TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Online aplikace pro transformace a validace datových struktur

Diplomová práce

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

Autor práce: **Bc. Michal Nekvasil**
Vedoucí práce: Ing. Mojmír Volf

Liberec 2017

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

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Michal Nekvasil**
Osobní číslo:	**M15000181**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Informační technologie**
Název tématu:	**Online aplikace pro transformace a validace datových struktur**
Zadávající katedra:	**Ústav nových technologií a aplikované informatiky**

Zásady pro vypracování:

Na základě neúspěšné obhajoby diplomové práce a ústního doporučení státní zkušební komise dne 1.2.2017 je vystaveno k dopracování, resp. přepracování toto zadání, které je se stejným názvem a následujícími body k vypracování, jako neobhájené zadání z 20.10.2015.

1. Detailně se seznamte s dostupnými standardy jazyků pro popis dat, programovými prostředky pro validaci a transformaci jazyků pro popis dat a možnostmi jejich implementace do online prostředí.
2. Navrhněte serverovou aplikaci s modulární architekturou validačních a transformačních nástrojů i formátů.
3. Implementujte zvolené řešení na vybrané serverové platformě s důrazem na minimální technická omezení z hlediska objemu dat, dobu jejich zpracování a softwarových nároků na klientská zařízení.
4. Implementujte správu uživatelů s možností pokročilejších funkcí validace a transformace periodicky opakované úlohy, dávkové úlohy.
5. Zdokumentujte řešení a připravte na budoucí rozšíření z hlediska podporovaných formátů a využívaných nástrojů.

Rozsah grafických prací: **dle potřeby**

Rozsah pracovní zprávy: **cca 60 stran**

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

Seznam odborné literatury:

[1] HAROLD, Elliotte Rusty a W MEANS. XML in a nutshell. 3rd ed. Sebastopol, CA: O'Reilly, c2004, xix, 689 p. In a nutshell (O'Reilly & Associates). ISBN 0596007647.
[2] DOUG TIDWELL. XSLT. 2nd ed. Farnham: O'Reilly, 2007. ISBN 9780596527211.
[3] VAN DER VLIST, Eric. RELAX NG. 1st ed. Sebastopol, CA: O'Reilly, c2004, xviii, 486 p. ISBN 0596004214.
[4] WALMSLEY, Priscilla. Definitive XML Schema. 2nd ed. Upper Saddle River, N.J.: Prentice Hall, c2013, xxxviii, 727 p. Charles F. Goldfarb definitive XML series. ISBN 9780132886727.

Vedoucí diplomové práce: **Ing. Mojmír Volf**
Ústav nových technologií a aplikované informatiky

Datum zadání diplomové práce: **1. února 2017**

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

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

L.S.

Technická univerzita v Liberci - 14 -

Ing. Josef Novák, Ph.D.
vedoucí ústavu

V Liberci dne 1. února 2017

Prohlášení

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

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

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

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

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

Datum: 15. 5. 2017

Podpis:

Poděkování

Chtěl bych poděkovat všem, kteří mě během tvorby této práce podporovali. Jedná se zejména o maminku a tatínka, kteří i přes ne úplně snadnou cestu k finálnímu výsledku, nepřestali věřit a podporovat mě v tvorbě této práce. Dále mým přátelům, kteří mě motivovali a podporovali ve chvílích, kdy jsem si nebyl jistý, zda vše zvládnu v čas a v pořádku. Poděkování patří také v neposlední řadě vedoucímu mé práce, panu Ing. Mojmíru Volfovi, který poskytoval cenné rady a vedl mne ke zdárnému výsledku.

Abstrakt

Tato práce se zabývá problematikou jazyků pro popis dat a následnou prací s nimi. Práce se konkrétně zabývá rozšiřitelným značkovacím jazykem XML. Tento jazyk umožňuje pomocí značek popis datových struktur a jejich zpracování. Dále se práce zabývá validací datových struktur a jejich transformacemi, převážně v jazyce XML. V rámci textu je popsán a vysvětlen princip značkovacích jazyků a jejich použití k popisování datových struktur, dostupné standardy pro operace s těmito strukturami a je navržena a implementována modulární online aplikace, která tyto operace umožňuje provádět a v reálném čase vracet výsledek uživateli. Implementace dále umožňuje pokročilou zprávu úloh. Uživateli je umožněno po přihlášení prohlížet výsledky minulých úloh, stáhnout si ho v textovém formátu na lokální uložiště, případně plánovat úlohy. Plánovat úlohu je možné jak v konkrétní čas, tak také periodicky.

Samotná implementace je poté provedena na serveru s operačním systémem Ubuntu a s využitím programovacího jazyku JavaScript. Tento jazyk je použit jak na klientské části, tak na části serverové. Jedná se zároveň o demonstraci využití jednoho jazyku pro obě části aplikace a tím usnadnění vývoje a snížení nároků na programátora. Architektura aplikace je navržena jako modulární. Modulární architektura je navržena ve smyslu snadného přidání podporovaných formátů, výstupů a možností plánování úloh. Text obsahuje popis a postup přidání rozšiřujících částí, což je realizováno formou dopsání částí kódu do aplikace. Není realizováno uživatelské rozhraní k přidávání modulů, což nebylo náplní práce.

K realizaci serverové části aplikace je využit framework Node.js. Tento framework umožňuje využít technologii JavaScript na straně serveru a je už ze svého principu postaven na modulární architektuře. K realizaci serveru je použita mimo jiné knihovna Express. Na straně klienta se jedná o stránky napsané v jazyce HTML s využitím JavaScriptu a frameworku Bootstrap. Celá aplikace byla vyhotovena a je k dispozici online.

Klíčová slova:

Jazyky pro popis dat, validace datových struktur, transformace datových struktur, Javascript, plánování úloh, webová aplikace

Abstract

This work deals with the issue of languages for data description and subsequent work with them. The work is mainly concerned about the extensible markup language XML. This language allows using tags describing data structures and it's processing. This work deals with the validation of the data structures and their transformations. Within the text are described and explained principles of markup languages and their uses to describe data structures and available standards for operations with these structures. And how the aplplication is designed and implemented as modular online application that allows to perform these operations in real time and return the result to the user. The implementation also provides advanced tasks. The user is allowed, after logging in, to view the results of previous tasks, download it in text format on disk and possibly schedule tasks. Users can schedule tasks to a particular time and also periodically.

The implementation itself is then performed on a server running on Ubuntu operation system and using the JavaScript programming language. This language is used on both the client part and on the server part. It is also a demonstration of the use of one language for both parts of the application and thereby facilitate the development and reduce the demands on the programmer. Application architecture is designed as modular. The modular architecture is designed to easily add other supported formats, output options and scheduling options. The text contains a description of how to add an extension to the application, which takes the form of writing sections of code into the application. User interface is not inplemented for adding modules, which was not the task of this work.

The server side implementation uses Node.js framework. The framework allows you to use JavaScript technology on the server and is already on its principles built on a modular architecture. Implementation uses, inter alia, library Express. The client side is written in HTML and is using JavaScript and Bootstrap framework. The application has been prepared and is available online.

Keywords:

languages for data description, validation of data structures, transformations of data structures, javascript, task cheduling, web applications

Obsah

Poděkování ... 4
Abstrakt ... 5
Abstract ... 6
Obsah ... 7
Seznam ukázek ... 8
Seznam obrázků ... 9
Úvod ... 10
1 Jazyky pro popis dat ... 11
1.1 Značkovací jazyky ... 12
1.2 Úvod do jazyka XML ... 13
1.3 Princip fungování XML ... 16
1.4 Vývoj jazyka XML ... 18
1.5 Pravidla jazyka XML ... 19
1.6 Struktura XML dokumentu ... 20
1.7 Validace dokumentu ... 22
1.7.1 Validace pomocí DTD ... 22
1.7.2 Validace pomocí XML Schema ... 24
1.8 Transformace XML ... 29
1.8.1 Transformace pomocí jazyka XSLT ... 30
2 Vlastní aplikace pro validaci a transformaci ... 33
2.1 Návrh aplikace ... 34
2.2 Zvolené technologie ... 35
2.2.1 Klientská část ... 36
2.2.2 Serverová část ... 36
2.3 Struktura aplikace ... 39
2.4 Zdrojový kód aplikace ... 42
2.4.1 Inicializační část programu ... 42
2.4.2 Zpracování dotazů uživatele ... 44
2.4.3 Konfigurace, zpracování dokumentů a balíčků ... 47
2.4.4 Uživatelské rozhraní ... 48
2.5 Pokročilé možnosti spouštění úloh ... 51
2.6 Modulární struktura aplikace ... 53
2.6.1 Rozšíření podporovaných formátů ... 54

2.6.2 Rozšíření možností spouštění naplánovaných úloh 60
Závěr 62
Seznam použité literatury 64

Seznam ukázek

Příklad 1 - XML dokument 16
Příklad 2 - jednoduchý XML dokument 20
Příklad 3 - složitější XML dokument 20
Příklad 4 - mixed content 21
Příklad 5 - zápis DTD validace 22
Příklad 6 - vložení DTD 23
Příklad 7 - validní XML dokument s DTD 23
Příklad 8 - nevalidní XML dokument 24
Příklad 9 - ukázkový XML dokument pro validaci XML Schema 25
Příklad 10 - zápis elementu v XML Schema 25
Příklad 11 - zápis atributu v XML Schema 26
Příklad 12 - přiřazení atributu k elementu 26
Příklad 13 - ukázka zápisu XML Schema 27
Příklad 14 - příklad XML pro transformaci 31
Příklad 15 - zápis transformace XSLT 31
Příklad 16 - výsledek transformace Příkladu 18 32
Příklad 17 - Node.js - asynchronní zpracování 37
Příklad 18 - ukázka struktury programu 40
Příklad 19 - ukázka obsahu souboru Server.js 42
Příklad 20 - kód spouštějící webový server 43
Příklad 21 - schéma databáze 44
Příklad 22 - zápis zpracování akce 44
Příklad 23 - funkce isLoggedIn 44
Příklad 24 – zpracování souboru 45
Příklad 25 - validace XML dokumentu 46
Příklad 26 - soubor database.js 47
Příklad 27 - značka pro EJS 48
Příklad 28 - volání funkce příkazové řádky z Node.js 55
Příklad 29 - přidání možnosti validace pomocí Relax NG 55
Příklad 30 - funkce validující pomocí Relax NG 56
Příklad 31 - přidání reference do route.js 57
Příklad 32 - přidání zpracování požadavku na validaci Relax NG 57
Příklad 33 - přidání zpracování Relax NG do pokročilých úloh 59
Příklad 34 - přidání nové možnosti periodického zpracování 61
Příklad 35 - přidání zpracování periodické úlohy 61

Seznam obrázků

Obrázek 1 - Hlavní stránka 49
Obrázek 2 - Ukázka stránky Nahrávání souborů 50
Obrázek 3 - Náhled stránky Práce se soubory 51
Obrázek 4 - Snímek stránky Plánování úloh 52

Úvod

Cílem této diplomové práce je popsat problematiku popisu datových struktur a dostupných jazyků a formátů určených k popisu těchto dat. Získané znalosti jsou poté využity k vytvoření online webové aplikace, která bude umožňovat tyto formáty validovat a transformovat. Aplikace pro transformace a validace datových struktur již existují, přínosem této aplikace by měla být hlavně možnost pokročilého spravování uživatelských účtů ve smyslu ukládání použitých a výsledných dat na vzdáleném uložišti, jejich správa a v neposlední řadě možnost spouštění naplánovaných úloh v daný čas, případně periodicky či dávkově. Aplikace vytvořená v rámci této práce by také měla být navržena jako modulární, tedy umožňovat rozšíření funkcionality. A to jak z hlediska podporovaných formátů a jazyků pro popis dat, tak také z hlediska možností pokročilého spouštění úloh. V této práci je nejdříve popsána problematika jazyků pro popis dat. Je popsán jejich účel a na vybraném jazyku ukázáno, jak vypadá struktura takového jazyka a jak se vytváří dokumenty, obsahující data popsaná tímto jazykem. Dále se v práci popisuje návrh aplikace, pro práci s těmito datovými strukturami. Nakonec je popsána realizace výsledné aplikace, včetně popisu pokročilých funkcí, jako správa proběhlých a naplánovaných úloh. Výsledkem této práce je poté vlastní aplikace umístěná na webovém serveru, která umožňuje provádět popsané funkce.

Přenos dat mezi systémy je v dnešní době důležitou součástí vývoje aplikací. Vnitřně každá aplikace zpracovává data zpravidla svým specifickým způsobem. Pro přenos dat mezi různými systémy je tedy vhodné využít nějaký obecný nástroj nebo formát, který umožňuje popis struktury a významu těchto dat. K těmto účelům jsou vhodné právě jazyky pro popis dat, které toto umožňují a navíc přidávají i další nástroje pro práci s těmito dokumenty.

První část pojednává o jazycích pro popis dat. Je zde popsáno, co jsou jazyky pro popis dat a k čemu slouží. Existuje množství formátů pro popis a uchování dat, základní rozdělení je na formáty binární a textové. Binární formáty jsou vhodné pro strojové zpracování, jejich forma je bližší strojům. Textové formáty jsou naopak zřetelnější z lidského pohledu, umožňují přímé čtení obsahu, protože se jedná o data složená výhradně z tisknutelných znaků. Velikost uložení každého znaku se může lišit dle použitého kódování. Jazyky pro popis dat jsou určeny k popisu struktury či významu těchto dat. Protože se jedná z pravidla o data textová, bude o nich dále v textu mluveno jako o dokumentech. Existuje více formátů těchto jazyků. V rámci této práce se bude popisován zejména formát *XML*, tedy *Extensible Markup Language*. Byl vybrán jako ukázkový zejména díky svému obecnému rozšíření a dostupnosti nástrojů a jazyků pro práci s ním. Jedná se o obecný značkovací jazyk, který umožňuje snadný popis dat a vytváření konkrétních značkovacích jazyků a jejich popisu opět pomocí jazyka *XML*. Poté je možno dále specifikovat strukturu i obsah dokumentu pomocí předpisů a tím umožnit validaci, tedy kontrolu, zda konkrétní dokument a všechny jeho části odpovídají pravidlům daného jazyka. A také provádět transformace dokumentu, tedy upravovat dokumenty, případně vybírat pouze určitá data z dokumentu a tak dále. Tyto operace jsou zde popsané, stejně jako jsou ukázány příklady dostupných prostředků k jejich provádění.

