Validace pole

Validační klíčová slova pro **pole**:

1. **maxItems** - hodnotou musí být kladný integer, validací projde pole, jehož délka je menší nebo rovna maxItems.
2. **minItems** - hodnotou musí být kladný integer, validací projde pole, jehož délka hodnoty je větší nebo rovna minItems.
3. **contains** - hodnotou je validní JSON Schema, validací projde pole právě tehdy, když alespoň jeden z jeho elementů je validní proti danému schématu.

#### 3.2.2.5 Validace objektu

Validační klíčová slova pro **objekt**:

1. **maxProperties** - hodnotou musí být kladný integer, validací projde objekt, jehož počet properties je menší nebo roven maxProperties.
2. **minProperties** - hodnotou musí být kladný integer, validací projde objekt, jehož počet properties je větší nebo roven maxProperties.
3. **required** - hodnotou musí být pole, elementy pole, pokud existují, musí být string a unikátní. Objekt je validní, pokud každá položka v poli je jméno property v instanci.
4. **properties** - hodnota properties musí být objekt a každá hodnota tohohle objektu musí být validní JSON Schema.

#### 3.2.2.6 Definované formáty

V JSON Schema máme i definované formáty, například pro string instance:

1. **Datum a čas** - atributy: date-time, date, time.
2. **Emailová adresa** - atributy: email, idn-email.
3. **Hostname** - atributy: hostname, idn-hostname.
4. a další.

Tyto atributy validují stringové instance, např. že string: ahoj@nevim.cz je validní formát emailu [8].

### 3.2.3 Hyper-schéma

„JSON Hyper-schéma je slovník JSON Schema pro anotování dokumentů JSON s hypertextovými odkazy a instrukcemi pro zpracování a manipulaci vzdálených zdrojů JSON prostřednictvím hyper-media prostředí, jako je HTTP.“ [9]

Zde 3.6 můžeme vidět jednoduchý příklad hyper-schéma, kde klíčové slovo „links“, které je polem, obsahuje jeden objekt - s relací IANA registrovaného typu ”self”, která je vytvořena z instance s jedním známým objektovým polem pojmenovaným „id“. Instance, která vypadá takto: „id“: 2402 a jeho základní URI - „base“ je „https://api.priklad.cz/“, pak „https://api.priklad.cz/item/2402“ je výsledný link cílového URI [9].

```
{
    "type": "object",
    "properties": {
```

```
        "id": {
            "type": "number",
            "readOnly": true
        }
    },
    "base": "https://api.priklad.cz/",
    "links": [
        {
            "rel": "self",
            "href": "item/{id}"
        }
    ]
}
```

Listing 3.6: Příklad Hyper-schéma

Chceme-li se dozvědět o tomto tématu více, přečteme si dokumentaci. V rámci této diplomové práci stačí vědět, že hyper-schéma existuje, ale implementovat ho nebudeme.

### 3.2.4 Relativní JSON ukazatelé

„JSON ukazatel je syntaxe sloužící pro specifikování míst v JSON Dokumentu.“[10]. Opět nám stačí vědět, že relativní JSON ukazatel existuje, více si můžeme přečíst v dokumentaci.

#### 3.2.4.1 Syntaxe

Relativní JSON ukazatel je Unicode string, který se skládá z kladného integeru, následuje znak: # nebo JSON ukazatel.

### 3.2.5 Shrnutí

Jak jsme se zde mohli dočíst, tak JSON Schema je šikovný prostředek, který slouží k validaci JSON formátu. Nabízí nám velké množství klíčových slov, podle kterých můžeme ověřit, zda vstupní JSON splňuje všechny požadavky, které jsou na něj kladeny. Výhodou je, že JSON Schema má formu JSONu, tudíž se v něm lze orientovat, ale s přibývajícími klíčovými slovy ztrácí přehlednost.

Kapitola 4

# Transformace mezi formáty JSON a XML

Abychom byli pro formát JSON schopni využívat X-definice, které potřebují vstupní data ve formátu XML, je nutné zařídit bezztrátovou obousměrnou transformaci mezi těmito formáty. Při bližším seznámením s formáty JSON, XML a jejich základní syntaxí si můžeme všimnout několika odlišností, které je třeba akceptovat a správně zakomponovat do návrhu kýžené bezztrátové transformace.

## 4.1 JSON vs XML

### 4.1.1 JSON

Formát JSON je formát, který je využíván k ukládaní a k výměně dat [11].

#### 4.1.1.1 Syntaxe

Syntaxe formátu JSON se řídí těmito základními pravidly:

1. Data v JSON jsou páry jméno/hodnota.
2. Data jsou oddělena čárkou.
3. Složené závorky udržují objekt.
4. Hranaté závorky představují pole.
5. Klíč může obsahovat libovolné Unicode znaky, kde znaky: ", \, / atd. musejí mít před sebou zpětné lomítko, řetězec je velmi podobný Stringu v Javě [12].

V příkladu 4.1 můžeme vidět, jak se vytváří jednoduchý pár, který se skládá jen ze jména a hodnoty. Jméno v JSONu musí být typu string a vždy mezi uvozovkami.

```
"povolani" : "hasic"
```

Listing 4.1: Příklad jednoduchého páru - jméno/hodnota

Hodnoty mohou být typu:

1. string,
2. číslo,
3. JSON objekt,
4. pole,
5. boolean,
6. null.

Hodnoty, které jsou typu string, musí být uvozeny uvozovkami stejně jako v již zmíněném jméně [13].

Hodnoty v poli mohou být z výše zmíněných typů, ale neobsahují páry. Nemůže tedy nastat tato situace4.2.

```
{
 "oceneni" : [ "co" : "hrdina"]
}
```

Listing 4.2: Příklad nevalidního pole

Na příkladu 4.3 můžeme pozorovat objekt složený z několika párů. Tyto páry mají všechny hodnoty, které jsou ve formátu JSON možné. Máme tu jméno „ocenění“, které reprezentuje pole s hodnotami „hrdina“, „hasič roku“. „Manželka“ představuje jméno páru, kde hodnota je objekt, který obsahuje další údaje o objektu.

```
{
 "jmeno": "Josef",
 "prijmeni": "Novak",
 "vek": 25,
 "povolani" : "hasic",
 "muz" : true,
 "dite" : null,
 "oceneni" : [ "hrdina", "hasic roku"],
 "manzelka" : { "jmeno" : "Eva", "prijmeni" : "Novakova" }
}
```

Listing 4.3: Příklad objektu obsahující veškeré zmíněné hodnoty

### 4.1.2 XML

Formát XML byl navržen k uložení a transformaci dat[11].

#### 4.1.2.1 Syntaxe

Syntaxe formátu XML je vcelku logická a jednoduchá. Dokument XML obsahuje XML elementy a musí obsahovat jeden kořenový element, který je rodičem všech ostatních elementů. XML element je vše od začátku tagu včetně až do koncového tagu elementu [14].

```
<povolani>hasic</povolani>
```

Listing 4.4: Příklad jednoduchého XML elementu

Element může obsahovat:

1. text,
2. atributy,
3. elementy,
4. nebo mix zmíněných.

Jméno elementu se může skládat z:

1. písmen,
2. číslic,
3. pomlček,
4. podtržítek,
5. teček [15].

V 4.5 lze vidět, jak takový komplexnější XML element může vypadat. Element „dítě“ - neobsahuje nic mezi tagy. Říkáme tomu prázdný element, který lze zapsat i takto: <dite/>. Kořenovým elementem je zde element „osoba“, která je rodičem ostatních elementů „jmeno“, „prijmeni“ atd..

```
<osoba>
  <jmeno>Josef</jmeno>
  <prijmeni>Novak</prijmeni>
  <vek>25</vek>
  <povolani typ="hlavni">hasic</povolani>
  <dite></dite>
  <manzelka>
        <jmeno>Eva</jmeno>
        <prijmeni>Novaka</prijmeni>
```

```
  </manzelka>
</osoba>
```

Listing 4.5: Příklad komplexnějšího XML elementu

### 4.1.3 Rozdíly

Hlavní rozdíly mezi oběma formáty jsou:

1. JSON absence tagů x XML tagy má,
2. JSON objekt x XML element,
3. JSON string hodnoty v uvozovkách x XML text bez uvozovek,
4. JSON nemá atributy x XML atributy má,
5. JSON nemá namespace x XML namespace má,
6. JSON triviální hodnoty mohou být string, číslo, boolean, null x v XML pouze text.
7. JSON klíče se skládají z libovolných Unicode znaků (před speciálními znaky musí být zpětné lomítko) x jména XML elementů se nemohou skládat z libovolných znaků, např. znak „&“ - není povolen.

Formát JSON, na rozdíl od XML, nemá typické tagy. Je proto pro lidské oko lépe čitelný, jelikož tagy značně narušují čitelnost. Dále JSON obsahuje pole - array, které v XML není implementováno.

## 4.2 Dosavadní konvertory

Před samotným návrhem je dobré se podívat, jak fungují dosavadní konvertory mezi formáty JSON a XML, které jsou k dispozici. Zda splňují bezztrátovost a jak vlastně fungují. Konkrétně se zaměříme na převod z formátu JSON do XML, který je obtížnější z důvodu absence polí a objektů JSON v XML. JSON, na kterém budeme testovat převod do XML formátu, je k vidění zde 4.6.

```
{
 "jmeno": "Alena",
 "prijmeni": "Preveditelna",
 "vek": 25,
 "povolani" : "ucetni",
 "muz" : false,
 "dite" : null,
 "kocky" : [ "Mnau", "Kocka"],
 "manzel" : { "jmeno" : "Marek", "prijmeni" : "Preveditelny" }
}
```

Listing 4.6: JSON, který budeme převáděn do XML

### 4.2.1 http://convertjson.com

#### 4.2.1.1 JSON → XML

Tento konvertor převede náš JSON do formátu XML takto:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <jmeno>Alena</jmeno>
  <prijmeni>Preveditelna</prijmeni>
  <vek>25</vek>
  <povolani>ucetni</povolani>
  <muz>false</muz>
  <dite/>
  <kocky>Mnau</kocky>
  <kocky>Kocka</kocky>
  <manzel>
    <jmeno>Marek</jmeno>
    <prijmeni>Preveditelny</prijmeni>
  </manzel>
</root>
```

Listing 4.7: Výsledné XML z convertjson.com

Můžeme si všimnout, že konvertor ze složených závorek vytvořil pomocný element „root“ který, jak jsme tu již zmiňovali, je povinný a je rodičem

ostatních elementů. S JSON polem „kočka“ si poradil tak, že element „kočka“ vytvořil tolikrát, kolik má pole prvků. Pár s hodnotou null převedl na prázdný element.

#### 4.2.1.2 XML → JSON

Zkusíme tedy převést jejich výsledný XML dokument zpět do formátu JSON.

```
{
    "root": {
        "jmeno": "Alena",
        "prijmeni": "Preveditelna",
        "vek": "25",
        "povolani": "ucetni",
        "muz": "false",
        "dite": "",
        "kocky": [
            "Mnau",
            "Kocka"
        ],
        "manzel": {
            "jmeno": "Marek",
            "prijmeni": "Preveditelny"
        }
    }
}
```

Listing 4.8: Výsledný JSON z convertjson.com

Na první pohled je patrné, že výsledný JSON se liší od našeho originálního, tudíž transformace není bezztrátová. Rozebereme si tedy v čem jsou odlišné:

1. Uměle vytvořený element root z XML se převedl na JSON objekt se jménem „root“, který nebyl součástí původního JSONu.

2. Prázdný element „dítě“ převedl na prázdný string.

Co ale převedl správně, je pole „kočka“, které má správný počet prvků.

#### 4.2.1.3 Závěr

Tento konvertor rozhodně není bezztrátový. S pomocnými elementy, které si vytvořil, si není schopen poradit a začleňuje je dál do výsledného JSONu, který se kvůli tomu liší od původního.

### 4.2.2 http://www.utilities-online.info

#### 4.2.2.1 JSON → XML

```
<?xml version="1.0" encoding="UTF-8" ?>
        <jmeno>Alena</jmeno>
        <prijmeni>Preveditelna</prijmeni>
        <vek>25</vek>
        <povolani>ucetni</povolani>
        <muz>false</muz>
        <dite />
        <kocky>Mnau</kocky>
        <kocky>Kocka</kocky>
        <manzel>
                <jmeno>Marek</jmeno>
                <prijmeni>Preveditelny</prijmeni>
        </manzel>
```

Listing 4.9: Výsledné XML z utilities-online.info

S převodem si tento konvertor poradil takřka stejně jako předchozí, až na jeden zásadní rozdíl a to je absence kořenového elementu. Tudíž se nejedná o validní XML dokument a výsledek je chybný.

#### 4.2.2.2 XML → JSON

Pokusíme se převést chybný XML dokument zpátky do JSONu. Jak můžeme vidět zde 4.10, konvertor hlásí chybu, kvůli absenci kořenového elementu.

```
{
  "parsererror": {
    "#text": "Chyba parsování XML:
Nesmysly za kořenovým prvkem dokumentu
Adresa:
http://www.utilities-online.info/xmltojson/#.XIpFZjEqXIV
Řádek 3, sloupec 2:",
    "sourcetext": "      <prijmeni>Preveditelna</prijmeni>
________^"
  }
}
```

Listing 4.10: Chybová hláška při pokusu převést nevalidní XML dokument zpět do formátu JSON z utilities-online.info

#### 4.2.2.3 Validní XML → JSON

Zkusíme doplnit kořenový element, abychom alespoň zjistili, jak si konvertor poradí s ostatními elementy. Upravený XML dokument je k vidění zde 4.11.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
        <jmeno>Alena</jmeno>
        <prijmeni>Preveditelna</prijmeni>
        <vek>25</vek>
        <povolani>ucetni</povolani>
        <muz>false</muz>
        <dite />
        <kocky>Mnau</kocky>
        <kocky>Kocka</kocky>
        <manzel>
                <jmeno>Marek</jmeno>
                <prijmeni>Preveditelny</prijmeni>
        </manzel>
</root>
```

Listing 4.11: Validní XML pro převod utilities-online.info

```
{
  "root": {
    "jmeno": "Alena",
    "prijmeni": "Preveditelna",
    "vek": "25",
    "povolani": "ucetni",
    "muz": "false",
    "kocky": [
      "Mnau",
      "Kocka"
    ],
    "manzel": {
      "jmeno": "Marek",
      "prijmeni": "Preveditelny"
    }
  }
}
```

Listing 4.12: Výsledný JSON po upraveném XML z utilities-online.info

Přehlédneme-li ve výsledném JSONu 4.12 námi doplněný kořenový element, můžeme si všimnout, že zcela chybí prázdný element „dite“. Tento element se v původním JSONu objevil, ve výsledném již nikoliv. Ostatní elementy jsou převedeny správně.

#### 4.2.2.4 Závěr

Zmíněný konvertor není kvalitně vytvořen. Je absolutně nepřijatelné, aby výsledkem při transformaci z JSONa do XML byl nevalidní XML dokument, i když původní JSON je validní. Vynechání prázdného elementu při převodu do formátu JSON také není zcela správným řešením, protože tím přicházíme o bezztrátovost, která je požadována.

### 4.2.3 https://www.freeformatter.com

#### 4.2.3.1 JSON → XML

Již na hlavní stránce nám tento konvertor nabízí, abychom si nastavili, jak se bude jmenovat kořenový element a element jednotlivých prvků JSON pole. Nechala jsem defaultní nastavení: kořenový element - „root“, element jednotlivých prvků v poli - „element“.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <dite null="true" />
    <jmeno>Alena</jmeno>
    <kocky>
        <element>Mnau</element>
        <element>Kocka</element>
    </kocky>
    <manzel>
        <jmeno>Marek</jmeno>
        <prijmeni>Preveditelny</prijmeni>
    </manzel>
    <muz>false</muz>
    <povolani>ucetni</povolani>
    <prijmeni>Preveditelna</prijmeni>
    <vek>25</vek>
</root>
```

Listing 4.13: Výsledné XML z www.freeformatter.com

Výsledné XML 4.13 má přeházené prvky a pomocné elementy navíc. Rozdíl oproti dvěma předcházejícím konvertorům je takový, že obsahuje element „dítě“, který má atribut null s hodnotou „true“.

#### 4.2.3.2 XML → JSON

I zde máme na výběr nastavit některé parametry - prefix atributů atd., ponecháme nastavení, která tam jsou.

```
{
    "dite": null,
```

```
    "jmeno": "Alena",
    "kocky": [
        "Mnau",
        "Kocka"
    ],
    "manzel": {
        "jmeno": "Marek",
        "prijmeni": "Preveditelny"
    },
    "muz": "false",
    "povolani": "ucetni",
    "prijmeni": "Preveditelna",
    "vek": "25"
}
```

Listing 4.14: Výsledný JSON z www.freeformatter.com

Na první pohled má výsledný JSON 4.14 proházené páry, které souvisí s přeházeným XML, ze kterého se tento JSON převáděl. Po bližším prozkoumání zjistíme, že původní i tento JSON se ve všem shodují. Výsledný JSON pozbývá pomocné elementy „root“ a „elementy“, které úspěšně převedl na JSON objekt a prvky JSON pole.

#### 4.2.3.3 JSON → XML

Vyzkoušíme si, jak si poradí s JSONem 4.15, který má v klíči znaky, které nejsou povolené ve jméně elementu v XML.