V druhé části jsou nejdříve popsány požadavky na klientskou a serverovou část aplikace a vysvětleno, co by mělo být jejím přínosem. Poté je popsáno, jaké technologie byly zvoleny pro vývoj aplikace. Výběr technologií vychází z vyjmenovaných požadavků. Aplikace je navržena jako serverová, tedy všechny operace s datovými strukturami probíhají na serveru. Zde jsou také vstupy a výstupy uloženy pro pozdější opětovné zpracování. Důležitým prvkem, při výběru technologie je požadavek, aby aplikace byla navržena jako modulární. Z toho plyne i výběr jazyka *JavaScript* jako programovacího

jazyka, ve kterém bude napsaná výkonná část kódu aplikace. A také výběr technologie *Node.js*[4] na serverové části.

Následující část se zabývá samotnou realizací aplikace. Na začátku je popsána struktura aplikace. Jak z hlediska principu fungování, tak i souborová struktura aplikace, tedy kde se nachází jaká část kódu. Je zde popsáno, jaké programátorské prostředky byly použity, jak byla aplikace naprogramována a popsáno řešení na zvolené serverové platformě. Při vývoji byly použity knihovny třetích stran, které jsou využívány pro konkrétní úkony. Tyto knihovny jsou zde vyjmenovány a je popsán jejich účel. V kapitole *Zdrojový kód aplikace* jsou poté podrobně popsány jednotlivé části programu. Jsou zde uvedeny ukázky zdrojového kódu, na kterých je popsáno fungování aplikace. Také se zde nachází popis konfigurace aplikace, tedy jak aplikaci zprovoznit. Důležitou částí je i popis uživatelského rozhraní a popis implementace správy uživatelských účtů.

Poslední část práce se zabývá implementací pokročilých možností spouštění úloh a modulární architekturou aplikace. Je zde popsána implementace uživatelského rozhraní určeného k pokročilému zpracování úloh uživatele. Na snímku obrazovky aplikace je ukázáno, jaké má uživatel možnosti pokročilejšího spouštění úloh, konkrétněji možnosti naplánování úloh na daný čas a opakování v určitém intervalu, či spouštění dávkových úloh. Aplikace mimo jiné umožňuje validaci a transformaci dokumentů umístěných na vzdáleném uložišti zadáním jejich *URL* adresy.

Již bylo zmíněno, že aplikace byla navržena jako modulární, je tedy možné rozšířit její funkcionalitu. O rozšiřitelnosti aplikace pojednává kapitula *Modulární struktura aplikace*. Rozšiřování funkčnosti je navrženo tak, že probíhá připsáním krátkých částí kódu do aplikace. Text popisuje dvě možná rozšíření. Rozšíření z hlediska podporovaných formátů a rozšíření z hlediska možností pokročilého spouštění. Je názorně předvedeno přidání nového formátu validování, konkrétně *Relax NG*, do aplikace. A také rozšíření možností spouštění, zde se jedná o ukázku, jak přidat vykonávání úlohy v určitém vybraném intervalu. Oba případy jsou popsány a realizace ukázána na částech kódu. Popis řešení provádí čtenáře krok po kroku přidáním nové funkcionality a poskytuje tak návod k budoucímu rozšiřování aplikace.

1 Jazyky pro popis dat

Tato práce se zabývá problematikou popisu a uchování dat. Potřeba uchovávat a přenášet data je logická. Ať už se v začátcích jednalo o přenos dat z terminálu na pracovní stanici, až po dnešní celosvětovou internetovou síť, která je celá o sdílení a přenosu dat. Data je možné uchovávat a přenášet v různých formách. Jak již bylo řečeno v úvodu, jedno z rozdělení je na formu binární a textovou. Samozřejmě v principu jsou všechny data uložena ve formě jedniček a nul, tedy v binárním kódu. Rozdělení je tedy z tohoto pohledu věcí interpretace dat.

Binární soubor je takový soubor, který obsahuje data, které je nutné po přečtení nějak interpretovat, je tedy nutné znát strukturu dat. Binární soubor může obsahovat text, zvuk, video a další data. V případě, že chceme data předat dál, je nutné, aby druhá strana měla struktury těchto dat. Obecně jsou srozumitelné pouze pro zasvěcené programy. Někdy proto bývá binární soubor doplněn hlavičkou, která obsahuje metadata o daném souboru. Metadata jsou data, která poskytují informace o jiných datech, například právě o struktuře. Zpracování takového souboru je poté věcí samotného programu, většinou se soubor čte jako proud bytů, tedy po osmi bitech. Může se však v obecnosti jednat o libovolná data a je pouze na tvůrci, jak bude s těmito daty pracovat. Tato data mohou být

v principu také v čitelné formě, tedy mohou být přečteny v textovém editoru, ale obecně mohou obsahovat i netisknutelné znaky. Existuje mnoho obecně známých binárních standardů. Uveďme aspoň nějaké příklady, jedná se například o multimediální formáty *GIF* a *JPEG*, které slouží pro uchování grafické informace. Dále například formát *EXE*, jedná se o spustitelný program, tedy obsahuje posloupnost instrukcí a operandů, která jsou vykonávány. Práce s binárním souborem je z hlediska strojového času rychlejší, zejména díky přímému přenosu z a do paměti bez nutnosti konverze. Zápis je obvykle z hlediska velikosti na disku kratší a pro stroj čitelnější, protože pracuje často s optimalizovanějšími datovými typy než textový soubor.

Druhým typem je tedy textový soubor. Textový soubor je takový soubor, který obsahuje pouze textová data. Takováto data jsou složena výhradně z tisknutelných znaků, s výjimkou speciálních formátovacích znaků, jedná se běžně o mezeru, tabulátor a odřádkování. V závislosti na zvoleném kódování se může lišit datová velikost jednotlivých znaků, obvyklá velikost je jeden, dva či čtyři bajty. Kódování je vlastně přiřazení hodnoty zapsané bity konkrétnímu znaku, určuje jaký znak je jak uložen. Jedná se například o historicky používanou *ASCII* znakovou sadu nebo dnes moderní *Unicode*. Znaková sada *ASCII* obsahuje definici 256 znaků, což je málo i pro evropské národní jazyky. Proto byla vytvořena znaková sada *Unicode*, která již umožňuje pojmout prakticky všechny možné používané světové znaky. Tato znaková sada je rozšířením znakové sady *ASCII* a prvních 256 znaků je shodných. V současné době obsahuje kolem 120 tisíc znaků. Výsledný soubor je čitelný po otevření v textovém editoru. Z lidského hlediska se jedná o výhodu, protože je snadnější data například zkontrolovat, případně editovat, pouhým otevřením v textovém prohlížeči, který je již standardní součástí všech operačních systémů. Počítačové zpracování je obecně pomalejší. S daty se pracuje v textové formě, což je pomalejší než u binárních souborů, hlavně z důvodů nutnosti konverze při zápisu i čtení, která může být i poměrně složitá. Textové soubory mohou kromě prostého textu obsahovat také další informace. Nejčastěji se jedná o informace o struktuře, významu nebo způsobu zobrazování. K tomuto účelu byly vytvořeny značkovací jazyky. Značkovací jazyky vkládají do textu dodatečné informace formou značek (tagů), příkazů nebo direktiv. Výsledný zdrojový text je stále pouze textový soubor, ale obsahuje navíc informace, které může program, zpracovávající daný soubor, využít a text například formátovat a zobrazit uživateli v upravené formě. Nejznámějšími značkovacími jazyky jsou *HTML* a *XML*, ze značkovacích jazyků určených primárně pro typografickou úpravu uveďme jazyk pro počítačovou sazbu textu *TeX*.

1.1 Značkovací jazyky

Jak již bylo řečeno, značkovací jazyk je v informatice prostředek pro obohacení textu o další informace. Tyto informace mohou být různého typu. Historicky se jednalo hlavně o typografické jazyky. Tyto jazyky byly určeny k formátování textu, například pro výstup na terminály či tiskárny. A také pro programy pro počítačovou sazbu, určené k tvorbě tištěného dokumentu za pomoci počítače. V současné době se značkovací jazyky používají velice často jako jazyky popisující strukturu dat. Cílem je mít určitý jednotný mechanismus, jak data popisovat. Z toho poté plyne možnost vytvořit obecné nástroje pro práci s těmito daty, jako například možnost kontroly a konverze těchto dat, což je jedna z hlavních náplní této práce a bude to dále popsáno v textu.

Existuje více rozdělení značkovacích jazyků. Jedním z dělení je na jazyky popisné, výkonné a prezentační. V této práci je věnována pozornost zejména popisným jazykům. Konstrukce těchto

jazyků slouží k popisu, co za informace je v souboru, respektive dokumentu, obsaženo. Umožňují, zjednodušeně řečeno, stanovit jaká část dokumentu má jaký význam, zda se například jedná o nadpis, kde končí odstavec a podobně. Obecněji mohou obsahovat prakticky jakoukoliv informaci o určité části dokumentu a je ponecháno na programu, který bude tento dokument zpracovávat, jak danou informaci použije. Moderní a doporučený přístup je oddělit obsahovou část od vzhledové. Nejznámějšími zástupci jsou formát *XML* a *HTML*, které jsou určeny k popisu obsahu. K popisu vzhledu dokumentu existuje další formát a to *CSS*, kaskádové styly. O *XML* dokumentech bude řeč podrobněji později. Jazyky výkonné, jinak také procedurální, obsahují i instrukce na úrovni programovacího jazyka. Obvykle se jedná například o možnost ukládat nebo načítat proměnné z paměti, přiřazovat jim hodnoty dle podmínek a tak dále. Tyto jazyky většinou umožňují velmi detailně popsat, jak bude výsledný dokument vizuálně vypadat. Příkladem procedurálního jazyku je *TeX* nebo *PostScript*. *PostScript* je programovací prostředek pro popis tisknutelných informací. Je nezávislý na zařízení, na kterém se má dokument tisknout. Je hojně využíván u tiskáren. *TeX* je formát, respektive program, který umožňuje počítačovou sazbu textu. Vznikl původně na tisk skript. A konečně jazyky prezenční, které se soustřeďují hlavně na to, jak bude výsledný označkovaný text zobrazen nebo vytištěn. Jedná se o přístup typický pro textové procesory. Ovšem na rozdíl od popisných jazyků, jako *XML*, používají binární značkování nebo formátovací příkazy ukládají odděleně od textu. Příkladem tohoto typu je například formát *RTF*, *Rich Text Format*.

V rámci této práce se bude pracovat hlavně s jazyky pro popis dat, které nejpřesněji spadají do kategorie jazyků popisných. V práci se v praktické části využívá formát *XML*, který je silně rozšířen a má společné rysy i s dalším, ne-li více rozšířeným formátem *HTML*. Tato práce se dále zabývá validací a transformací těchto struktur. I to je důvod, proč byl jako referenční formát vybrán právě *XML*. Validovat se dají i jiné formáty, jako například *JSON*, již zmiňované *CSS* nebo *YAML*. *JavaScript Object Notation*, zkráceně *JSON*, je způsob zápisu dat v textové formě. *JSON* umožňuje jako vstup libovolně složitou strukturu a výstupem je vždy textový řetězec. Za velkým rozšířením tohoto formátu stojí mimo jiné i to, že rozšířený formát *HTTP*, využívaný v celosvětové internetové síti, je z návrhu textový, tudíž je potřeba data přenášet v textové podobě a to právě *JSON* umí. V kombinaci s rozšířeným používáním technologie *JavaScript* na klientské straně, se *JSON* stal jedním z hlavních formátů pro výměnu dat a tvoří jistý protiklad *XML* v této kategorii.

1.2 Úvod do jazyka XML

Jazyk *XML*, anglicky *Extensible Markup Language*, je obecný značkovací jazyk. Jedná se o jazyk, který byl vyvinut a standardizován konsorciem *W3C*, což je zkráceně mezinárodní konsorcium, které ve spolupráci s veřejností vyvíjí webové standardy. Obecný znamená, že obsahuje definici formy, jak má být validní *XML* dokument zapsán. Uživatel si poté sám definuje konkrétní podobu jazyka, jeho značky (dnes je již rozšířené počeštěný název z angličtiny tagy) a strukturu. Je dostatečně flexibilní, aby bylo možné ho použít dle konkrétních potřeb. Lze použít na webové stránky, výměnu elektronických dat, vektorovou grafiku, seznamy staveb, serializaci objektů, vzdálené volání procedur a tak dále. Je možno napsat vlastní program, který bude *XML* zpracovávat a manipulovat s daty v dokumentech. Navíc existuje řada volně dostupných knihoven, takže vývojář se může soustředit na logiku své aplikace a práci s *XML* zanechat těmto knihovnám. Ve zbytku kapitoly je čerpáno především z publikace *XML in Nutshell*[1].