```
{
    "dite&pes": null
}
```

Listing 4.15: JSON, který má v klíči nepovolený znak pro jméno v XML elementu

Transformátor nám hlásí toto: „Unable to parse any JSON input. 0x26 is not a legal NCName character“.

#### 4.2.3.4 Závěr

Tento konvertor se zpočátku zdál být obousměrně bezztrátový, ale nezvládá převést JSON, který v klíči obsahuje speciální znaky, které nejsou povolené ve jméně XML elementu. Pomáhá si pomocnými elementy, které ale při převodu dál umí vynechat a převést do správného tvaru. Jako jediné mínus se jeví to, že při transformaci XML na JSON si už nemůžeme nastavit, jak máme pojmenovaný kořen či prvky pole. Pokus jsem zopakovala s jinak pojmenovanými pomocnými elementy a výsledek byl stejný - bezztrátový.

### 4.2.4 Altova

Jedná se o aplikaci, kterou lze bezplatně využívat po dobu třiceti dnů. Tato aplikace umí transformovat z formátu JSON do formátu XML a naopak. Také umí obousměrně transformovat XML schéma do JSON schéma.

#### 4.2.4.1 JSON → XML

Při převodu necháme všechno zaškrtnuto tak, jak je. Konvertor nám dává možnost zvolit si jméno elementu, které bude reprezentovat prvek pole.

```
<?xml version="1.0"?>
<ajson:json
 xmlns:ajson="http://www.altova.com/json"
 ajson:type="object">
  <jmeno ajson:type="string">Alena</jmeno>
  <prijmeni ajson:type="string">Preveditelna</prijmeni>
  <vek ajson:type="number">25</vek>
  <povolani ajson:type="string">ucetni</povolani>
  <muz ajson:type="bool">false</muz>
  <dite ajson:type="null">null</dite>
  <kocky ajson:type="array">
    <item ajson:type="string">Mnau</item>
    <item ajson:type="string">Kocka</item>
  </kocky>
  <manzel ajson:type="object">
    <jmeno ajson:type="string">Marek</jmeno>
    <prijmeni ajson:type="string">Preveditelny</prijmeni>
  </manzel>
</ajson:json>
```

Listing 4.16: Výsledné XML z Altova

Výsledný XML dokument 4.16 dodržuje pořadí JSON dat, ze složených závorek vytvořil pomocný element „json“ s prefixem „ajson“. Jednotlivá jména JSON párů transformoval na elementy s příslušnými jmény. Dále do atributu „type“ vložil typ, který představuje hodnota v páru. To znamená, že JSON objekt pod atributem „type“ má hodnotu „object“, JSON pole tam má „array“ a triviální typy - „string“, „number“, „null“, „bool“. Mezi tagy u triviálních dat je hodnota JSON páru. Prvky JSON pole převádí na elementy se jménem „item“, které jsme si zvolili.

#### 4.2.4.2 XML → JSON

Při zvolení transformace se nám objeví okno s možnostmi. Necháme vše zaškrtnuté až na možnost „Use XML Schema type info“.

```
{
        "jmeno": "Alena",
        "prijmeni": "Preveditelna",
        "vek": 25,
        "povolani": "ucetni",
        "muz": false,
        "dite": null,
        "kocky": [
                "Mnau",
                "Kocka"
        ],
        "manzel": {
                "jmeno": "Marek",
                "prijmeni": "Preveditelny"
        }
}
```

Listing 4.17: Výsledný JSON z konvertoru Altova

Výsledný JSON 4.17 je naprosto totožný s původním JSONem 4.6.

#### 4.2.4.3 JSON → XML

Opět zkusíme, jak si konvertor poradí s tímto JSONem 4.15.

```
<?xml version="1.0"?>
<dite_x0026_pes
  xmlns:ajson="http://www.altova.com/json"
  ajson:type="null">
   null
</dite_x0026_pes>
```

Listing 4.18: Výsledné XML, kde JSON jméno obsahovalo zakázané znaky pro formát XML

Můžeme vidět zde 4.18, že nepodporovaný znak „&“ se transformoval na hexadecimální číslo s prefixem „_x“ a postfixem „_“.

#### 4.2.4.4 XML → JSON

Zkusíme toto XML 4.18 převést zpět do JSON formátu.

```
{
   "dite&pes": null
}
```

Listing 4.19: Výsledný JSON

Opět můžeme vidět, že výsledný JSON 4.19 je naprosto totožný s původním 4.15.

#### 4.2.4.5 JSON → XML

Abychom si byli jistí funkčností převaděče, zkusíme převést tento JSON 4.20

```
{
    "" : ""
}
```

Listing 4.20: Vstupní JSON

Konvertor nám vrací zcela prázdné XML, které, pokud chceme přeložit zpět do formátu JSON, vypíše chybu „Failed to convert XML.“.

#### 4.2.4.6 Závěr

U tohoto konvertoru můžeme říci, že testované převedl bezztrátově až na poslední JSON 4.20. Poradil si i se speciálními znaky, které XML formát ve jménech neakceptuje.

### 4.2.5 Shrnutí

Ze zmíněného vyplývá, že takřka bezztrátovým konvertorem je Altova konvektor. Inspirací nám může být převádění speciálních znaků a vkládaní typů hodnot do atributu „type“.

## 4.3 Standard pro převod

Při navrhování řešení je doporučeno zjistit zda neexistuje nějaký standard, který popisuje, jak při určitém problému postupovat. Takový standard, který nám říká jak převádět JSON data existuje. Nalezneme ho zde `https://www.w3.org/TR/xslt-30/#json`.

### 4.3.1 Pravidla pro převod z JSONu do XML

Z výše uvedené stránky můžeme zjistit, jaká pravidla jsou použita při převodu JSON objektu do XML, která nám zajistí bezztrátovost. My se na ně teď společně podíváme.

1. JSON hodnota null se převede na element pojmenovaný „null“ s prázdným obsahem.
2. JSON hodnoty true nebo false se převedou na element se jménem „boolean“ s obsahem korespodujícím s typem xs:boolean.
3. JSON hodnota number se převede na element pojmenovaný „number“ s obsahem korespodujícím s typem xs:double s omezením, že obsah nesmí být negativní, či pozitivní, nekonečno nebo NaN.
4. JSON hodnota string se převede na element pojmenovaný „string“ s obsahem korespodujícím s typem xs:string.
5. JSON hodnota pole se převede na element pojmenovaný „array“, obsahem je sekvence child elementů, které reprezentují členy pole v pořadí. Na každého člena pole jsou aplikována tato pravidla rekurzivně.
6. JSON objekt se převede na element pojmenovaný „map“, obsahem je sekvence child elementů, kde každý child element reprezentuje pár - jméno a hodnota v objektu. Reprezentace páru - jméno a hodnota je získána tak, že se použije element, který reprezentuje hodnotu (aplikováním těchto pravidel rekurzivně), a přidání atributu „key“, jehož hodnota je jméno v páru a je instanci xs:string [16].

#### 4.3.1.1 Příklad převodu

Z našeho již vytvořeného JSONu 4.6 by nám mělo vzniknout toto 4.21 XML dokument, kde transformace postupuje přesně podle zmíněných pravidel.

```
<map xmlns="http://www.w3.org/2005/xpath-functions">
        <string key="jmeno">Alena</string>
        <string key="prijmeni">Preveditelna</string>
        <number key="vek">25</number>
        <string key="povolani">ucetni</string>
```

```
        <boolean key="muz">false</boolean>
        <null key="dite"/>
        <array key="kocky">
                <string>Mnau</string>
                <string>Kocka</string>
        </array>
        <map key="manzel">
                <string key="jmeno">Marek</string>
                <string key="prijmeni">Preveditelny</string>
        </map>
</map>
```

Listing 4.21: Příklad převedeného JSONu do XML podle pravidel standardu

### 4.3.2 Pravidla pro převod z XML s JSON elementy do JSONu

Máme-li k dispozici XML dokument, který je rozšířen o JSON elementy: map, string, number, array, null atd., jsou pravidla pro tento převod následující:

1. Uzel dokumentu s jedním podřízeným uzlem potomka je zpracován zpracováním tohoto potomka.

2. Element pojmenovaný „null“ má za výsledek výstup null.

3. Element pojmenovaný „boolean“ má za výsledek výstup true nebo false, záleží na obsahu elementu.

4. Element pojmenovaný „number“ má za výsledek výstup typu string s obsahem elementu.

5. Element pojmenovaný „string“ má za výsledek výstup typu string s obsahem elementu, který je uvozen uvozovkami.

6. Element pojmenovaný „array“ má za výsledek výstup potomků tohoto elementu, každý je zpracován rekurzivně podle těchto pravidel.

7. Element pojmenovaný „map“ má za výsledek výstup sekvence prvků mapy, které korespondují s potomky tohoto elementu. Sekvence je uzavřena mezi složenými závorkami a oddělena čárkou. Každý prvek obsahuje hodnotu „key“ atributu child elementu, uzavřeného mezi uvozovkami. Následuje přidání dvojtečky a nakonec výsledek zpracování každého child elementu aplikováním těchto pravidel rekurzivně [16].

### 4.3.3 Pravidla pro převod z obyčejného XML do JSONu

Nemáme-li k dispozici XML dokument s rozšířením JSON, máme podle standardu dvě možnosti:

1. převést obyčejný XML dokument do XML dokumentu rozšířeného o JSON a následně aplikovat již známou transformaci dle pravidel,
2. transfomovat XML do JSON formátu přimo použítím vlastních pravidel šablony. [16].

### 4.3.4 Závěr

Výše zmíněná pravidla nám velmi dobře definují převod z jednoho formátu do druhého a naopak. Využijeme je tedy při tvorbě našeho návrhu.

## 4.4 Návrh

V první řade bude potřeba upravit stávající návrh tak, aby byl schopen transformovat jednotlivé formáty mezi sebou s ohledem na pravidla standardu. Při použití pravidel při převodu z JSON do XML nám vzniká XML rozšířené o formát JSON (dále jen „XML“). Níže zmíněnými typy Object, Map a List, máme na mysli ty, které patří do balíku **java.util** a **java.lang**.

### 4.4.1 Návrh při převodu z JSON → XML

Současný návrh převádí JSON do typu Object, který pak postupně zpracovává a převádí typ Object do formátu XML. Tohoto převodu využijeme a upravíme výsledné zpracování typu Object na elementy XML.

#### 4.4.1.1 JSON → Object

První fází je schopnost převést JSON na Object, toto nám značně ulehčí následnou transformaci do formátu XML. Object je datový typ, který získáme zpracováním příslušného JSONa. Na výsledku 4.22 můžeme vidět, jak vypadá objektová reprezentace JSONu 4.6. Také si můžeme všimnout, že prvky nejdou za sebou tak, jako v původním JSONu.

```
{dite=null, jmeno=Alena, kocky=[Mnau, Kocka],
manzel={jmeno=Marek, prijmeni=Preveditelny},
muz=false, povolani=ucetni,
prijmeni=Preveditelna, vek=25}
```

Listing 4.22: Příklad vypsání převedeného JSONu do Objectu metodou toString()

Jednotlivé páry jsou reprezentovány jako „jmeno=hodnota“ a odděleny čárkou. Pokud je hodnota netriviální - jedná se o JSON objekt nebo JSON pole, přidá se příslušný znak na začátek i konec. U JSON objektu jde o složené závorky, u pole jsou to závorky hranaté. Rozdíl mezi původním JSONem a touto objektovou reprezentací je jen náhrada dvojtečky za rovnítko. Jedná se ovšem jen o výpis. Vnitřní struktura je reprezentována jako vnoření typů Object - JSON objekt je reprezentován jako Map <String, Object> a JSON pole je reprezentováno pomocí List<Object>.

Při zpracování formátu JSON, bychom měli brát v potaz, že uživatel může kdekoli v JSONu vložit komentáře „/* */“.

#### 4.4.1.2 Object → XML

Další fází je převedení výsledného Objectu, který vznikl ze vstupního JSONu, do výsledného XML. Object může být reprezentován mapou či listem. Záleží na tom zda vstupní JSON byl JSON objekt či JSON pole. Mapu či list budeme postupně procházet a vytvářet z nich příslušné elementy tak, abychom

dodržovali pravidla. To znamená, že z hlavní mapy, kde pod klíčem „jmeno“ je hodnota „Alena“, vytvoříme tento element 4.23.

Zpracování JSON klíčů, které obsahují nepovolené znaky pro jména XML elementů, je v současné verzi vyřešeno načítáním jednotlivých znaků klíče. Jedná-li se o nepovolený znak, přetransformuje se na hexadecimální zápis tohoto znaku a před něj se přidá „_u“ a z něj se přidá „_“.

```
<string key=jmeno>Alena</string>
```

Listing 4.23: Příklad vytvoření elementu XML z Objectu

Výsledkem je XML rozšířené o formát JSON.

### 4.4.2 Návrh při převodu z XML → JSON

Nyní bude následovat rozšíření a úprava stávajícího návrhu tak, aby byl schopen převádět XML do JSONu. Zejména je nutné upravit jej tak, aby zpracovával rozšířené XML o formát JSON.

Převod obyčejného XML bez pojmenovaných elementů: <map>, <array>, <string> atd. nás nemusí zajímat, jelikož při rozšiřování frameworku X-definice o zpracování formátu JSON se pracuje se vstupem, který je ve formátu JSON.

#### 4.4.2.1 XML → Object

První fází převodu je zpracování jednotlivých elementů a vytvoření výsledných Objectů, které mohou být Map<String,Object> nebo List<Object>.

Při převodu se orientujeme podle jména elementu. V tomto případě bude podle 4.24 výsledný Object je Mapa, která má jeden prvek, kde klíč této mapy je „clovek“ a hodnota pod tímto klíčem je Object - Map, který se skládá z klíče „jmeno“ a hodnoty „Alena“.

Obsahuje-li v současné verzi XML u jmen elementů speciální znaky s prefixem „_u“, převedou se tyto znaky do své původní podoby.

```
<map>
        <map key=clovek>
                <string key=jmeno>Alena</string>
        </map>
</map>
```

Listing 4.24: Příklad rozšířeného XML

#### 4.4.2.2 Object → JSON

Posledním úkolem je převod výsledného Objectu do formátu JSON. Výsledný Object může být instancí typu Map nebo List. Převodu docílíme tak, že zjistíme pod kterou instanci Object patří a postupně procházíme jeho strukturu.

## 4.5 Implementace

Z návrhu vyplývá, co vše je potřeba upravit a naprogramovat v původní verzi frameworku. Hlavní funkcionality jsou:

1. parsování JSON, jehož výsledkem je objekt typu Object,
2. převedení objektu typu Object do XML,
3. převedení XML do objektu typu Object,
4. převedení objektu typu Object do JSON.

### 4.5.1 Převod JSON → Object

Parsování JSONu v původní verzi je zajištěno metodou **parseJSON** ve třídě **JSONUtil**. Metoda postupně načítá vstup - JSON typu String, ze kterého vytváří výsledný objekt, který je instancí typu Map nebo List. Byl kladen požadavek, aby se bez problémů převedl i JSON s komentáři 4.25. A jelikož původní verze toto neumí, musela být implementována metoda **isWhitespace** 4.26, která ignoruje mezery i komentáře. Ve výsledném objektu se komentáře nevyskytují a převod proběhne bez chyb.

```
{
 "jmeno": "Josef", /* zde zadejte jmeno */
 "prijmeni": "Novak",
 "vek": 25,
 "povolani" : "hasic",
 "muz" : true,
 "dite" : null, /* jestli je bezdetny, tak null */
 "oceneni" : [ "hrdina", "hasic roku"],
 "manzelka" : { "jmeno" : "Eva", "prijmeni" : "Novakova" }
}
```

Listing 4.25: Příklad JSONu s komentářem

```
public final boolean isWhitespace() {
        boolean result = isSpaces();
        if (isToken("/*")) {
                result = true;
                if (!findToken("*/")) {
                throw new SRuntimeException("Unclosed comment "
                                + "in the script");
                }
                else {
                                nextChar();
                                nextChar();
```

```
                                        isSpaces();
                    }
            }
            return result;
}
```

Listing 4.26: Metoda isWhitespace()

V původní verzi nenásledovaly prvky za sebou tak jako ve vstupním JSONu 4.22. K nápravě došlo změnou z typu TreeMap na LinkedHashMap. Výsledný objekt v upravené verzi vypadá nyní takto 4.27.