Tento značkovací jazyk je široce používaný. Existuje řada nástrojů a standardů, pro práci s těmito dokumenty. Což jsou důvody, proč byl zvolen jako referenční značkovací jazyk této práce. Lze na něm dobře demonstrovat možnosti jazyků pro popis dat. Obecně se jazyk *XML* používá zejména pro uchování a přenos dat mezi aplikacemi. Z praktických důvodů vznikla již zmíněná řada nástrojů a standardů, které umožňují s dokumenty *XML* dělat pokročilé operace. Samozřejmostí je validace dokumentu. Ať již obecná validace správnosti zápisu *XML*, což znamená, že dokument musí mít minimálně následující vlastnosti. Musí mít právě jeden kořenový element. *XML* dokument je ve své podstatě hierarchický strom. Dále neprázdné elementy musí být ohraničeny startovací a ukončovací značkou, prázdné elementy mohou být označeny tagem „*prázdný element*“. Všechny hodnoty atributů musí být uzavřeny v jednoduchých nebo dvojitých uvozovkách, pár si samozřejmě musí odpovídat a opačný pár uvozovek může být použit uvnitř hodnot. A nakonec pravidlo, že elementy mohou být vnořeny, ale nemohou se překrývat. Jinak řečeno, každý nekořenový element musí být celý obsažený v jiném elementu. To opět vychází ze stromové povahy *XML* dokumentu. Toto jsou pouze základní pravidla zápisu *XML* dokumentu, neřeší validaci konkrétně navrženého jazyka. Validace konkrétního jazyku vyžaduje znát jeho strukturu. Je tedy potřeba určit nějaký předpis, dle kterého se validace provede. K tomuto lze v *XML* využít například jazyk *DTD* nebo *XML Schema*. Další užitečnou vlastností je schopnost dokumenty *XML* transformovat. Tedy z jednoho dokumentu vytvořit jiný. Využívaná je například transformace z *XML* do *HTML*. Opět se provádí například pomocí předpisu ve formátu *XSLT*. O těchto vlastnostech bude podrobněji psáno později. Jsou zde zmíněny, jako úvod do univerzálnosti jazyka a jako demonstrace, proč je jazyk *XML* dál dopodrobna rozebírán. Pro demonstraci je to ideální jazyk. Vzhledem k rozšířenosti obsahuje všechny důležité nástroje a vlastnosti, je dobře čitelný i lidského čtenáře a má velice blízko k formátu *HTML*, který je díky své jednoduchosti jedním z nejznámější i pro laickou veřejnost. Nyní již konkrétně o jazyce *XML*.

Jak již bylo řečeno, *XML* je obecný značkovací jazyk pro textové dokumenty. Data jsou v dokumentech ve formě textových řetězců. Tyto data jsou ohraničena textovými značkami, které tyto data popisují. Základní jednotka *XML*, složená ze značky a dat se nazývá element. Formát *XML* přesně definuje, jak má taková značka vypadat, jak jsou odděleny elementy, jaký tvar má mít značka, jaké názvy jsou akceptované, jaký je tvar atributů a tak dále. Jak již bylo řečeno, jazyk *XML* je velice podobný jazyku *HTML*, ale jsou zde určitě podstatné rozdíly.

Hlavním rozdílem je, že *XML* je obecný jazyk. To znamená, že nemá žádnou pevně definovanou sadu značek či elementů. Neexistují tedy žádné značky, které by fungovali obecně v každém dokumentu. Místo toho *XML* umožňuje vývojářům definovat si vlastní značky dle potřeby. Chemičtí pracovníci si mohou definovat elementy, které budou popisovat atomy, molekuly, vazby a tak dále. A naopak například prodejce nemovitostí si definuje elementy, které popíší daný byt, dům, lokaci, kde se nachází a podobně dle potřeby. Nápodobně muzikant může definovat elementy popisující noty, doby a texty písní. Jazyk má již v názvu *extensible*, tedy rozšiřitelný. Jeho hlavní vlastností je možnost rozšiřovat ho a přizpůsobit si ho mnoha různým využitím.

Ačkoliv je jazyk *XML* poměrně flexibilní v tom, jaké elementy povoluje definovat, v mnoha dalších vlastnostech je poměrně přísný. Obsahuje pravidla, jak mohou být značky umístěny, které názvy značek jsou přípustné, pravidla pro práci s atributy a tak dále. Tyto pravidla jsou dostatečně přesná, aby umožnila vytvoření obecných nástrojů pro procházení a čtení libovolných *XML* dokumentů. Dokument, který tyto pravidla splňuje, se označuje jako „*well-formed*“, v překladu správně vytvořený dokument. Pokud toto dokument nesplňuje, program ho obvykle odmítne zpracovat, tedy skončí

s chybou. Z důvodů přenositelnosti je možné vytvořit pravidla, jaké značky budou používány. Poté se již jedná o konkrétní aplikaci *XML* na daný problém.

Značky v *XML* dokumentu popisují jeho strukturu. Umožňují vidět, který element je asociován s jakým jiným elementem. Ve správně navrženém *XML* dokumentu značky také popisují význam jednotlivých bloků. Značka může například vyjadřovat, že daný element představuje příjmení osoby nebo barvu vozu. Naopak ve správně navrženém dokumentu značky nemají vliv na zobrazení. Tedy neříkají, zda je obsah elementu zobrazen například tučným písmem nebo jakou má mít barvu. *XML* je strukturální a sémantický značkovací jazyk, nikoliv prezentační, jak bylo popsáno v úvodu kapitoly o *XML*.

Značky použité v určitě aplikaci jazyka *XML* mohou být popsány ve schématu. A tyto dokumenty poté mohou být s tímto schématem porovnány. Dokumenty, které tomuto schématu odpovídají, jsou takzvaně validní, naopak dokumenty, které schéma nesplňují, jsou nevalidní. Zda je dokument validní, tedy závisí na schématu, proti kterému je dokument validován. Ovšem ne každý dokument musí být validní dle schématu, pro mnoho účelů je dostatečné, když je dokument pouze „*well-formed*“. Existuje mnoho jazyků pro definování schématu *XML* s různými možnostmi definování pravidel. Nejvíce podporovaný a jediný, který je definovaný standardem *XML* ve verzi 1.0 je jazyk *DTD*, v angličtině *Data Type Definition*. *DTD* obsahuje seznam všech povolených značek a specifikuje, kde a jak tyto značky mohou být použity v dokumentu. *DTD* je nepovinnou součástí *XML*. Na druhou stranu možnosti *DTD* nemusí vždy stačit. Syntaxe *DTD* je poměrně omezená a neumožňuje například definovat, jaký datový typ daný element obsahuje nebo jaký má mít rozsah. Pro takovéto potřeby existuje jazyk *XML Schema*. Je opět spravovaný konsorciem W3C a umožňuje definovat právě i typ a rozsah obsahu daného elementu. Kromě těchto dvou existují další jazyky, jako například *RELAX NG*, *Schematron*, *Hook* nebo *Examplotron*.

Všechny současné jazyky pro tvorbu schémat jsou deklarativní, což znamená, že říkají, co se má udělat, ale ne jak se to má udělat. Opakem je imperativní programování, kde jsou jednotlivé úkony popisovány algoritmem. Vždy zde budou výjimky, případy, které lze vyřešit jen programovacím jazykem, který splňuje kompletní nároky *Turingova stroje*. Například nelze pro každou položku vzít například její cenu a vynásobit kvantitou, vše sečíst a zkontrolovat, zda součet odpovídá jiné hodnotě. Samozřejmě programátor má možnost tyto podmínky vyřešit v kódu. Data jsou získávána pomocí programu na parsování *XML* dokumentu a programátor řeší další logiku.

Jak již z předchozího odstavce vyplývá, jazyk *XML* není všemocný, jsou oblasti, kde se použít nedá. *XML* je značkovací jazyk. To znamená, že zdaleka není řešením všech problémů. Zaprvé se nejedná o programovací jazyk. Neexistuje *XML* kompilátor, který by četl *XML* soubor a produkoval spustitelný kód. Lze samozřejmě definovat skriptovací jazyk, který by byl zapsaný v *XML* kódu a interpretovaný binárním programem. Lze tedy v *XML* zapsat například posloupnost příkazů, které potom nějaký interpret vykoná. V *XML* se například občas zapisují konfigurační soubory. Ovšem práci pořád vykonává samotný program a ne *XML* dokument. Stejně jako *XML* není transportní protokol. *XML* data neposílá, stejně jako třeba *HTML*. Ovšem je možné, a v praxi i používané, zapsat posílaná data v *XML* formátu a ty poté poslat běžným protokolem, jako například *HTTP*. *XML* také není databáze. *XML* sice v principu slouží k uchování dat a informací o nich, ale nenahrazuje databázové systémy. Samozřejmě je možné napsat databázový program, který bude pro uchování dat na disku využívat formát *XML*. Ovšem takový program by byl velice pomalý. V praxi databázové systému uchovávají data

ve složitých binárních strukturách a dále si uchovávají informace o vazbách a například o indexech, které pak umožňují nad těmito daty provádět rychle operace jako hledání a úpravu. *XML* v principu neslouží jako formát pro dlouhodobé uchování dat, neměl by tedy být takto používán. Jeho úloha je spíš jako univerzální nástroj pro přenos dat.

V oblasti přenosu dat nabízí *XML* velké možnosti. *XML* umožňuje přenášet data napříč platformami a s podporou i do budoucna. Historicky se data ukládala různými způsoby a přenášet je ze systému do systému byl kolikrát problém, protože každý si implementoval vlastní způsob. A v tom právě přichází síla *XML*. *XML* je velice jednoduchý, dobře zdokumentovaný formát. *XML* je možné přečíst běžným textovým prohlížečem, a to nejenom data, ale také značky. Takže i pro člověka je obsah velice srozumitelný a lze kontrolovat obsah i bez potřeby dokument zpracovávat a nějak interpretovat. Navíc značky jsou i ukončené, což přesně určuje, kde se nachází daná hodnota. Vše je srozumitelně, explicitně zobrazeno přímo v dokumentu. Není potřeba složitě zjišťovat, jak věc funguje, případně hledat v často nekompletní dokumentaci. Z dlouhodobého hlediska je lepší používat dobře zdokumentované, lehce pochopitelné textové formáty, jako právě *XML*. *XML* umožňuje snadné přenášení dat ze systému do systému. Navíc s možností kontrolovat data, pomocí nástrojů na validaci, stejně jako data transformovat.

1.3 Princip fungování XML

V této části práce se bude popisovat, jaké jsou principy fungování *XML*, jeho zápis a struktura. Uveďme si na začátek příklad, jak by mohl vypadat *XML* dokument. V následujícím případě se jedná o popis produktu, který je například někde uskladněn.

```
<?xml version="1.0"?>
<výrobek id="18">
     <výrobce>Firma s.r.o.</výrobce>
     <název>Stůl</název>
     <množství>3</množství>
     <velikost>1.6m</velikost>
     <barva>Bílá</barva>
     <popis>Jídelní stůl typ 4</popis>
</výrobek>
```

Příklad 1 - XML dokument

Příklad 1 - XML dokument ukazuje, jak vypadá standardní a validní *XML* dokument. Jak je zřejmé z ukázky, tento dokument obsahuje informace o výrobci, názvu, množství naskladněných kusů, velikosti, barvě a popisu určitého výrobku. Dokument je textový, lze jej uložit a editovat běžnými textovými editory. Není potřeba používat specializované nástroje pro práci s *XML*, i když takové nástroje samozřejmě existují. Použití těchto editorů může být výhodné, ovšem často jsou navrženy na úkor jednoduchosti. Programy, které řeší i strukturu a význam těchto dat, se nazývají *parsery*.

Parsování, formálnější označení je syntaktická analýza, je v informatice proces, při kterém se analyzuje posloupnost prvků, například řetězce znaků, s cílem určit strukturu dané soustavy, v našem případě *XML* dokumentu. Program, který tuto analýzu provádí, se nazývá *parser*, opět

formálněji řečeno syntaktický analyzátor. Tento program má za úkol daný dokument rozdělit na jednotlivé elementy, atributy a další části. A poté poskytuje určité přístupové rozhraní k obsahu dokumentu, což programátorovi usnadňuje práci. Umožňuje vývojářům přistupovat k dokumentu postupně, procházet jednotlivé elementy a vykonat s nimi potřebné akce. Zde se dostáváme opět k validnímu dokumentu. Za ideálních podmínek, by měl být dokument validní. Pokud *parser* během procházení narazí na chybu, obvyklou reakcí je ukončení procházení a vrácení chyby.

Existují dva přístupy, které se používají při *parsování XML* dokumentů. Prvním je objektový přístup, nazývá se *DOM*, *Document Object Model*. Pro daný dokument se v paměti vytvoří struktura, objektově orientovaná, a celý obsah dokumentu se do ní nahraje. Jak již bylo řečeno, *XML* dokument má formu stromu, proto i forma uložení do objektů do paměti má formu stromu. Výsledek po *parsování* je, že programátor má k dispozici celý dokument nahraný v paměti a může k němu přistupovat. Tento princip je optimální, pokud se k elementům přistupuje v náhodném pořadí nebo opakovaně. Historicky existovalo více přístupů, jak *DOM* vytvořit, zejména v oblasti internetových prohlížečů. Proto se již zmiňované konsorcium *W3C* rozhodlo přístup sjednotit a vytvořilo specifikaci *W3C Document Object Model*. Tato specifikace je platformě a jazykově nezávislá. A druhým přístupem je použití metody *SAX*, anglicky *Simple API for XML*. Tato metoda naopak umožňuje rychlé sekvenční procházení. Zpracování probíhá proudově. Dokument je rozdělen na jednotlivé části a celý průběh *parsování* je řízen voláním událostí. Pokud *parser* narazí na nějakou značku, případně jinou část dokumentu, tak vyvolá událost. Je pak již na programátorovi, jak danou událost zpracuje a jaké kroky dále provede. I když se na první pohled může zdát, že je tato metoda náročná, má své uplatnění ve složitějších způsobech zpracovávání dokumentů *XML*. V případech, kdy nepotřebujeme k obsahu přistupovat náhodně, ale například potřebujeme celý dokument sekvenčně jednou projít, je rychlost zpracování vyšší než u *DOM*, stejně jako paměťové nároky. U tohoto typu *parseru* nedochází k načtení celého obsahu do paměti, ale obsah se čte postupně.

Programy, využívající formátu *XML*, obvykle vyžadují přísnější podmínky pro strukturu využívaného *XML* dokumentu a nestačí jim pouze základní kontrola, zda je *XML* dokument „*well-formed*“. Existují jazyky, které umožňují definovat, kde a jaký element je možno použít. Jedná se například o již zmiňovaný *W3C XML Schema Language*, dále například *RELAX NG* a *DTD*. V *XML* dokumentu může, ale nemusí být uveden odkaz na takovýto soubor s validačními pravidly. *Parser*, pokud odkaz na schéma objeví, může provést před samotným čtením dat validaci dokumentu tímto schématem. A pokud najde při validace chybu, tak to oznámí aplikaci, ve které běží a umožní jí na to reagovat. Pokud se nejedná o fatální chybu nebo o základní chybu „*well-formed*“ dokumentu, tak je v určitých případech možné pokračovat ve zpracovávání a chybu ignorovat. Validace dokumentu je popsána v kapitole *Validace dokumentu*.

Stejně tak je možné provádět transformace těchto dokumentů, změnit strukturu *XML* nebo dokonce formát výsledku. Výsledkem transformace tedy nemusí být dokument ve formátu *XML*, ale libovolný textový soubor. Pro transformaci se nejběžněji využívá jazyk *XSLT*, *Extensible Stylesheet Language Transformation*. Transformaci se věnuje kapitola *Transformace XML*.

1.4 Vývoj jazyka XML

Jazyk *XML* je potomkem jazyka *SGML*, anglicky *Standard Generalized Markup Language*. Tento jazyk byl vymyšlen ve společnosti *IBM* v 70. letech minulého století. Na vývoji se celosvětově podílelo několik stovek lidí, než byl v roce 1986 standardizován. Jazyk *SGML* řešil podobné problémy jako jazyk *XML*. Jedná se také o sémantický a strukturální značkovací jazyk. Jazyk *SGML* je velice silný nástroj, dosáhl řady úspěšných využití, ve vesmírném programu k uchovávání technické dokumentace, například. Jednou z nejznámějších a nejúspěšnějších aplikací je jazyk *HTML*. Nicméně *HTML* je pouze aplikací jazyka *SGML*, nejedná se o obecný jazyk a ve své podstatě teda ani zdaleka nenabízí možnosti, jako jazyk *SGML*. Obsahuje definované značky a jejich počet je omezen. Tyto značky jsou určeny pouze k popisu webové stránky a to k popisu prezentačně orientovanému. Jedná se vlastně o značkovací jazyk, který byl přijat víceméně jednotně internetovými prohlížeči. Což jeho použití omezuje pouze na použití v návrhu webových stránek. *HTML* ve své podstatě neumožňuje výměnu dat například mezi různými databázemi. *HTML* je určeno k tvorbě webových stránek, což dělá velice dobře, ale tím jeho možnosti končí. Problémem jazyka *SGML* je jeho opravdu velká komplikovanost. Oficiální dokumentace k *SGML* má přes sto padesát stránek technického textu. Řeší se zde množství speciálních případů, i těch, které mohou nastat jen velice nepravděpodobně. Je to dokonce tak komplikované, že téměř žádný program tento standart zcela neimplementoval. Programy, které používaly jen jistou podmnožinu možností *SGML* byly mezi sebou velice často nekompatibilní.

V roce 1996 pánové *Jon Bosak*, *Tim Bray*, *C. M. Sperberg, James Clark* a další začali pracovat na odlehčené verzi *SGML*. Cílem bylo zachovat co nejvíce schopností *SGML* a zároveň jazyk očistit od věcí, které byly prokázané jako redundantní, příliš komplikované pro implementaci, matoucí pro konečné uživatele nebo jednoduše nepoužívané, jak bylo zjištěno za léta zkušeností se *SGML*. Výsledkem bylo v 1998 vydání jazyk *XML* ve verzi 1.0. Okamžitě po uvedení sklidilo *XML* velký úspěch. Velké množství vývojářů vědělo, že potřebuje strukturální značkovací jazyk, ale nebyli ochotni akceptovat složitost a komplexnost jazyka *SGML*. Toto byl jen začátek, další standardy následovaly. Dalším rozšířením byl standard *XML Namespaces*, neboli jmenných prostorů. Cílem byla snaha umožnit používání značek z různých *XML* aplikací být používán v jednom dokumentu bez toho, aby došlo ke konfliktu názvů. Například webová stránka může mít element *nadpis*, který bude znamenat nadpis stránky, a zároveň může obsahovat informace o knížce a ta má taky element *nadpis*.

Následoval standard *XSL, Extensible Stylesheet Language*. Aplikace *XML* sloužící k transformování *XML* dokumentů do formy zobrazitelné webovými prohlížeči. Tento standard se brzy rozdělil do *XSLT* (*XSL Transformation*) a *XSL-FO* (*XSL Formatting Objects*). *XSLT* se stalo obecným jazykem pro transformaci jednoho *XML* dokumentu do jiného dokumentu, ať už se jedná o transformaci k zobrazení dokumentu jako webovou stránku nebo k jinému účelu. *XSL-FO* je aplikace *XML* sloužící k popisu jak stránky k tisku, tak webových stránek, který je soupeřem formátu *PostScript*. Nicméně *XSL* není jediná možnost, jak stylovat *XML* dokumenty. *CSS*, *Cascading Style Sheets*, byl již používán v *HTML* a šel snadno použít i v *XML*. S příchodem *CSS Level 2*, se stalo stylování *XML* dokumentů hlavním účelem kaskádových stylů.

Dalším jazykem byl *XLink*, *Extensible Linking Language*. Cílem bylo umožnit lepší možnosti propojení dokumentů s většími možnostmi než *HTML* tag *<a>*. Také tento standard se rozdělil do dvou různých. Jedním je *XLink*, který slouží k popisu propojení dokumentů. A druhým je *XPointer*, který naopak slouží k adresování částí *XML* dokumentu. V této fázi vyplynulo, že *XSLT* a *XPointer* ve své podstatě dělá stejnou věc, identifikuje určitou část dokumentu. Přesto byly tyto technologie

nekompatibilní. Kvůli tomu byla část obou standardů identifikující části dokumentu vzata z obou standardů a zkombinována do nového, *XPath*.

Další potřebnou částí, pro práci s *XML* dokumenty bylo vytvoření rozhraní pro práci s nimi v jazycích jako *Java*, *JavaScript* nebo *C++*. Nejjednodušší přístup bylo zacházet s *XML* dokumentem jako s objekty. Již existoval záměr vytvořit *DOM* pro *HTML*, rozšíření na *XML* již bylo snadné. Mimo W3C se v tu dobu začal vytvářet jiný způsob *parsování* dokumentu, vedoucí k již zmíněnému *SAX*, *Simple API for XML*.

Jedním z překvapujících faktů bylo, že se *XML* více využívalo v datově orientovaných aplikacích, jako například při serializování dat, než pro popis struktur, což byl původní účel *SGML*. Definování schématu přes *DTD* fungovalo dobře pro popisné struktury. Ale v aplikaci pro datově orientované struktury narážela na své limity. Nemožnost definovat datové typy a to, že *DTD* ve své podstatě nejsou *XML* dokumenty, bylo viděno jako zásadní problém. Proto byly zahájeny práce na novém standardu, který by toto umožňoval. Opět v rámci konsorcia W3C vznikla pracovní skupina, která přišla s řešením, v roce 2001 vyšla první verze standardu *W3C XML Schema Language*. Tento standard byl opět velice komplexní. Z toho důvodu vznikly další standardy jako například *RELAX NG* nebo *Schematron*.

Vzniklo spousty dalších specifikací, vyjmenujme aspoň několik z nich. *XML Query Language*, určený k získávání informací dle zadaných kritérii z XML dokumentů. *Canonical XML* což je standardní algoritmus pro porovnání, zda jsou dva XML dokumenty stejné. *XInclude*, který slouží k sjednocení více „*well-formed*“ *XML* dokumentů do jednoho. *XML Signatures*, standard pro digitální podepisování a autentifikaci *XML* dokumentů. Rozšíření existuje mnoho a s trochou nadsázky lze říct, že existují nástroje pro všechny potřebné operace s *XML*.

1.5 Pravidla jazyka XML

V následující části bude vysvětleno a ukázáno, jak se tvoří jednoduché *XML* dokumenty. Bude ukázáno, že *XML* dokument je tvořen elementy. Obsahem elementu jsou značky, takzvané *tagy*. Dále je element tvořen obsahem, tedy samotnou informace, kterou obsahuje. Obvykle je ohraničen začáteční a ukončovací značkou, ale existují i prázdné značky, které nemají obsah. Obecně značky mohou vypadat například takto *<nadpis>*. Vypadají tedy přesně jako *HTML* značky. V *HTML*, jak již bylo řečeno, jsme nicméně omezeni pevným počtem značek, které jsou navíc omezeny svým účelem k popsání webové stránky. V *XML* můžete vytvořit značky dle potřeby a tyto značky budou spíše popisovat obsah a ne způsob formátování nebo informace o zobrazení. V *XML* se neřeší, zda je text tučně nebo kurzívou, z principu se jedná spíš o informace typu, zda se jedná o knížku nebo časopis.

Ačkoliv *XML* nabízí oproti *HTML* volnost při tvorbě značek, má mnohem přísnější pravidla, kde a jak mohou být značky zapsány. V zásadě každý *XML* dokument musí být „*well-formed*“, musí splňovat pravidla a to například že má jeden kořenový element, každá startovací značka musí mít ukončovací značku a hodnoty atributů v uvozovkách. Tyto pravidla jsou pevná, což usnadňuje *parsování* dokumentu, i když naopak lehce znesnadňuje vytváření dokumentů.

Jak již bylo řečeno, *XML* je textový dokument, obsahuje tedy textová data, nikdy ne binární. Může být otevřen ve všech programech, které umí číst textové dokumenty. Následující příklad *Příklad 2* je ukázkou kompletního a „*well-formed*“ dokumentu.

```
<jméno>
      František Novák
</jméno>
```

Příklad 2 - jednoduchý XML dokument

Toto může být obsah celého dokumentu, pojmenovaného například *jmeno.xml*. Nicméně není nutné pojmenovávat soubory příponou *.xml*, klidně se může jednat o soubor s příponou *.txt* nebo libovolnou. Operační systém z toho možná nebude úplně nadšený, například Windows určuje typ souboru právě podle přípony, ale *XML parseru* toto nevadí. Dokument ani nemusí být uložen na disku. Může se jednat o záznam nebo pole v databázi. Nebo může být generován za běhu. Dokonce může být uložen ve více souborech, i když pro takto jednoduchý případ je to velice nepravděpodobné.

Nyní se dostáváme k syntaktické části. *Příklad 2* se skládá z jednoduchého elementu pojmenovaného *jméno.* Tento element začíná začátečním znakem *<jméno>* a je ukončen koncovým znakem *</jméno>*. Vše mezi počátečním a koncovým tagem se nazývá obsah elementu. V tomto příkladu je obsahem textový řetězec *František Novák.* Prázdné znaky před a za jsou součástí obsahu, ačkoliv hodně aplikací je ignoruje.

Značka neboli *tag* vypadá velice podobně jako v jazyce *HTML*. Počáteční značka začíná *<* a koncová *</*. Obě dvě jsou následovány jménem elementu a ukončeny znakem *>.* Na rozdíl od *HTML* si zde uživatel může vytvářet značky dle libosti. Názvy si uživatel volí sám. *XML* umožňuje používat prázdné elementy. Prázdný element má svou vlastní speciální konstrukci. Začíná *<* a končí */>.* Prázdný tag *jméno* z příkladu *Příklad 2* by tedy měl tvar *<jméno />*. To je jeden z rozdílů oproti *HTML*, které běžně používá nepárové neprázdné znaky, například HTML tag *
.* V *HTML* běžný znak, ovšem ve formátu *XML* by se musel zapsat *
,* jinak by se jednalo o nevalidní *XML* dokument. Dalším rozdílem oproti *HTML* je to, že *XML* je tzv. *case sensitive*. Tedy v zápise záleží na velikosti písmen. Značka *<jméno>* není to stejné jako *<Jméno>.* Možné jsou oba zápisy, jen je potřeba dodržovat konzistence.

A nakonec pravidlo, že každý element, kromě kořenového, musí být celý součástí jiného elementu. Tedy není možné překrývat značky. Toto pravidlo vychází ze struktury *XML*. *XML* dokument je svoji strukturou strom, proto musí mít jeden kořenový element a každá značka musí být součástí právě jedné větve stromu. O struktuře *XML* dokumentu pojednává další kapitola.

1.6 Struktura XML dokumentu

Abychom pochopili strukturu *XML* dokumentu, ukažme si nejdříve o něco složitější příklad, než ten uveden v příkladu *Příklad 2*.

```
<osoba>
      <jméno>
            <křestní>František</křestní>
            <příjmení>Novák</příjmení >
      </jméno>
      <profese>programátor</profese>
      <profese>designer</profese>
</osoba>
```

Příklad 3 - složitější XML dokument

Popišme si nejdříve, co se na *Příkladu 3* nachází. Tento *XML* dokument se skládá z jednoho elementu *osoba*. Ovšem tento element tentokrát neobsahuje pouze textový obsah, ale obsahuje i elementy vložené do něj, takzvané potomky. Konkrétně se jedná o elementy *jméno* a *profese*. Element *jméno* dále obsahuje potomky *křestní* a *příjmení*. Element *osoba* je rodičovský element elementů *jméno profese*. Stejně jako element *jméno* je rodičovským elementem elementů *křestní* a *příjmení*.

Každý rodič může mít více potomků. Ale na rozdíl od lidského světa má každý potomek právě jednoho rodiče. Výjimkou je pouze kořenový element, který pochopitelně rodiče nemá. Ostatní elementy jsou tedy kompletně součástí jiného element. Jinak řečeno, pokud je počáteční značka elementu v nějakém elementu, musí být i koncová značka součástí tohoto elementu. Zakázáno v *XML* je i prolínání značek. Zápis *<osoba><jméno>František</osoba></jméno>* je v *XML* neplatný. Což opět vychází z pravidla, že element musí končit v elementu, ve kterém začínal, ale element *jméno* začíná v elementu *osoba*, ale končí mimo tento element.

Kořenový element již byl zmíněn. Každý *XML* dokument má právě jeden element, který nemá žádného rodiče, říká se mu kořenový element. Jedná se o první element v dokumentu a obsahuje v sobě všechny další elementy. V příkladu *Příklad 2* je kořenovým elementem element *jméno,* v příkladu *Příklad 3* se jedná o element *osoba*. A vzhledem k tomu, že všechny elementy kromě kořenového mají právě jednoho rodiče a elementy se nemohou překrývat, tak *XML* dokumenty tvoří datovou strukturu v informatice nazývanou strom. Datová struktura strom má výhodné vlastnosti, mimo jiné se dá dobře procházet. Existují například algoritmu na procházení stromu do hloubky nebo do šířky, čehož se dá opět výhodně využít.