```
{jmeno=Alena, prijmeni=Preveditelna,
vek=25, povolani=ucetni, muz=false,
dite=null, kocky=[Mnau, Kocka],
manzel={jmeno=Marek, prijmeni=Preveditelny}}
```

Listing 4.27: Výsledný objekt v aktuální upravené verzi

#### 4.5.1.1 TreeMap vs. LinkedHashMap

V **TreeMapě** se iteruje podle přirozeného pořadí klíčů v rámci jejich compareTo metody - podle abecedy [17], oproti tomu se v **LinkedHashMap** iteruje podle pořadí vložení jednotlivých párů. A to je náš záměr [18].

### 4.5.2 Převod Object → XML

Tento převod je v původní verzi řešen metodou **jsonToXml** ve třídě **JSONUtil**, která jako vstup požaduje JSON typu Object a vrací nám Element. Jak již bylo řečeno při návrhu, je nutné upravit metody tak, aby vstup byl transformován dle pravidel standardu.

Správný převod JSON klíčů, bez speciálních znaků, které vadí formátu XML, je zajištěn metodou **toXmlName**.

Na příkladu 4.28 můžeme vidět výsledný JSON transformovaný z 4.6. Nyní si rozebereme, jak původní transformátor převádí jednotlivé JSON data do výsledného XML.

1. Jednoduchá data, která nejsou JSON objekt či pole, převádí do elementu, kde jméno elementu je jméno v páru, obsahem je hodnota páru.

2. Jedná-li se JSON data, kde hodnota je pole, transformátor vytvoří element se jménem páru, dále se přidá element „js:array“, kde jednotlivé prvky pole jsou převedeny na elementy „js:item“ s hodnotou jednotlivých prvků.

3. Reprezentuje-li hodnota páru JSON objekt a data jsou:

a) triviální - transformace je následující: jméno v páru se převede na element s příslušným jménem a každá položka objektu se převede na atribut, který reprezentuje jméno v dané položce a jeho hodnota je opět hodnotou v páru dané položky.

b) netriviální - JSON objekt, pole, vytvoří se pomocný element „js:mapItems“, který obsahuje elementy netriviálních dat viz. 4.29

```
<js:map xmlns:js="http://www.syntea.cz/json/1.0">
        <dite/>
        <jmeno>Alena</jmeno>
        <kocky>
                <js:array>
                        <js:item>Mnau</js:item>
                        <js:item>Kocka</js:item>
                </js:array>
        </kocky>
        <manzel jmeno="Marek" prijmeni="Preveditelny"/>
        <muz>false</muz>
        <povolani>ucetni</povolani>
        <prijmeni>Preveditelna</prijmeni>
        <vek>25</vek>
</js:map>
```

Listing 4.28: Příklad převodu JSON na XML - původní verze před úpravou

```
<manzel jmeno="Marek" prijmeni="Preveditelny">
  <js:mapItems>
        <auto>
                <js:array>
                        <js:item>Ford</js:item>
                        <js:item>BMW</js:item>
                </js:array>
        </auto>
        <pes jmeno="Fido"/>
  </js:mapItems>
</manzel>
```

Listing 4.29: Příklad převodu pojmenovaného JSON objektu, kdy objekt obsahuje netriviální data - původní verze před úpravou

Vidíme tedy, že nám stačí upravit původní verzi zpracování takto:

1. Triviální data převedeme na elementy, kde názvy mohou být „js:string“, „js:boolean“, „js:null“, „js:number“, podle typu hodnoty v páru. A doplníme atribut „js:key“, který bude mít hodnotu jména páru, je-li triviální

hodnota prvkem pole, atribut vynecháváme. Element bude mít mezi tagy hodnotu daného páru.

2. Netriviální data převedeme na elementy, kde názvy mohou být „js:array“ - hodnota je pole, „js:map“ - hodnota je JSON objekt. A doplníme atribut „js:key“, který bude mít hodnotu jména páru, jedná-li se o prvek pole, atribut vynecháme. Element bude mít mezi tagy jednotlivé prvky mapy nebo pole, které se zpracují podle těchto pravidel rekurzivně. Výsledkem implementace upravené transformace je toto XML 4.30.

```
<js:map xmlns:js="http://www.xdef.org/json/1.0">
        <js:string js:key="jmeno">Alena</js:string>
        <js:string js:key="prijmeni">Preveditelna</js:string>
        <js:number js:key="vek">25</js:number>
        <js:string js:key="povolani">ucetni</js:string>
        <js:boolean js:key="muz">false</js:boolean>
        <js:null js:key="dite"/>
        <js:array js:key="kocky">
      <js:string>Mnau</js:string>
      <js:string>Kocka</js:string>
        </js:array>
        <js:map js:key="manzel">
      <js:string js:key="jmeno">Marek</js:string>
      <js:string js:key="prijmeni">Preveditelny</js:string>
        </js:map>
</js:map>
```

Listing 4.30: Příklad převodu JSON objektu - nová verze

Můžeme si všimnout, že jednotlivá data respektují pravidla standardu a jsou v takovém pořadí, v jakém jsme je zadali při vstupu.

### 4.5.3 Převod XML → Object

Převod z XML formátu do Objectu obsahující JSON data je zajišťován metodou **xmlToJson** ve třídě **JSONUtil**, kde vstupem je root - kořenový element XML typu Element a výstupem jsou JSON data typu Java Object.

Správný převod XML řetězce do JSON řetězce je zajištěn metodou **toJsonName**.

Původní verze je samozřejmě spjatá s jinými pravidly, než těmi, která se používala při převodu JSONu do XML. Musíme tedy opět upravit jednotlivé metody tak, aby byly schopny zpracovávat náš tvar XML. Výsledkem z XML 4.30 je tento Object 4.27, který je naprosto stejný jako při převodu z formátu JSON do Objectu.

### 4.5.4 Převod Object → JSON

Výsledný Object je převáděn do formátu JSONu metodou **toJSONString**, kde vstupem je Object a výsledkem String s JSON daty. Výsledek je shodný s původním vstupním JSONem 4.6.

## 4.6 Testování

### 4.6.1 Unit testy

Testování transformace z formátu JSON do formátu XML a naopak nalezneme v testovací třídě**test.common.json.TestJSON3.java**. V této třídě testujeme správnost vzájemného převodu. Třída **TestJSON3** volá metodu **check**, která vyžaduje dva vstupní parametry typu String, třídy **test.common.json.TestJSON.java**, tato třída volá metody:

1. Metodu **check** na první parametr, je-li parametr JSON, zavolá metodu **chkj**, v opačném případě zavolá metodu **chkx**, kde vstupním parametrem je první parametr.

2. Metodu **check** na druhý parametr, je-li parametr XML, zavolá metodu **chkx**, v opačném případě zavolá metodu **chkj**, kde vstupním parametrem je druhý parametr.

3. Metody **chkx** nebo **chkj**, které jako vstup vyžadují oba parametry, se zavolají podle toho, jestli první parametr je v JSON nebo v XML formátu. Je-li první parametr JSON, zavolá se **chkj**, v opačném případě se zavolá metoda **chkx**.

Metoda **chkj** s jedním parametrem převede String, který reprezentuje JSON, na **o1** - instanci třídy Object pomocí metody **JSONUtil.parseJSON**. Tato instance se převede na **e1** - instanci třídy Element pomocí metody **JSONUtil.jsonToXml**. Instance **e1** se převede zpátky pomocí metody **JSONUtil.xmlToJson** a vytvoří se instance **o2** třídy Object. Dále se metodou **JSONUtil.jsonEqual** porovnává, zda instance **o1** a **o2** jsou stejné. Jsou-li instance totožné, převede se **o2**, instance třídy Object, na instanci třídy Element **e2** metodou **xmlToJson**. Zda se vytvořené instance třídy Element **e1** a **e2** rovnají, se ověří metodou **JSONUtil.equalJElems**. Metoda **chkj** vrací true, jsou-li instance třídy Object, které vznikly převodem **e1** a **e2**, totožné. Nejsou-li totožné, vrací false.

Metoda **chkx** s jedním parametrem převede String, který reprezentuje XML, na **e1** - instanci třídy Element pomocí metody **KXmlUtils.parseXml**. Tato instance se převede na **o1** - instanci třídy Object pomocí metody **xmlToJson**. Instance **o1** se převede zpátky pomocí metody **jsonToXml** a vytvoří se instance **e2** třídy Element. z této instance se voláním metody **xmlToJson** vytvoří instance třídy Object **o2**. Metoda **chkx** vrací true, jsou-li instance třídy Object, **o1** a **o2**, a instance třídy Element, **e1** a **e2**, totožné.

Metody **chkx** a **chkj**, které vyžadují na vstupu dva parametry, testují, zda převedené vstupní parametry na stejný formát jsou navzájem ekvivalentní.

Ve třídě **TestJSON3** 4.31 můžeme nalézt i testy, které jsou zaměřené na krajní případy - jméno JSON páru obsahuje znak, který není přípustný

v XML atributu. Třída pomocí metody **check** pokrývá všechny metody, které se používají při převodu.