Na závěr ještě zmínka o *mixed content*. Z předchozích případů, by mohlo vypadat, že pouze listový element může mít nějaký obsah uvnitř značek a ostatní elementy mohou mít jako obsah pouze další elementy. Ovšem není to tak, v *XML* existuje *mixed content*. V překladu něco jako promíchaný obsah. Znamená to, že rodičovské elementy mohou obsahovat i data. Vše nejlépe ukáže následující příklad.

```
<článek>
     Tento článek napsal <jméno>František Novák</jméno>
     Autor je studentem <odkaz>TUL</odkaz>
</článek>
```

Příklad 4 - mixed content

I tato konstrukce je platná v *XML*. Využívá se například při psaní zpráv, novinových článků, esejí, webových stránek a tak dále. Použití není tak časté a práce s takovým dokumentem je složitější, ovšem i to demonstruje flexibilitu a možnosti využití *XML*.

1.7 Validace dokumentu

Jazyk XML je velmi flexibilní. Programy, které dokumenty *XML* čtou, již tak flexibilní být nemusí a mohou vyžadovat určitou specifickou strukturu tohoto dokumentu. V rámci práce s *XML* dokumenty je tedy někdy vyžadováno zkontrolovat, zda zpracovávaný dokument odpovídá požadavkům programu, který tento dokument zpracovává. Například jazyk *XHTML*, což je zjednodušeně řečeno úprava jazyka *HTML* taková, aby odpovídal pravidlům jazyka *XML*, tedy byl „*well-formed*“, umožňuje použití značky *li* pouze uvnitř značek *ul* nebo *ol*. Takovéto pravidla umožňuje popsat jazyk *DTD*. Zápis dokumentů *DTD* umožňuje formálně popsat strukturu *XML* dokumentu, například tedy říci, že element *li* musí potomkem elementu *ol* nebo *ul*, jinde je jeho použití neplatné a dokument není validní. Různé programy samozřejmě mohou používat různé *DTD* předpisy. Program provede validaci pomocí validačního *parseru*, který dokument projde a zkontroluje, zda odpovídá *DTD* předpisu. Pokud neodpovídá, záleží na programu, zda dokument odmítne, bude chyby ignorovat nebo se pokusí chyby opravit. Tato validace je volitelná a na rozdíl od kontroly, zda je dokument „*well-formed*“, se nejedná nutně o fatální chybu a je možné dokument i přesto zpracovávat.

Pokud tedy dokument obsahuje definici *DTD* je validní právě tehdy, pokud jeho obsah vyhovuje pravidlům zapsaným v tomto *DTD*. Validace funguje na principu vše, co není dovoleno, je zakázáno. Tedy definice *DTD* musí obsahovat vše, co může být v dokumentu obsaženo. Pokud dokument tomuto předpisu vyhovuje, je validní, v opačném případě je nevalidní.

Existuje řada případů, které *DTD* postihnout neumí. *DTD* například neumí říct, jaký je kořenový element, neumí říct, kolikrát se daný element může v dokumentu objevit ani jak data uvnitř elementu vypadají. Neumí také říct, zda je obsahem elementu datum nebo jméno nějaké osoby. *DTD* popisuje pouze strukturu dokumentu, neříká nic o délce, významu, struktuře ani povolených hodnotách obsahu elementů. Pro tyto účely existují další nástroje a jazyky. Například *W3C XML Schema*, o kterém se zmíníme později.

1.7.1 Validace pomocí DTD

V této části se budeme věnovat validaci pomocí předpisu *DTD*. Tento předpis tedy umožňuje definovat strukturu *XML* dokumentu a validovat ho tímto předpisem. Využijme ukázku *Příklad 3* a vytvořme předpis, který by tento dokument validoval. Na následující ukázce je jedna z možností, jak takový předpis zapsat.

```
<!ELEMENT osoba       (jméno, profese*)>
<!ELEMENT jméno       (křestní, příjmení)>
<!ELEMENT křestní     (#PCDATA)>
<!ELEMENT příjmení    (#PCDATA)>
<!ELEMENT profese     (#PCDATA)>
```

Příklad 5 - zápis DTD validace

Takovýto předpis by byl pravděpodobně uložen v jiném souboru a přikládal by se ke konkrétnímu *XML*. V tomto konkrétním případě každý řádek deklaruje jeden element, obecně to tak být nemusí a definice elementu může být víceřádková. Jsou zde tedy definovány elementy *osoba*, *jméno*, *křestní*, *příjmení* a *profese.* Dále je zde řečeno, že element *osoba* obsahuje elementy *jméno* a *profese.* V obsahu elementu *osoba* musí být jeden element *jméno*, a může, ale nemusí být různý

počet elementů *profese*. To značí znak *hvězdičky* za názvem elementu *profese*. Zápis dále říká, že element *jméno* musí být první a elementy *profese* musí následovat až za ním. Druhý řádek říká, že element *jméno* musí mít potomky *křestní* a *příjmení*. Následující tři řádky říkají, že obsahem elementů *křestní*, *příjmení* a *profese* jsou *#PCDATA,* jinak řečeno že obsah je formě textu, který už neobsahuje žádné značky nebo potomky. Na pořadí řádků v předpise nezáleží.

Takový to *DTD* předpis se do XML dokumentu vkládá pomocí značky *!DOCTYPE,* názorně je to ukázáno na následujícím příkladu.

```
<!DOCTYPE osoba SYSTEM "http://www.mojedtd.cz/osoba.dtd">
```

Příklad 6 - vložení DTD

Tento zápis říká, že kořenový element následujícího dokumentu je element *osoba* a dané DTD je k dispozici na *URI* adrese *http://www.mojedtd.cz/osoba.dtd.* Tato deklarace musí být umístěna za deklaraci *<?xml … ?>* a před kořenový element. Uveďme si nyní řadu příkladů, které názorně předvedou, jak zápis *DTD* funguje na ukázce validních a nevalidních dokumentů.

```
<!DOCTYPE osoba SYSTEM "http://www.mojedtd.cz/osoba.dtd">
<!-- první validní element -->
<osoba>
      <jméno>
           <křestní>František</křestní>
           <příjmení>Novák</příjmení>
      </jméno>
      <profese>programátor</profese>
      <profese>designer</profese>
 </osoba>
<!-- druhý validní element -->
<osoba>
     <jméno>
          <křestní>František</křestní>
          <příjmení>Novák</příjmení>
     </jméno>
</osoba>
```

Příklad 7 - validní XML dokument s DTD

Příklad 7 je příkladem validního *XML* dokumentu, obsahuje vše přesně tak, jak bylo definováno v *Příkladu 5*. První element *osoba* obsahuje elementy *jméno* a libovolný počet elementů *profese*. Element *jméno* obsahuje elementy *křestní* a *příjmení*. A konečně elementy *křestní*, *příjmení* a *profese* obsahují již pouze data a žádné vnořené elementy. Validní je druhý element *osoba* v příkladu, protože dle předpisu nemusí obsahovat element *profese.* Naopak následující příklad je nevalidní *XML* dokument, vzhledem k zadanému DTD.

```
<?xml version="1.0" standalone="no">
<!DOCTYPE osoba SYSTEM "http://www.mojedtd.cz/osoba.dtd">
<!-- první nevalidní element -->
<osoba>
      <profese>programátor</profese>
      <profese>designer</profese>
</osoba>
<!-- druhý nevalidní element -->
<osoba>
      <profese>programátor</profese>
      <jméno>
            <křestní>František</křestní>
            <příjmení>Novák</příjmení>
      </jméno>
      <profese>designer</profese>
</osoba>
```

Příklad 8 - nevalidní XML dokument

Dokument v *Příkladu 8* je nevalidní, protože element *osoba* neobsahuje element *jméno*. Podobně je druhý element *osoba* nevalidní, protože element *profese* je před elementem *jméno*.

Syntaxe *DTD* umožňuje definovat další možnosti, jedná se o poměrně obsáhlý formát a náplní této práce není dopodrobna rozebírat možnosti *DTD*. Přechozí informace jsou čerpány z publikace *XML in Nutshell* [1], kde se problematice *DTD* věnuje celá kapitola 3.

1.7.2 Validace pomocí XML Schema

DTD jsou široce používané a podporované. Přesto vývojářům možnosti *DTD* přestaly brzo stačit. Konsorcium *W3C* se rozhodlo vytvořit nový jazyk, který by popisoval *XML* dokumenty, jejich strukturu a také obsah. Byla tedy založena pracovní skupina v rámci *W3C* která se nazývala *XML Schema Working Group*. Tato skupina přišla po dvou letech se dvěma tzv. doporučeními. *XML Schema část 1: Struktury* a *XML Schema část 2: Datové typy*. Uveďme si na začátek příklad *XML* dokumentu, který budeme dál v této kapitole používat.

```
<?xml version="1.0">
<knihovna>
      <kniha id="k005891" k_dispozici="true">
            <isbn>005891</isbn>
            <název jazyk="cz">Kuchařka II.</název>
            <autor id="123">
                  <jméno>Karel Pospíšil</jméno>
                  <narození>12.05.1974</narození>
                  <profese>Kuchař</profese>
            </autor>
      </kniha>
            <kniha id="k005897" k_dispozici="false">
            <isbn>005897</isbn>
            <název jazyk="cz">Encyklopedie</název>
            <autor id="258">
                  <jméno>Martin Novák</jméno>
                  <narození>12.05.1949</narození>
                  <smrt>18.09.2001</smrt>
            </autor>
      </kniha>
</knihovna>
```

Příklad 9 - ukázkový XML dokument pro validaci XML Schema

Předchozí příklad berme jako validní *XML* dokument. Z příkladu je jasné, že se jedná o hypotetický seznam knih nějaké knihovny. Každá kniha má *ISBN*, *název* a jednu nebo více *postav*. Každý *autor* má *jméno*, *datum narození* a volitelně *datum úmrtí* a *profesy*. Kniha má atributy *id* a k_*dispozici*. *Název* má atribut *jazyk*. Nyní si ukážeme, jak vytvořit *XML Schema* popisující tento dokument. Existuje více přístupů, začneme stylem bližším *DTD*. Nejdříve si popíšeme všechny seznam elementů a atributů. Definice je k dispozici na adrese *http://www.w3.org/2001/XMLSchema* a tento jmenný prostor obvykle bývá označován prefixem *xs* neb *xsd.* Ale není to pravidlo a záleží na autorovi, jako prefix zvolí. Popis, obsažený v těle *XML Schema* definice charakterizuje, o jaký typ elementu se jedná, a listové elementy, které zde mohou být obsaženy. Nebere ohled na atributy. Existují následující typy elementů. P*rázdný*, pokud není očekáván jako obsah žádný element ani text. J*ednoduchý* element, pokud obsahuje pouze text. Dále *komplexní,* pokud obsahuje pouze elementy, a neobsahuje text. A konečně *smíšený,* který obsahuje jak text, tak i elementy. Zatím nás zajímají pouze elementy. Atributy v této fázi neřešíme. Ignorují se také komentáře a procesní instrukce.

Nyní si rozebereme, jaké vlastně *Příklad 9* obsahuje typy elementů. Elementy *jméno*, *narození*, *smrt*, *profese* a *isbn* jsou *jednoduchého* typu. Obsahem je tedy pouze text, nikoliv vložené elementy. Následuje ukázka definice jednoho takového elementu.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="jméno" type="xs:string" />
</xs:schema >
```

Příklad 10 - zápis elementu v XML Schema

Tedy *element* se definuje klíčovým slovem *element* a má atributy *name*, který značí název elementu, a atribut *type.* Existují předdefinované typy, které jsou v rámci *XML Schema* k dispozici. Také ony se prefixují. Podobně se definice provede pro zbytek elementů jednoduchého typu. S jedinou modifikací a to je správně určit typ. Elementy *narození* a *smrt* budou zjevně mít formát datumu, takže

se bude jednat datový typ *xs:date.* Podobně toto funguje i s atributy. Zde například atribut *k_dispozici* bude mít typ *boolean*, tedy zápis typu bude *xs:boolean*. Následuje ukázka zápisu atributu.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:attribute name="k_dispozici" type="xs:boolean" />
</xs:schema >
```

Příklad 11 - zápis atributu v XML Schema

Je potřeba zmínit, že na rozdíl od *DTD*, se atributy definují samostatně, ne jako součást elementu. Díky čemuž je s nimi v rámci *XML Schema* zacházeno stejně jako s elementy. Pořadí, v jakém se definice zapisují, není důležitá. Nyní je potřeba definovat, k jakému elementu daný atribut náleží, takový element je již potřeba definovat jako *komplexní* typ. Definice je následující.

```
<xs:element name="název">
     <xs:complexType>
          <xs:simpleContent>
               <xs:extension base="xs:string">
                    <xs:attribute ref="k_dispozici "/>
               </xs:extension>
          </xs:simpleContent>
     </xs:complexType>
</xs:element>
```

Příklad 12 - přiřazení atributu k elementu

Zápis v *Příkladu 12* v lidské řeči znamená, že element *název* je komplexního typu, jednoduché typy nemohou mít atributy. Proto se tento element definuje jako komplexní typ a rozšiřuje se pomocí značky *extension*. Zbylé elementy jsou již komplexní, tedy obsahují v sobě další elementy a ne textová data.

Základy byly vysvětleny, nyní by již nic nemělo bránit vytvořit kompletní *XML Schema*, validující *Příklad 9*.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="jméno" type="xs:string"/>
      <xs:element name="profese" type="xs:string"/>
      <xs:element name="narození" type="xs:date"/>
      <xs:element name="smrt" type="xs:date"/>
      <xs:element name="isbn" type="xs:string"/>
      <xs:attribute name="id" type="xs:string"/>
      <xs:attribute name="jazyk" type="xs:string"/>
      <xs:element name="název">
      <xs:complexType>
      <xs:simpleContent>
      <xs:extension base="xs:string">
      <xs:attribute ref="jazyk"/>
      </xs:extension>
      </xs:simpleContent>
      </xs:complexType>
      </xs:element>
            <xs:element name="knihovna">
                  <xs:complexType>
                  <xs:sequence>
      <xs:element ref="kniha" />
      </xs:sequence>
            </xs:complexType>
            </xs:element>
            <xs:element name="autor">
                  <xs:complexType>
                        <xs:sequence>
                              <xs:element ref="jméno"/>
                              <xs:element ref="narození"/>
                        </xs:sequence>
                        <xs:attribute ref="id"/>
                  </xs:complexType>
            </xs:element>
            <xs:element name="kniha">
                  <xs:complexType>
                        <xs:sequence>
                              <xs:element ref="isbn"/>
                              <xs:element ref="název"/>
                              <xs:element ref="autor"/>
                        </xs:sequence>
                  <xs:attribute ref="id"/>
                  <xs:attribute ref="k_dispozici"/>
            </xs:complexType>
      </xs:element>
</xs:schema>
```

Příklad 13 - ukázka zápisu XML Schema

Takto by tedy mohl vypadat *XML Schema* předpis, pro validaci *XML* dokumentu, uvedeného v začátku této kapitoly. Všimněme si několika věcí. *XML Schema* je modulární. V předchozím příkladu jsme definovali jednoduchý elementy a ty poté použili vytvoření elementů komplexních. Dále, na rozdíl

od *DTD*, definuje jak strukturu, tak datové typy. Samotné doporučení *W3C* je rozděleno na dvě části. Na část strukturní a část o datových typech.

Z příkladu by tedy mělo být zřejmé, jak jsme postupovali. Definovali jsme jednoduché elementy a z nich potom vytvořili komplexní. Takovýto přístup se nazývá *plátkování.* Princip *plátkování* je tedy takový, že na nejvyšší úrovni se definují všechny prvky a atributy. Takovéto komponenty se nazývají *globální*. Z čehož plynou jejich vlastnosti. Může na ně být odkazováno kdekoliv v tomto schématu, co víc, dokonce na ně může být odkazováno v jiném schématu, ve kterém je toto schéma vloženo. Odkazování probíhá pomocí atributu *ref.*

Druhým postupem, jak vytvářet schémata je postup zvaný *Matrjoška,* anglicky *Russian Doll*. Tento přístup nedefinuje elementy a atributy globálně, ale prvky se do sebe ve schématu vkládají přímo. Tedy v tomto schématu je jeden kořenový prvek a definice každého elementu je vložena do svého předka. Tedy z objektového hlediska jsou všechny definice lokální. Vytvoření takovéhoto předpisu má výhodu v krátkém a kompaktním zápisu. Nevýhodou je naopak, že pro složitější jazyky je nepřehledný a nelze opakovaně využít dílčí definice.

A konečně další možným postupem je takzvaný *Slepý Benátčan*. Princip je takový, že se předem na globální úrovni definují typy. Poté se prvky definují lokálně, jako při postupu *Matrjoška,* ovšem triviálněji, protože využívá vytvořené typy. Tento systém je nejflexibilnější, je možné opakovaně využívat již vytvořené typy. Je vhodný spíše pro složité jazyky.

Cílem této práce není detailně popsat, jak vytvářet *XML Schema* definice, cílem je spíše ukázat možnosti, které tento jazyk nabízí. V *Příkladu 10* je ukázáno použití atributu *type*, který slouží k určení typu obsahu elementu. *XML Schema* obsahuje řadu předdefinovaných typů, jednoduchých a i složitějších, které se z nich odvozují. Jednoduché typy mají specifický význam, a proto nemohou být odvozeny od jiných typů. Nejobecnější je typ *anySimpleType*. Jedná se typ, který akceptuje jakoukoliv hodnotu a neklade na hodnoty žádné podmínky. Nelze z něj derivovat uživatelský typy a jeho přesná forma není ani stanovena ve specifikaci. Tento typ by neměl být příliš používán.

Nyní si v krátkosti probereme vybrané předdefinované jednoduché typy a z nich typy odvozené. První typem budiž typ *string*. Jedná se datový typ uchovávající text. Tento datový typ jako jediný neprovádí nahrazení prázdných znaků mezerou a zachovává je. Ostatní typy nahrazují znaky tabulátor, znak nového řádku a znak pro návrat na začátek řádku znakem mezera, a zachovává tak přesně podobu řetězce. Odvozeným typem od tohoto typu je typ *normalizedString*. U tohoto typu se provádí nahrazení prázdných znaků znakem mezera, ale zanechává počet těchto znaků. Ostatní typy nahrazují tyto znaky, aby nebyly na začátku a konci slova a zároveň maximálně jednou za sebou. Typ, který toto provádí, je *token*. Jak již tedy bylo řečeno, provádí nahrazení prázdných znaků mezerou a zároveň odstraňuje více prázdných znaků za sebou. Mezi dalšími již pouze jmenujme typ *anyURI*, který slouží, jak již název napovídá, k uchování adresy URI. Dále typy jako *language*, *Name* a *ID*. Nakonec textových typů ještě uveďme typy *hexBinary* a *base64Binary*. Tyto typy slouží k uchování binárního obsahu, který musí být zakódován do řetězce tisknutelných znaků, z důvodů, které byly vysvětleny na začátku. *XML Schema* tedy definuje dva. *HexBinary* definuje jednoduchý postup, jak zakódovat každou osmici bitů do dvou hexadecimálních číslic. A jako druhý nabízí formát *base64Binary*. Toto kódování odpovídá kódování známému jako *base64.* Toto kódování mapuje šestibitové skupiny na 64 tisknutelných znaků.

Jazyk *XML Schema* samozřejmě také podporuje číselné datové typy. Všechny numerické typy se odvozují ze čtyř základních. Jedná se o *decimal*, *double*, *float* a *boolean*. Typ *decimal* reprezentuje desetinná čísla. Počet míst není omezen. Oddělovač desetinných míst je vždy desetinná tečka a první znakem může být *+* nebo *-*. Povolen není ani vědecký zápis „*E+3*“. Z typu *decimal* jsou odvozeny další typy. Jedná se například o typ *integer*, ze kterého se dále odvozují typy *nonPositiveInteger*, *nonNegativeInteger* a například *long*.

Typy *float* a *double* jsou oba také základní datové typy. Reprezentují číselný typ s plovoucí desetinou čárkou. Tedy skládají se z mantisy a exponentu. Což umožňuje uložit velká čísla s fixní velikostí na disku, která je 32, respektive 64 bitů. Tyto datové typy akceptují několik speciálních hodnot, jako například plus nekonečno (*INF*), mínus nekonečno (*-INF*) nebo hodnotu říkající že se nejedná o vyjádřitelné číslo (*NaN*).

A nakonec typ *boolean*. Jedná se o velice jednoduchý typ, který akceptuje pouze hodnoty *true* (1) a *false* (0).

Ještě uveďme další velice používaný typ a to typ *dateTime*. Tento typ, jak už název napovídá, slouží k uchování informace o datu a času. Tento datový typ se plně řídí definicí *ISO 8601*. A tato definice je jediná akceptovaná jazykem *XML Schema*. Dalšími typy pro práci s časem a datem jsou například *date*, *gYearMonth*, *gYear* nebo *time*. Popisem práce s těmito datovými typy se zabývá kapitola 4.5 v publikaci *XML Schema*[2].

Jazyk *XML Schema* nabízí další předdefinované typy, jako například typy pro práci s polem prvků oddělených mezerou a spousty dalších. Umožňuje také definovat vlastní typy. Ať už omezením rozsahu již definovaných nebo například definováním povolené hodnoty ve výčtovém typu. *XML Schema* je silný nástroj a není cílem této práce rozebrat vše, co umí. Tvorbou vlastních datových typů se zabývají kapitoly 5, 6 a 7 v již zmíněné publikaci[2].

1.8 Transformace XML

O transformaci *XML* dokumentů byla zmínka už v *Úvodu* a v kapitule *Úvod do jazyka XML*. Transformace znamená, že se vstupní dokument převede na výstupní v požadovaném formátu. K tomu se využívá určitý předpis. Tento předpis, vzhledem k tomu, že se pohybujeme v jazyce *XML*, by měl ideálně mít také podobu *XML* dokumentu. Jazyk *XML* se stal hned po vydání široce používaným. Jeho flexibilita v popisu strukturovaných dat byla ideální pro přenos dat mezi aplikacemi, databázemi a tak dále. K čemu je tedy dobré dokument transformovat? Jak již bylo řečeno, *XML* je jazyk vytvořený k popisování dat. Pomocí značek popisuje strukturu a význam jednotlivých částí dokumentu. Umožňuje tedy nějak obecně popsat data a ty pak předat dál. Součástí *XML* dokumentu může být i *DTD*, případně *XML Schema*, které popisuje jak má dokument vypadat a i lidský čtenář z něj může pochopit strukturu dokumentu. Aplikace ale mohou používat různé *XML* jazyky. V první fázi se k práci s *XML* dokumenty používali *parsery*. Ať již se jednalo o *DOM* nebo *SAX* přístup. Ovšem tento přístup znamenal další mezikrok ve zpracování dokumentů a uživatel nebyl schopen vidět, co se v programu zpracovávající tento dokument děje. Program dokument načetl, zpracoval informace, upravil si je dle potřeby a dál s nimi tak pracoval. Programátor musel pro každý nový formát dokumentu napsat víceméně speciální část kódu, který toto zpracuje do formátu, se kterým už bude umět daná aplikace pracovat. Naskytla se otázka, zda by se rovnou nedalo dokument upravit a potom pracovat už s ním, bez nutnosti

mezikroku. Jako řešení se hodilo vytvořit flexibilnější nástroj se širokými možnostmi zpracování a transformování dokumentů. A zároveň něco dostatečně jednoduchého, pro široké použití. Jedním z nejpoužívanějších je jazyk *XSLT*. Ale existují i další, pro příklad uveďme *XQuery*, *XML Document Transform*, *STX* a třeba *XProc*. Široce používaná je například transformace z *XML* do *HTML*. Jazyk *HTML* není *XML* jazyk, jeho původní definice nesplňovala všechny požadavky jazyka, například povolovala neukončené znaky. Touto transformací je možné z *XML* dokumentu snadno vytvořit *HTML* dokument a prezentovat ho třeba na webu.

1.8.1 Transformace pomocí jazyka XSLT

XSLT, neboli *anglicky Extensible Stylesheet Language Transformation*, existuje jako oficiální doporučení *W3C*, které ho vydává a spravuje jeho verze. Poskytuje flexibilní a mocný nástroj pro převod *XML* dokumentů na něco jiného. Ať už se jedná o dokument ve formátu *HTML*, *PDF* nebo *SVG*. Může se také jednat o kód v jazyce *Java*, případně čistě textový výstup. Uživatel napíše *XSLT* předpis, kde definuje pravidla převodu dokumentu a *XSLT* procesor udělá zbytek.

Ještě je dobré dodat, že konsorcium *W3C* definuje dva typy standardů pro předpis *XML*. Nejstarší a nejjednodušší jsou Kaskádové styly, v angličtině *Cascading Style Sheets* (*CSS*). Jedná se o způsob, jak přidat značkám různé vlastnosti. *CSS* se převážně používá pro stylování *HTML* dokumentů, ale lze ho použít i s *XML* dokumentem. *CSS* například umí říct, že ten a ten element bude vytisknut s modrým pozadím a nějaký jiný obsah elementu například tučně. Tuto práci vykonává dobře. Ovšem je spoustu věcí, které neumí. *CSS* například neumí měnit pořadí prvků tak, jak se objeví v dokumentu, neumí tedy řadit prvky podle nějaké vlastnosti. Neumí také dělat výpočty, například sečíst hodnotu všech konkrétních prvků a zobrazit ji. Neumí spojovat dokumenty a vytvořit z nich jeden výstup. *CSS* je určeno k úpravě zobrazení dokumentu, nic víc od něj nebylo očekáváno.

Z toho důvodu byl vytvořen právě *XSLT* jazyk, tedy umožnit transformaci, změnu pořadí elementů, případně umožnit provádět v dokumentu výpočty. Základní teze *XSLT* jsou následující. *XSLT* předpis je opět *XML* dokumentem, splňuje tedy pravidla *XML*, z čehož vyplývá možnost, napsat předpis, který transformuje předpis. Principem XSLT je porovnávání, tedy funguje na principu „pokud najdeš tento element, proveď s ním tohle“. Dále by *XSLT* mělo být navrženo tak, aby nezpůsobovalo vedlejší efekty. Tedy, že na jeden dokument může být zároveň aplikováno více předpisů. Největším dopadem tohoto pravidla je, že proměnné nemůžou být změněny, jakmile jsou jednou inicializovány. Pokud by to šlo, mohla by změna proměnné při aplikování jednoho předpisu ovlivnit průběh zpracování jiným předpisem. *XSLT* je silně ovlivněno funkcionálním programování. Zkráceně funkcionální programování pracuje na principu definování posloupnosti funkcí, které na korektní vstup vrátí korektní výstup, který může další funkce použít jako korektní vstup a vrátit díky tomu zase korektní výstup. Jazyk *XSLT* k procházení a transformaci nepoužívá klasické *for* a *do-while* cykly. Místo toho používá *rekurzi* a *iteraci*. *Iterace* v principu znamená „vem všechny prvky, které vypadají tak a tak a udělej s nimi toto“. Rekurze znamená opakované vnořené volání, jinak řečeno tedy pokud je potřeba k výsledku volání znát nějakou hodnotu vracenou stejnou funkcí, zavolá se znovu. Jako příklad se obvykle uvádí faktoriál, algoritmus je stylem „mám číslo, vynásobím ho číslem o jedna menším, pokud se nejedná o číslo jedna, tak opět vynásobím číslem o jedna menším, pokud je to jedna, vrátím výsledek“.

Jmenujme pár praktický případů, kde se hodí využití *XSLT*. Například doručení informací různým aplikacím, představme si webovou stránku, která se bude zobrazovat na různých zařízeních. S využitím *XSLT* může programátor napsat pouze jednu stránku a poté dle potřeby ji transformovat, aby odpovídala potřebám. S využitím jazyka *XSLT*, bez potřeby zasahovat do zdrojového kódu. Dalším příkladem může být výměna dat mezi různými databázemi. Vyexportovaná data převedeme do formátu, který bude kompatibilní s cílovou databází. Lze také vytvořit z *XML* pomocí transformace rovnou *SQL* příkazy, případně převést dokument například do formátu *CSV*, anglicky *comma-separated values*, hodnoty oddělené čárkami. Jedná se o přepis tabulky do hodnot oddělených čárkou, první řádek obvykle definuje význam každého sloupce.

Nyní si ukažme, jak se vytváří jednoduchý *XSLT* předpis a jak funguje. Začněme příkladem *XML* dokumentu, který chceme transformovat.