```
public void test() {

assertTrue(check(
 "{\"dite&pes\": null}",

 "<js:map xmlns:js=\"http://www.xdef.org/json/1.0\">"
   + "<js:null js:key=\"dite_u26_pes\"/>"
+ "</js:map>"));

assertTrue(check(
 "{\"\": \"\"}",

 "<js:map xmlns:js=\"http://www.xdef.org/json/1.0\">"
   + "<js:string></js:string>"
 + "</js:map>"));

assertTrue(check(
 "[\"dite&pes\"]",

 "<js:array xmlns:js=\"http://www.xdef.org/json/1.0\">"
  + "<js:string>dite&amp;pes</js:string>"
 + "</js:array>"));
 ...
 }
```

Listing 4.31: Testování v TestJSON3.java

Kapitola 5

# Rozšíření X-definice o zpracování formátu JSON

Při diskuzi s vedoucím této práce, jsme se dohodli, že vytvoříme vlastní formát - J-definice a implementujeme validační mód pro formát JSON.

## 5.1 Návrh

V této práci se má brát zřetel na již zmíněné JSON Schema 3.1. JSON Schema nám při přidávání různých kontrol, například datových typů, přišlo velmi nepřehledné. Proto jsme se rozhodli vytvořit formát J-definice, který si následně převedeme do formátu X-definic. Ty pak zajistí validování.

### 5.1.1 J-definice

J-definice je jednoduchý formát, který popisuje strukturu JSON dokumentů. Je to XML dokument, kde textová hodnota elementu je popis struktury ve formátu JSON - model. Příklad 5.1 kopíruje přesně vstupní JSON 4.6 s tím rozdílem, že hodnoty jsou nahrazeny validačními metodami a kontrolou výskytu.

```
<xd:jdef xmlns:xd="http://www.xdef.org/3.2"
xmlns:js="http://www.xdef.org/json/1.0"
  xd:name="JSON1">
{  "jmeno": "string();",
   "prijmeni": "string();",
   "vek": "optional int(0,130);",
   "povolani" : "string();",
   "muz" : "optional boolean();",
   "dite" : "jnull();",
   "kocky" : [ "xd:script optional;", "string();", "string();"],
   "manzel" : { "xd:script": "optional;",
```

```
                    "jmeno" : "string();",
                    "prijmeni" : "string();"
                    }
}
</xd:jdef>
```

Listing 5.1: Příklad J-definice

#### 5.1.1.1 Syntaxe

U **triviálních hodnot/prvků** typu string, číslo, boolean a null se místo pravé hodnoty vyskytuje sekce skriptu 1.1.3 bez označení „xd:script“, která se nachází mezi uvozovkami. Povinná je validační sekce. Na rozdíl od X-definice u triviálních hodnot **musí být ve validační sekci přítomna kontrola typů**. Chybí-li informace o výskytu, jde o povinný prvek. Jednotlivé sekce jsou od sebe odděleny středníkem.

Na příkladu 5.1 vidíme pár 5.2 s validační sekcí, která se skládá z **kontroly výskytu** - „optional“ a **kontroly typu** - „int(0,130)“. To znamená, že vstupní JSON, který se bude validovat na základě této J-definice, nemusí obsahovat JSON data, kde jméno je „vek“. Ale pokud jej obsahuje, tak hodnota musí být číslo, které je v rozsahu 0 až 130. Validací by tedy prošel JSON bez tohoto páru se jménem „vek“ nebo například tento JSON 5.3. JSON, který by validací neprošel, je například tento 5.4, protože hodnota překračuje stanovenou hranici.

```
{
...,
"vek": "optional int(0,130);",
...
}
```

Listing 5.2: Pár s validační sekcí místo hodnoty

```
{
...,
"vek": 60,
...
}
```

Listing 5.3: JSON, který by prošel úspěšně validací

```
{
...,
"vek": 200,
...
}
```

Listing 5.4: JSON, který by neprošel validací

V **JSON objektu** se **na prvním místě** může objevit pár, kde jméno je „xd:script“. Tento skript se týká tohoto objektu a má stejnou syntaxi a sémantiku jako v X-definicích 1.1.3. Kontrola hodnot se zde nevyskytuje, ale může se tu objevit kontrola výskytu. Chybí-li kontrola výskytu, jde o povinný prvek. U JSON objektu **nezáleží na pořadí párů**.

Zde 5.5 vidíme konkrétní JSON objekt, který obsahuje pár, kde jméno je „xd:script“. Tento skript má jako hodnotu „optional“, to znamená, že JSON objekt se jménem „manzel“ se může či nemusí objevit a přesto projde validací.

```
{
...,
"manzel" : { "xd:script": "optional;",
   ...,
   ...
   }
}
```

Listing 5.5: JSON objekt, který obsahuje pár se jménem xd:script

V **JSON array** se **na prvním místě** může objevit prvek se začátkem „xd:script“. Tento skript se opět týká tohoto objektu a opět následuje syntaxi a sémantiku X-definic 1.1.3. Platí to samé, co u JSON Objektu. U JSON pole **záleží na pořadí prvků**.

Opět můžeme vidět zde 5.6, že JSON array obsahuje prvek „xd:script optional;“, který nám sděluje, že vstupní JSON nemusí toto pole se jménem vůbec obsahovat.

```
{
...,
"kocky" : [ "xd:script optional;", "string();", "string();"],
...;
}
```

Listing 5.6: JSON array, který obsahuje prvek, který začíná slovem xd:script

#### 5.1.1.1.1 Více JSONů

Je-li přítomno více modelů, které popisují JSONy, tak tyto JSONy musí být textovou hodnotou ve **svých elementech** v J-definici. Tento element musí začínat prefixem **js** a za ním následuje jméno modelu. Příklad je k vidění zde 5.7.

```
<xd:jdef xmlns:xd="http://www.xdef.org/3.2"
xmlns:js="http://www.xdef.org/json/1.0"
  xd:name="JSON2">
<js:clovek>
  {
    "jmeno": "string();",
```

```
    "prijmeni": "string();",
    "vek": "optional int(0,130);",
    "kocka" : "xd:script optional; ref js:kocka"
}
</js:clovek>
<js:kocka>
  {
    "jmeno": "string();",
    "vek": "optional int(0,19);"
  }
</js:kocka>
</xd:jdef>
```

Listing 5.7: Příklad J-definice s více modely

#### 5.1.1.1.2 Reference

Reference odkazuje na konkrétní model. U datech v páru, kde hodnotu chceme nahradit konkrétním modelem, zapíšeme do hodnoty:
„xd:script ref *jmeno_modelu_vcetne_prefixu*“, kde do skriptu můžeme přidat kontrolu výskytu i další akce, které jsou od sebe odděleny středníkem viz. 5.7. Chceme-li odkazovat v poli na konkrétní model, prvek bude mít tuto hodnotu: „xd:script ref *jmeno_modelu_vcetne_prefixu*“, tuto hodnotu můžeme opět obohatit o kontrolu výskytů atd..

### 5.1.2 Návrh při převodu z J-definice → X-definice

Stěžejní částí je návrh převodu formátu J-definic do formátu X-definic, které se pomocí známých metod 2.3 postarají o validování. Zde 5.8 můžeme vidět převedenou J-definici 5.1 do XML formátu.

#### 5.1.2.1 J-definice → XML

Nejdříve je nutné získat z J-definice JSON, který má reference nahrazeny konkrétními modely a který si pomocí metody převedeme do XML formátu. Ten je rozšířený o JSON prvky (string, map, array atd.). Návrh nalezneme v kapitole **Transformace mezi formáty JSON a XML** v sekci **Návrh** 4.4.1

```
<js:map xmlns:js="http://www.xdef.org/json/1.0">
  <js:string js:key="jmeno">string();</js:string>
  <js:string js:key="prijmeni">string();</js:string>
  <js:string js:key="vek">optional int(0,130);</js:string>
  <js:string js:key="povolani">string();</js:string>
  <js:string js:key="muz">optional boolean();</js:string>
  <js:string js:key="dite">jnull();</js:string>
  <js:array js:key="kocky">
    <js:string>xd:script optional;</js:string>
    <js:string>string();</js:string>
    <js:string>string();</js:string>
  </js:array>
  <js:map js:key="manzel">
    <js:string js:key="xd:script">optional;</js:string>
    <js:string js:key="jmeno">string();</js:string>
    <js:string js:key="prijmeni">string();</js:string>
  </js:map>
</js:map>
```

Listing 5.8: Příklad XML vyrobeného z J-definice 5.8

#### 5.1.2.2 XML → X-definice

Dále si toto 5.8 XML upravíme do podoby X-definice tak, že:

1. Triviální elementy (js:string, js:number, js:null, js:boolean) přejmenujeme na js:item.
2. Jednotlivé klíče „js:key“ elementů vložíme do atributu „xd:scriptu“ elementu takto: „xd:script="match @js:key=='*hodnota_klice*'";“.
3. Je-li v textové části triviálního elementu (js:string, js:number, js:null, js:boolean) kontrola výskytu, přidáme tuto kontrolu do atributu „xd:scriptu“:

„xd:script=”optional; match @js:key==’*hodnota_klice*’;”“, zbytek ponecháme v textové hodnotě.

4. Má-li mapa dva a více elementů, které nemají klíč „xd:script“, přidáme child element „xd:mixed“ - nezáleží na pořadí párů v mapě, který bude rodičem ostatních child elementů mapy.

5. Vyskytuje-li se pod mapou child element, kde klíčem je „xd:script“, vytvoříme atribut „xd:script“, který vložíme do elementu mapy. Obsah atributu bude totožný s obsahem hodnoty pod daným klíčem. Obsahuje-li child element hodnotu „xd:choice“, vytvoříme child element „xd:choice“, který bude rodičem ostatních child elementů mapy nebo pokud již existuje child element mapy „xd:mixed“, přejmenujeme ho na „xd:choice“.

6. Vyskytuje-li se pod polem child element, kde textová hodnota začíná slovem „xd:script“, vytvoříme atribut „xd:script“, který vložíme do elementu pole, obsah atributu bude totožný se zbylou textovou hodnotou bez „xd:script“.

7. Přidáme element „NamedItem“, kde rodičem je element „xd:def“, který nám sděluje, že hodnota atributu „js:key“ má být string. Tento element pak referencujeme ve všech „xd:scriptů“ ostatních elementů: „ref NamedItem“.

Výslednou X-definici vytvořenou podle pravidel z J-definice 5.8 můžeme vidět zde 5.9.

```
<xd:def xd:name="JSON1"
  xd:root="js:map"
  xmlns:js="http://www.xdef.org/json/1.0"
  xmlns:xd="http://www.xdef.org/3.2">
  <js:map>
    <xd:mixed>
    <js:item xd:script="match @js:key=='jmeno';
     ref NamedItem">
        string();
    </js:item>
    <js:item xd:script="match @js:key=='prijmeni';
     ref NamedItem">
        string();
    </js:item>
    <js:item xd:script="optional ;match @js:key=='vek';
     ref NamedItem">
        int(0,130);
    </js:item>
    <js:item xd:script="match @js:key=='povolani';
```

```
    ref NamedItem">
        string();
    </js:item>
    <js:item xd:script="optional ;match @js:key=='muz';
    ref NamedItem">
        boolean();
    </js:item>
    <js:item xd:script="match @js:key=='dite';
    ref NamedItem">
        jnull();
    </js:item>
    <js:array xd:script="optional; match @js:key=='kocky';
    ref NamedItem">
        <js:item>
            string();
        </js:item>
        <js:item>
            string();
        </js:item>
    </js:array>
    <js:map xd:script="optional; match @js:key=='manzel';
    ref NamedItem">
        <xd:mixed>
            <js:item xd:script="match @js:key=='jmeno';
            ref NamedItem">
                string();
            </js:item>
            <js:item xd:script="match @js:key=='prijmeni';
            ref NamedItem">
                string();
            </js:item>
        </xd:mixed>
    </js:map>
    </xd:mixed>
    </js:map>
    <NamedItem js:key="string();"/>
</xd:def>
```

Listing 5.9: Příklad X-definice vytvořené z J-definice 5.8

### 5.1.3 Návrh při převodu ze vstupního JSONu → XML, kde triviální elementy jsou nahrazené js:item

Nyní nám stačí navrhnout převod vstupního JSONu, který se bude validovat podle J-definice, do formátu XML, kde jména triviálních elementů jsou nahrazena jménem „js:item“. Vstupní JSON je k vidění zde 4.6.

#### 5.1.3.1 JSON → XML

Tento převod již máme navržený v kapitole **Transformace mezi formáty JSON a XML** v sekci **Návrh** 4.4.1. Výsledné XML můžeme vidět zde 4.30.

#### 5.1.3.2 XML → XML, kde triviální elementy jsou nahrazené js:item

Aby nám XML, které vzniklo převodem ze vstupního JSONu, korespondovalo s převedenou J-definicí do X-definice, musíme všechna jména triviálních elementů nahradit jménem „js:item“.

```
<js:map xmlns:js="http://www.xdef.org/json/1.0">
        <js:item js:key="jmeno">Alena</js:item>
        <js:item js:key="prijmeni">Preveditelna</js:item>
        <js:item js:key="vek">25</js:number>
        <js:item js:key="povolani">ucetni</js:item>
        <js:item js:key="muz">false</js:item>
        <js:item js:key="dite"/>
        <js:array js:key="kocky">
                <js:item>Mnau</js:item>
                <js:item>Kocka</js:item>
        </js:array>
        <js:map js:key="manzel">
                <js:item js:key="jmeno">Marek</js:item>
                <js:item js:key="prijmeni">Preveditelny</js:item>
        </js:map>
</js:map>
```

Listing 5.10: Výsledné upravené XML, kde jména triviálních elementů jsou nahrazena jménem „js:item“

## 5.2 Implementace

### 5.2.1 Převod J-definice → X-definice

O tento převod se stará metoda **jDefToXDef** ve třídě **XJUtils2**, která jako vstup požaduje Element, který reprezentuje J-definici. Dále v této metodě potřebujeme z J-definice získat textovou hodnotu, která obsahuje JSON. O toto se stará metoda **getJsonFromJDef**, která kopíruje všechny elementy s prefixem „xd“ do Elementu X-definice (xd:macro, xd:definition atd.), a zároveň vrací HashMapu, kde klíčem je jméno elementu, který obsahuje JSON (je-li přítomen jenom jeden JSON, může být textovou hodnotou elementu „<xd:jdef>“ a klíč je potom prázdný) a hodnotou je textová hodnota daného elementu.

Dále se v **jDefToXDef** řeší reference, kde jejich zpracování zajistí metoda **addReferenceObject**, která nahradí referencovaný objekt skutečným obsahem referencovaného objektu.

#### 5.2.1.1 Převod J-definice → XML

Dalším krokem je převod získané textové hodnoty z elementů, které reprezentují formát JSON, do XML formátu. Tento proces zajistí metoda **jsonToXml** 4.5.2, která na vstup potřebuje Object. Ten reprezentuje JSON, vytvořený pomocí metody **parseJSON** 4.5.1. Obě metody jsou popsány v kapitole **Transformace mezi formáty JSON a XML**, konkrétně v sekci **Implementace** 4.5.

### 5.2.2 Převod XML → X-definice

Máme-li převedený JSON J-definice do formátu XML zavoláme metodu **jd2xd**, která na vstup dostane převedený JSON J-definice ve formátu XML, jméno J-definice, jméno konkrétního elementu, který obsahoval JSON, a počet převáděných JSONů. Tato metoda volá na základě jména elementu metody. Je-li první element „js:map“, zavolá metodu **jdMapToXD**. Je-li element „js:array“, zavolá metodu **jdArrayToXD**. Tyto metody postupně procházejí a zpracovávají vstupní XML (převedený JSON J-definice) podle postupu zmíněného v návrhu 5.1.2.2. Narazí-li na triviální element, zavolají metodu **jdItemToXD**, která se opět řídí postupy v návrhu 5.1.2.2. Narazí-li na netriviální elementy, volají přišlusné metody **jdArrayToXD** nebo **jdMapToXD**. Metodu **jd2xd** nám vrací Element, který reprezentuje X-definici. Tuto metodu můžeme vidět zde 5.11. Výsledkem je XML, které se nyní může validovat podle X-definice.

```
private void jd2xd(final Node xml, final Node parent,
    final String name, final Integer size) {
  Element e = null;
  if (size == 1) {
    e = (Element)parent;
```

```
    }
    else {
      e = appendJSONElem(parent, name);
    }
    if (J_MAP.equals(xml.getLocalName())) {
      jdMapToXD(xml, e);
    }
    if (J_ARRAY.equals(xml.getLocalName())) {
      jdArrayToXD(xml, e);
    }
    if (size != 1) {
      popContext();
    }
}
```

Listing 5.11: Metoda jd2xd

### 5.2.3 Převod vstupního JSONu → XML, kde triviální elementy jsou nahrazené „js:item“

Nyní potřebujeme vstupní JSON převést do formátu XML, kde triviální elementy jsou nahrazené „js:item“, abychom ho mohli validovat podle X-definice, která vznikla převedením J-definice. O tento převod se stará metoda **jsonToXmlWithItems**, která jako vstup vyžaduje JSON typu String, ve třídě **XJUtils2**, tato metoda volá metody, které si popíšeme níže.

#### 5.2.3.1 Převod vstupního JSONu → XML

První volanou metodou je **jsonToXml** 4.5.2, která na vstup potřebuje Object, který reprezentuje JSON, vytvořený pomocí metody **parseJSON** 4.5.1. Obě metody jsou popsány v kapitole **Transformace mezi formáty JSON a XML** konkrétně v sekci **Implementace** 4.5.

#### 5.2.3.2 Převod XML → XML, kde triviální elementy jsou nahrazené „js:item“

Výsledné XML typu Element se převede na typ String a použije se jako vstupní parametr metody **xmlToXmlItem**, která nahradí jména triviálních elementů na „js:item“.

## 5.3 Testování

### 5.3.1 Unit testy

Validaci vstupních JSON formátů podle J-definic testujeme v testovací třídě **test.jdef.Test000.java** 5.12. V této třídě se testuje:

1. validace XML podle X-definice,
2. validace XML podle J-definice,
3. validace JSON podle J-definice.

V testovací třídě můžeme nalézt různé testy na validaci: výskyt, kontrola datových typů, pořadí atd..