```
<?xml version="1.0"?>
<pozdrav>
      Ahoj světe!
</pozdrav>
```

Příklad 14 - příklad XML pro transformaci

V *Příkladu 14* je jednoduchý *XML* dokument, který obsahuje značku *pozdrav* a obsahem této značky je text. Nyní by nás mohlo napadnout, transformovat tento *XML* dokument třeba do *HTML*, aby se dal zobrazit v prohlížeči. Vytvořme následující předpis této transformace.

```
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
      <xsl:output method="html"/>
      <xsl:template match="/">
            <xsl:apply-templates select="pozdrav"/>
      </xsl:template>
      <xsl:template match="pozdrav">
            <html>
            <body>
            <h1>
                  <xsl:value-of select="."/>
            </h1>
            </body>
            </html>
      </xsl:template>
</xsl:stylesheet>
```

Příklad 15 - zápis transformace XSLT

Pokud provedeme transformaci *Příkladu 14* pomocí předpisu definovaného v *Příkladu 15,* bude výsledek vypadat následovně.

```
<html>
<body>
<h1>
      Ahoj světe!
</h1>
</body>
</html>
```

Příklad 16 - výsledek transformace Příkladu 18

Tento dokument je už možné zobrazit ve webovém prohlížeči a výsledkem bude bílá stránka s tučným nápisem „Ahoj světe!“. Popišme si nyní s využitím předchozího příkladu, jak zpracování takové transformace probíhá. Nejdříve si program, provádějící transformaci, načte celý transformační předpis a převede ho na stromovou strukturu. Také *XML* dokument, který má být transformován, je převeden na stromovou strukturu. První řádek *XSLT* definuje, že se jedná o *XSLT* předpis a říká, kde je definice jeho jmenného prostoru. Další řádek oznamuje programu, jaký očekává výstup. Definice *XSLT* definuje tři možné typy výstupu a to *XML*, *HTML* a *text*. *Parser* může nabízet více možností, to je zcela na tvůrci. Následují už konkrétní pravidla transformace.

Program prochází *XML* dokument, začne u kořene dokumentu. Podívá se, jestli existuje v předpise pravidlo pro kořen. Ano existuje. Předpis pro kořen je řádek *<xsl:template match="/">.* K jakému elementu je pravidlo vázané se určuje pomocí atributu *match*. Znak „/“ tedy značí kořen dokumentu. Obsahem je jediný příkaz, jeho významem je „pokud objevíš element *pozdrav*, aplikuj na něj předpis“. *XSLT* procesor tedy začne hledat kořenový element *pozdrav.* Pokud tento element najde, musí s ním provést nějaké operace. Pokud je elementů více, provede operace na všech postupně, jak je nachází. Pokud tedy element najde, podívá se opět do *XSLT* předpisu, jestli je tam pravidlo, jak transformovat element *pozdrav.* Obsahem tohoto předpisu jsou nejdříve tři *HTML* značky. *XSLT* parser si jich nevšímá a reprezentuje je čistě jako text, protože neobsahují *XSLT* jmenný prostor v deklaraci. Do výstupu tedy zapíše tyto tři HTML značky a do nich vloží hodnotu definovanou řádkem *<xsl:value-of select="."/>.* Příkaz *value-of* znamená „zapiš sem hodnotu“, jakou hodnotu se určuje atributem *select.* V tomto případě je jako hodnota uvedeno „.“, to znamená hodnotu obsaženou v momentálně procházeném elementu. Nakonec se ještě doplní ukončující znaky *HTML*. Element *pozdrav* je celý zpracovaný. Procházení se vrátí opět do kořene dokumentu. Vyhodnotí se, že již zde nejsou žádné elementy *pozdrav* a procházení se ukončí. A transformace je hotova.

Transformace *XML* je velice mocný nástroj, pomocí *XSLT* lze využít k nepřebernému množství úloh. Pokrýt všechny možnosti jazyka není v možnostech ani v zadání této práce. Detailní popis možností *XSLT* a postupů vytváření *XSLT* předpisů, je popsán například v publikaci *XSLT* od *Douga Tidwella*[3].

2 Vlastní aplikace pro validaci a transformaci

V předchozích kapitolách byly předvedeny možnosti, které jazyk *XML* nabízí. Byla uvedena řada jazyků umožňující práci s *XML* dokumenty. Nabízí se tedy, zda existují nástroje, které tyto operace s *XML* umí provádět. Vzhledem k rozšířenosti *XML* je zřejmé, že takových nástrojů bude existovat velké množství. K dispozici jsou knihovny, umožňující práci s těmito dokumenty aplikacím, napsaným ve všech nejpoužívanějších vyšších jazycích a pro všechny běžné operační systémy. Například pro jazyk *C#* je v rámci frameworku *.NET* k dispozici knihovna pro práci s *XML* již v základu, jmenný prostor je *System.Xml*. Jazyk *Java* má také již v základní sadě knihoven zabudovanou podporu práce s *XML*. Mimo to existuje řada knihoven od třetích stran. Jako příklad uveďme *JDOM*, *dom4j* a třeba *Woodstox*. Pro další část práce je podstatné, že pro operační systém *Linux* a odvozené typy existují *open source* knihovny, které podporují v předchozích částech vyjmenované nástroje a jazyky. Tyto knihovny lze využít při tvorbě vlastní aplikace, zpracovávající *XML* dokumenty, dle vlastního návrhu.

Z předchozích kapitol by mělo být zřejmé, že nelze vytvořit aplikaci, která zpracovává obecně všechny *XML* dokumenty. Co ovšem lze vytvořit, a takové nástroje již existují, jsou aplikace, validující a transformující dokumenty. V předchozích kapitolách se často zmiňovala rozšířenost *XML*, takže existence těchto nástrojů je očekávaná. Desktopové aplikace jsou tedy k dispozici. Stejně tak existují online aplikace, které přímo na webu umožňují provádět validace a transformace zadaných *XML* dokumentů. Jedním z bodů zadání této práce je vytvořit online aplikaci pro validaci datových struktur. Nabízí se tedy otázka, proč takovou aplikaci vytvářet, když již existují? A jaký by taková aplikace tedy měla přínos?

Výhodou online aplikací je, že jsou dostupné kdekoliv, kde je dostupné připojení k internetu. Dnes je pokrytí připojením k internetu ve vyspělém světě, s malou nadsázkou, dostupné kdekoliv. To umožňuje již tak velkou flexibilitu *XML* ještě rozšířit. Uživatel může své dokumentu validovat a transformovat odkudkoliv. Využívání veřejných nástrojů s sebou nese jistá bezpečnostní rizika, zejména pokud je obsah dokumentu z nějakých důvodů soukromý. Nelze zaručit, že se zpracovávaný dokument nedostane do cizích rukou. Pokud to situace vyžaduje, je samozřejmě možné napsat vlastní aplikaci a umístit jí také na web. A omezit přístup pomocí aplikování určitých přihlašovacích postupů, omezení přístupu pomocí *VPN* a tak dále. Jedním z cílů této práce, je vytvořit základ modulární aplikace pro zpracovávání *XML* dokumentů. Tedy vytvořit kostru programu, ukázku jak takovou aplikaci navrhnout, a umožnit její snadné rozšíření dle potřeby, například přidáváním podpory nových jazyků a formátů. Jinak řečeno by mělo být možné vzít zdrojový kód této diplomové práce a použít ho jako základ své specifické aplikace. Přidat, případně upravit, jednotlivé části pro své potřeby a zaměřit se na funkcionalitu, která je potřeba, a nemuset vytvářet jádro aplikace a řešit věci jako komunikace, přihlašování klientů, případně provázání jednotlivých stránek.

Toto samotné by samo o sobě nemělo takový přínos. Různých online validátorů a aplikací na transformaci *XML* dokumentů je celá řada. Konsorcium *W3C*, které spravuje standard *XML*, má takové nástroje na svých stránkách k dispozici. Jedná se o online nástroje dostupné na adrese *https://validator.w3.org/*, respektive *http://www.w3schools.com/xsl/*. Existuje nepřeberné množství dalších online nástrojů. Společným rysem většiny těchto nástrojů je to, že buď výsledek rovnou vrátí v okně prohlížeče. Nebo v případě transformace pošle výsledek na zadaný email nebo v lepším případě nabídne stažení výsledného dokumentu. Tyto nástroje zpravidla neumožňují spravovat výsledky provedených validací a transformací a jejich činnost končí vrácením výsledku. Neumožňují shromažďovat výsledky úloh na jednom místě a vracet se k nim dle potřeby. Neumožňují udržovat si

jak výchozí, tak výsledné dokumenty na jednom místě a přistupovat k nim pomocí internetové sítě. Pravidlem nebývá ani podpora dávkových úloh.

Co je tedy praktickým přínosem této práce? Představme si následující modelovou situaci z praxe. Mějme webovou aplikaci, která má za úkol shromažďovat data, například seznam položek internetového obchodu. Tyto položky chceme někde evidovat a umožnit uživateli naší aplikace tyto položky procházet a porovnávat. Zde tedy vyvstává potřeba přenášet data mezi různými systémy. Je tedy potřeba mít data jednotně uložená a znát jejich strukturu. K tomu je ideální využít níže popsané jazyky pro popis dat. V obecnosti samozřejmě nechceme shromažďovat data pouze z jednoho obchodu, ale z více. Dnes je běžné, že takovéto porovnávací webové aplikace mají definovaný takzvaný *XML feed*. Pokud chce majitel nějakého webového obchodu vystavit svoji nabídku na tomto porovnávacím webu, musí poskytnout datový soubor v očekávaném formátu. Tento soubor se většinou generuje automaticky z databáze. Obvykle je právě ve formátu *XML* popisovaném níže. Během generování se mohou vyskytnout chyby, například neexistující povinný údaj, z tohoto důvodu, aby se zabránilo odmítnutí souboru, který může a zpravidla je i velice rozsáhlý, tak se soubor před zpracováním validuje. Porovnávací webové aplikace obvykle nabízejí jednoduchou validaci vstupních souborů zadáním přímo souboru nebo URL adresy, na které se soubor nachází. Ovšem velice často se stává, že provozovatel nemá jeden, ale více internetových obchodů. Generuje se tedy větší množství souborů, které je potřeba validovat. Aplikace popsaná a vyvinutá v rámci této práce umožňuje provádět validace a případně transformace těchto souborů. Navíc umožňuje provádět tyto akce dávkově, tedy zadat více souborů najednou a hromadně je zpracovat. Také umožňuje tyto úlohy naplánovat na konkrétní čas, případně provádět tyto úlohy v pravidelných intervalech. Výsledky úloh jsou poté uloženy a je možné si je prohlédnout nebo stáhnout pro další použití. Tímto se značně usnadňuje správa více webových obchodů najednou.

Kapitolou sama pro sebe je poté již zmíněné pokročilé zpracování úloh. V tomto smysl se pokročilým zpracováním úloh myslí spouštění úloh v daný čas nebo zpracovávání úloh opakovaně se zadanou periodou. Ať už se jedná o spouštění každou hodinu, den či týden. Jako pokročilé zpracování se může brát i dávkové zpracování úloh, tedy zpracování několika transformací či validací zároveň. Takové možnosti žádný s běžných online nástrojů nenabízí. A toto, a možnost rozšířit aplikaci z hlediska podporovaných formátů a možností zpracování, má být právě další přínos aplikace vyvinuté v rámci této práce.

2.1 Návrh aplikace

Na úvod si v krátkosti shrňme, jaké jsou požadavky na výslednou aplikaci. První požadavkem je, aby se jednalo o online aplikaci. Tedy aplikaci přístupnou na internetové síti a to pokud možno veřejně. Z toho vyplývá další požadavek. A to co nejmenší nároky na klientskou část. Tedy uživatel by měl být schopný aplikaci používat přes běžně dostupné internetové prohlížeče, bez nutnosti instalace dalších prostředků. Co se týká serverové části, i zde je kladen důraz na minimální technická omezení. Aplikace by měla běžet na běžně dostupném operačním systému. Předpokládá se přenositelnost aplikace a tudíž je také potřeba řešit nasazení aplikace na jiném umístění.

Dalším požadavkem na aplikaci je, aby byla navržena jako modulární. Modulární aplikace znamená taková aplikace, která umožňuje rozšíření přidáním takzvaných nových částí. Tyto rozšíření mohou být různá. Dnes je například běžné, že si uživatel do aplikace může nahrát nový vzhled pouhým

nahráním souboru. Běžné internetové prohlížeče zase například umožňují rozšířit svoji funkcionalitu, jako příklad uveďme *RSS* čtečky, podpora čtení *PDF* dokumentů, emailové klienty a spousty dalších. Uvedené příklady ukazují aplikování modulární architektury, která je uživatelsky založená. Tedy umožňuje běžnému uživateli nahráním určitého souboru danou aplikaci rozšířit. Po uživateli se nevyžaduje znalost zdrojového kódu programu ani programátorské vzdělání. Vše se řeší v rámci uživatelského rozhraní. Samotná logika, jak přidání proběhne je interní záležitost a je na tvůrci aplikace, jak toto provede. V rámci této práce se o takovouto modularitu nejedná. Nebylo cílem práce vytvořit takovouto aplikaci, kterou by si běžný uživatel mohl sám rozšiřovat, ale rozšiřování aplikace probíhá pomocí zásahů do kódu. Řečeno jinak, pokud bude chtít uživatel rozšířit existující funkcionalitu, rozšíření provede napsáním kratších částí kódu, specifických pro daný problém. Uživatelské rozhraní, komunikace se serverem, správa uživatelských účtů a tak dále jsou již hotovy. Vytvářet uživatelsky orientovanou modulárnost nebylo zadáním práce.

Ve fázi návrhu bylo také potřeba zohlednit, že se jedná o program vytvořený v rámci závěrečné práce na vysoké škole. To má samozřejmě také vliv na výsledný návrh aplikace. Účelem práce by měl být jistý přínos, inovace. Je tedy nevhodné používat zastaralé nebo neperspektivní technologie. Ze stejného důvodu je také nevhodné použití komerčních technologií a nástrojů. Je tedy lépe používat software s otevřeným zdrojovým kódem.

Další požadavky jsou spíše praktické. Vychází z účelu a požadovaných funkcí aplikace. Tedy měly by být dostupné knihovny pro validaci a transformaci *XML* dokumentů. Z důvodů rozšiřitelnosti aplikace by se mělo jednat o technologii, pro kterou existuje podpora i pro práci s dalšími formáty. Aplikace by také měla umět pokročilé zpracování úloh, tedy mít možnost spouštět úlohy v daný čas, případně opakovaně. Z programátorského hlediska již každý vyšší jazyk obsahuje komponenty nebo programové prostředky potřebné k vytvoření aplikace, která bude toto umět. Je samozřejmě výhodou, pokud existuje pro daný jazyk knihovna, která tyto úlohy zvládá. Není důvod programovat něco, co už někdo jiný udělal.

2.2 Zvolené technologie

V předchozí části byly vyjmenovány určité nároky na aplikaci a použité technologie. Výběr technologie značně ovlivňuje výslednou aplikaci a celý její vývoj. V této části se budeme věnovat technologiím, které byly použity při vývoji aplikace vyvíjené v rámci této práce a zároveň bude i krátce vysvětlen důvod výběru dané technologie.

Jako programovací jazyk, ve kterém se bude psát logika aplikace, byl zvolen jazyk *JavaScript*. *JavaScript* je objektově orientovaný skriptovací jazyk. Jedná se o interpretovaný jazyk. Kód vykonává přímo, není tedy nutné program kompilovat, tedy převádět do kódu strojového. Strojový kód je ovšem obvykle specifický pro daný procesor. Díky tomu je *JavaScript* silně multiplatformní, stačí mít nainstalován interpret pro daný systém a kód je spustitelný bez nutnosti nějakých úprav. *JavaScript* je označení používané firmou *Netscape*, což je původní tvůrce tohoto jazyka. Standardizovaná verze je pojmenovaná jako *ECMAScript*. V současné době se převážně vkládá do *HTML* stránek, kde se používá jako programovací jazyk, který ovládá různé interaktivní prvky jako tlačítka, textová pole. Ale mohou se pomocí něj také tvořit animace, přidávat efekty prvkům, odesílat a přijímat asynchronně data ze serveru nebo kontrolovat vstupy zadané uživatelem. Na rozdíl od skriptovacích jazyků jako *ASP* nebo *PHP*, které běží na serveru a vytvoří obsah stránky ještě před odesláním, se *JavaScript* spouští až po

stažení stránky z internetu. Tento skriptovací programovací jazyk je využit na straně klienta i na straně serveru.

2.2.1 Klientská část

Jako klientská část se v rámci webových aplikací označuje část aplikace, běžící ve většině případů v prohlížeči uživatele. Stránka se obvykle skládá z *HTML* kódu, který určuje obsah dokumentu. Dále z *CSS*, kaskádového stylu, který naopak upravuje vzhled stránky. I v rámci této práce jsou stránky napsané v *HTML* jazyce. Pro úpravu zobrazení je také využito *CSS*. Konkrétně se jedná o využití frameworku *Bootstrap*[5]. *Bootstrap* je volně stažitelná řada nástrojům, které slouží pro úpravu uživatelského rozhraní. Umožňuje definovat základní prvky, které jsou nutné pro zobrazení a odesílání dat uživatelů.

Ke zpracování interaktivních událostí straně klienta se využívá čistý *JavaScript* a částečně framework *JQuery.* Obdobně jako v *CSS* existuje řada frameworků, které usnadňují práci a přidávající řadu funkcí, které by si jinak programátor musel sám a často složitě programovat. Jedná se například o již zmíněný *JQuery* nebo *AngularJS*. V rámci této práce ovšem není potřeba vykonávat nějaké složitější úkony pomocí *JavaScriptu* na straně klienta. V zásadě jediné, k čemu je využit, je jednoduché odesílání a příjem dat, skrývání částí stránky, případně stahování náhledů dokumentů. Frameworky jsou komplexní nástroje s řadou funkcí a jejich nadměrné použití v tomto případě by aplikaci zpomalilo.

Nakonec zbývá dodat, že v rámci klienta se využívá ještě templatovací nástroj *EJS*, *Embedded JavaScript*[6]. Tento nástroj umožňuje do *HTML* stránky vkládat části, které obsahují *JavaScript* kód. Stránka se vyrenderuje, tedy vykreslí, a kód uvnitř těchto značek je proveden a proměnné jsou nahrazeny hodnotami. Takováto stránka je potom odeslána uživateli. V rámci této aplikace je *EJS* využíváno k zobrazení zpráv uživateli.

2.2.2 Serverová část

Aplikace vyvíjená v rámci této práce je webová aplikace. Pro prezentaci stránek, je potřeba mít webový server, tedy server, který zpracovává dotazy a zpět zasílá výsledky. Dotazy obvykle posílá uživatel pomocí webového prohlížeče, který dotaz sestaví a odešle na server přes protokol. *HTTP* je internetový protokol určený pro výměnu dokumentů ve formátu *HTML*. Jedná se o textový protokol, data jsou posílána jako čistý text. Funguje na principu dotaz-odpověď. Uživatel pošle dotaz, server na něj odpoví. Tento protokol je bezestavový, server vyřídí dotaz a tím to pro něj končí. Neukládá si žádná data o klientovi ani stav komunikace. Dotazy spolu nemají souvislost. Což komplikuje složitější úkony, jako například uchovávání informací o klientovi na serveru. Z tohoto důvodu byl tento protokol rozšířen o tzv. *cookies*. Tento prostředek umožňuje serveru si tyto informace uchovávat. Princip je takový, že server vygeneruje *cookies* s informacemi, které potřebuje uložit, přiloží je k *HTTP* dotazu a zašle klientovi. Ten si je uloží a při další komunikaci tohoto serveru tato data pošle opět serveru zpět přiložená do dotazu.

Je tedy potřeba vytvořit webový server pro vyvíjenou aplikaci. V rámci práce byl k dispozici virtuální server umístěný na síti *Technické univerzity v Liberci*, který bylo možné vzdáleně spravovat. Na server byl nainstalován operační systém *Ubuntu*. Jedná se o systém založen na distribuci *Debian*, který využívá *Linuxové* jádro a je šířen pod licencí *GNU GPL*. Webový server je vlastně program, který

naslouchá na daném portu a pokud přijde HTTP dotaz, tak ho vyřídí. Nejrozšířenějším světovým webovým serverem je *Apache HTTP Server*. Dále například *Internet Information Services*. V této práci se jako webový server používá systém *Node.js* a knihovna *Express*[7].

Node.js je vysoce výkonné, událostmi řízené prostředí pro *JavaScript*. Primárním účelem je platforma pro vývoj webových aplikací. Platforma znamená, že zahrnuje i webový server. Není tedy nutné instalovat nezbytně nutné instalovat žádný další webový server. Už v základní konfiguraci obsahuje modul nazvaný *http,* který vytváří webový server a zpracovává *HTTP* požadavky. Základem *Node.js* je interpret jazyka *JavaScript*, který se nazývá *V8* a je vyvíjen firmou *Google*, je obsažen například v prohlížeči *Chrome*. Nad ním je tenká vrstva napsaná v kódu *C++*, která například zajišťuje zpracování příchozích událostí nebo obsluhuje vstupní a výstupní buffery. Proč byl *Node.js* vybrán? Umožňuje používat *JavaScript* na serveru. To je jeden ze základů jeho rychlého rozšíření, protože většina vývojářů klientských webových aplikací se již s *JavaScriptem* setkala. Takže je zde velké množství programátorů, kteří mohou začít programovat serverovou část aplikace bez nutnosti učení se nového jazyka. Také většinu knihoven používaných pro klientský *JavaScript* je možné používat i na serveru. I díky tomu se *Node.js* rychle rozšířil a vzniklo velké množství balíčků. Mimo to je *JavaScript* rozšířen i do oblastí, kde by dříve nikdo nečekal jeho použití. Používá se například pro komunikaci s *NoSQL* databázemi, ukládání data ve formátu *JSON* nebo psaní aplikací pro mobilní platformy. Má podporu u velkých firem, využívají ho například firmy *IBM*, *Microsoft*, *Yahoo*, *LinkedIn*, *Netflix* a spousty dalších.

Nyní ještě krátce k architektuře *Node.js*. Základním rysem, díky čemuž je velice rychlý a škálovatelný je to, že vše běží v jednom vlákně. Tedy na rozdíl například od serveru *Apache*, který pro každý dotaz vytvoří nové vlákno, které požadavek zpracuje. To s sebou nese velkou režii, vytvoření vlákna, načtení konfigurace projektu, vytvoření spojení s databází a tak dále. Což bývá často neefektivní nebo rovnou zbytečné. Aplikace *v Node.js* jsou při spuštění celé načteny a inicializovány, až poté začne aplikace naslouchat příchozí požadavky, které směřuje na konkrétní akce. Dalším rysem je asynchronní zpracování. *Node.js* používá takzvaný událostmi řízený neblokující I/O model. Princip je následující. V běžném jazyce, jako například *PHP,* je obvykle jeden příkaz na řádek, dokud se tento příkaz nevykoná, program čeká a až poté se provede příkaz na dalším řádku. Tedy blokuje ostatní příkazy. I v *Node.js* je možné psát tímto způsobem, ovšem pouze v částech, kde nedochází k požadavkům uživatele. Pokud se jedná o zpracování konkrétního HTTP požadavku, už je nutné používat asynchronní metody a funkce. A to z důvodů, že aplikace pracuje v jednom vlákně, tedy synchronní metody, dokud nejsou zpracovány, blokují celý server. Uveďme si příklad, na kterém toto vysvětlíme.