```
public void test() {
...

/********* TESTING JDEF and JSON
* Data - 'jmeno' can be missing
* Validation should be without errors ********/

jdef = "<xd:def xmlns:xd=\"http://www.xdef.org/3.2\""
+ "xmlns:js=\"http://www.xdef.org/json/1.0\"\n" +
"  xd:name=\"JSON2\" xd:root=\"js:array\">\n" +
"[\n" +
"  \"date();\",\n" +
"  {\"xd:script\": \"occurs 1..*;\",\n" +
"      \"jmeno\": \"optional string(2, 50);\",\n" +
"      \"plat\":  \"int(1000, 1000000);"
                        + "finally outln('ahoj');\",\n" +
"      \"nar\":   \"date();\"\n" +
"  }\n" +
"]\n" +
"</xd:def>";

...

json = "[\n" +
"  \"2019-03-13\",\n" +
"  { \"jmeno\": \"Novak Jan\",\n" +
"    \"plat\": 10000,\n" +
"    \"nar\": \"1990-03-13\"\n" +
"  },\n" +
"  {  " +
```

```
"     \"plat\": 2000\n" +
"     \"nar\": \"1980-09-03\"\n" +
"   }\n" +
"]";

el = KXmlUtils.parseXml(jdef).getDocumentElement();
xdef = KXmlUtils.nodeToString(XJUtil2.jDefToXDef(el), true);
xml = XJUtil2.jsonToXmlWithItems(json);

xp = compile(xdef);
assertEq(xml, parse(xp, "", xml, reporter));
assertNoErrors(reporter, xml);
}
```

Listing 5.12: Testování v Test000.java

# Závěr

Cílem této práce bylo navržení a implementace zpracování formátu JSON ve frameworku X-defice, což se podařilo. Aby byl tento cíl splněn, musela jsem se seznámit s dosavadní verzí frameworku X-definic, se specifikací JSON Schema a s obousměrnou transformací mezi formáty JSON a XML.

Při transformaci mezi formáty JSON a XML jsem postupovala dle doporučení W3C, které je v této práci popsáno. Dále jsem při návrhu o rozšíření frameworku X-definice o možnost zpracování formátu JSON měla brát ohledy na stávající návrh specifikace JSON Schema. S vedoucím této diplomové práce jsme se rozhodli vytvořit nový formát - J-definice, který se nám jeví přehlednější než JSON Schema. O tom, jak taková J-definice vypadá, se můžeme dočíst v této diplomové práci. Zpracování vstupního formátu JSON a J-definic je implementováno tak, že se vstupní JSON převede na formát XML. A J-definice se převede na X-definice, která umí zpracovávat formát XML.

Výsledná praktická část není dokonalá. Je zde několik vylepšení, která se nabízejí. J-definice není plně integrována do aparátu X-definic, tudíž se převody na formáty XML a X-definice musejí vytvářet před voláním metody parse. Dále můžeme diskutovat o zjednodušení struktury převedeného formátu JSON do formátu XML. Framework z X-definic vytváří X-komponenty, které jsou vygenerované Java třídy a strukturou odpovídají určitému modelu X-definic. Každému elementu X-definic odpovídá jedna X-komponenta [19]. Při stávající struktuře převedeného XML dokumentu jsou X-komponenty nepřehledné. Možné řešení u triviálních elementů (js:null, js:number, js:boolean, js:string) je změnit pojmenování elementu na jméno, které obsahuje atribut js:key. Dalším možným vylepšením může být rozšíření kontroly výskytu o nullable v X-definicích a v J-definicích, které by nám s kombinací optional sdělilo, že při absenci prvku ve vstupním formátu JSON/XML se vypíše tento prvek, který bude mít hodnotu nullable.

Vidíme, že tato výsledná praktická část se v budoucnosti může rozšířit o výše zmíněná rozšíření.

# Literatura


[1] Syntea software group a.s.: *Technologie - X-definice [online]*. [cit. 2018-11-23]. Dostupné z: https://www.syntea.cz/technologie/#1446030654562-9481112b-0f5f

[2] Syntea: *Technologie - X-definice [online]*. [cit. 2018-11-23]. Dostupné z: https://www.syntea.cz/technologie/#1446030654562-9481112b-0f5f

[3] X-definice 3.2. - Language description [online]. [cit. 2019-4-15]. Dostupné z: http://xdef.syntea.cz/tutorial/en/userdoc/xdef-32.pdf

[4] X-definice 3.2. - Java programming guide [online]. [cit. 2019-4-17]. Dostupné z: http://xdef.syntea.cz/tutorial/en/userdoc/xdef-32_Programming.pdf

[5] Droettboom, M.: What is a schema? [online]. 2019, [cit. 2019-4-9]. Dostupné z: https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html

[6] Schema, J.: Specification [online]. 2019, [cit. 2019-4-12]. Dostupné z: http://json-schema.org/specification.html

[7] Schema, J.: A Media Type for Describing JSON Documents [online]. 2019, [cit. 2019-4-12]. Dostupné z: http://json-schema.org/latest/json-schema-core.html

[8] Schema, J.: A Vocabulary for Structural Validation of JSON [online]. 2019, [cit. 2019-4-12]. Dostupné z: http://json-schema.org/latest/json-schema-validation.html

[9] Schema, J.: A Vocabulary for Hypermedia Annotation of JSON [online]. 2019, [cit. 2019-4-12]. Dostupné z: http://json-schema.org/latest/json-schema-hypermedia.html

[10] Schema, J.: Relative JSON Pointers [online]. 2019, [cit. 2019-4-12]. Dostupné z: http://json-schema.org/latest/relative-json-pointer.html

[11] Schools, W.: JSON - Introduction [online]. 1999-2019, [cit. 2019-3-13]. Dostupné z: https://www.w3schools.com/js/js_json_intro.asp

[12] Json.org: Introducing JSON [online]. 2017, [cit. 2019-4-15]. Dostupné z: https://www.json.org/

[13] Schools, W.: JSON - Syntax [online]. 1999-2019, [cit. 2019-3-13]. Dostupné z: https://www.w3schools.com/js/js_json_syntax.asp

[14] Schools, W.: XML Syntax Rules [online]. 1999-2019, [cit. 2019-3-13]. Dostupné z: https://www.w3schools.com/xml/xml_syntax.asp

[15] W3: Extensible Markup Language (XML) 1.0 (Fifth Edition) [online]. 2017, [cit. 2019-4-15]. Dostupné z: https://www.w3.org/TR/xml/

[16] W3: Processing JSON Data [online]. 2017, [cit. 2019-3-15]. Dostupné z: https://www.w3.org/TR/xslt-30/#json

[17] Oracle: Class TreeMap<K,V> [online]. 1993 - 2018, [cit. 2019-4-6]. Dostupné z: https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

[18] Oracle: Class LinkedHashMap<K,V> [online]. 1993 - 2018, [cit. 2019-4-6]. Dostupné z: https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html

[19] X-definice 3.2. - X-components [online]. [cit. 2019-5-3]. Dostupné z: http://xdef.syntea.cz/tutorial/en/userdoc/xdef-32_X-component.pdf

Příloha **A**

# Seznam použitých zkratek

**XML** Extensible Markup Language

**JSON** JavaScript Object Notation

**YAML** YAML Ain't Markup Language

**W3C** World Wide Web Consortium

Příloha **B**

# Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `examples` ..... adresář se spustitelnými příklady
- `jar` adresář s výsledným JAR souborem potřebným k práci s X-definicemi
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `document` ..... dokumentace
  - `userdoc` ..... uživatelská dokumentace k X-definicím
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `zadani.pdf` ..... zadání práce ve formátu PDF