```
var fs = require('fs');
fs.readFile('soubor.txt', function(err, data){
  console.log('text1');
});
console.log('text2');
```

Příklad 17 - Node.js - asynchronní zpracování

První řádek načte modul *fs*, který slouží k práci se soubory. Na dalším řádku se volá metoda *readFile*, která načítá soubor s názvem *soubor.txt.* Druhým argumentem je funkce, která se zavolá, po ukončení načítání. Jako parametry má *err*, což je chybová hlášení a *data*, což jsou data načtená ze

souboru, pokud se neobjeví chyba v *err*. Příkaz *console.log* vypíše do konzole zadaný text. Jak tedy bude vypadat výpis v konzoli po spuštění tohoto programu? V konzoli bude vypsáno nejdříve „text2“ a až poté „text1“. Totiž ve chvíli, kdy skript narazí na příkaz *readFile,* tak spustí čtení z disku, ale na pozadí. A pokračuje ihned dál, tedy vypíše „text2“. Teprve až dojde k načtení dat z disku, tak dojde k zavolání anonymní funkce, která je druhým parametrem *readFile* a dojde k vypsání „text1“. Tedy v *Node.js* existuje něco jako fronta událostí, které se poté volají v pořadí, v jakém byly do fronty vkládány. Uživatelský kód by tedy měl být napsán jako neblokující. Pokud se vyskytnou operace, které mohou zabrat více času, je potřeba to řešit jiným způsobem. Například použít modul pro spuštění úlohy ve vlastním vlákně, což umí například balíček *child_process*. Na závěr ještě dodejme, že vstupní a výstupní operace, jako práce s databází, čtení a zápis souboru a podobně se spouštějí mimo frontu událostí na pozadí ve vlastních vláknech. Neblokují tedy běh serveru. Další vlastností je modulárnost. *Node.js* je založeno na modulární architektuře. Obdobně jako funguje například správa balíčků *apt-get* v *Ubuntu*, stejně v *Node.js* funguje *npm*. Některé moduly, či jinak řečeno balíčky, jsou již dodávány přímo jako součást platformy, ostatní je možné pomocí *npm* stáhnout. V projektu poté obvykle bývá soubor *package.json*, ve kterém se nachází popis projektu, jméno autora, verze a mimo jiné i soupis všech balíčků použitých v daném projektu s jejich verzemi. Tento soubor vytvoří programátor a stažení balíčků se provádí spuštěním příkazu *npm install* v kořenovém adresáři projektu. Platforma *Node.js* a všechny dále vyjmenované balíčky jsou, pokud nebude uvedeno jinak, volně k dispozici pod licencí *MIT*. Licence stanovuje, že pod touto licencí je možné je použít jak v softwaru s uzavřeným kódem, tak i v kódech s licencí *GPL*, která vyžaduje, aby zdrojové kódy byly zveřejněny.

Důvody k výběru technologie *Node.js* pro serverovou část aplikace jsou tedy vysvětleny. tedy zřejmé. Jedná se perspektivní technologii, která umožňuje využití stejného jazyka na serverové i klientské části, tím usnadňuje vývoj aplikací. Dále je takovýto systém poměrně jednoduše horizontálně škálovatelný a je tedy možné jeho možnosti zpracování úloh dále zrychlit přidáním dalších serverů. Díky velkému rozšíření existuje řada knihoven a díky správci balíčků *npm* není problém najít a použít knihovnu na téměř každý běžný účel. A zároveň je velice výhodné použít stejný jazyk na klientu i serveru. Další výhodou je snadné vytvoření webového serveru. Již byl zmíněn základní modul *http*, v aplikaci je ovšem použit modul *Express*. Jedná se o modul, který přidává vrstvu nad modul *http*. Oproti němu navíc automaticky zpracovává běžné operace, jako parsování *cookies* nebo těla dotazu. Programátor tedy nemusí tyto činnosti zbytečně opakovat při každém zpracování a modul *Express* to udělá za něj.

V následující části budou vyjmenovány a krátce popsány další balíčky, které byly použity při vývoji aplikace. Úvodem ještě dodejme, že kromě balíčků pro *Node.js*, byly ještě použity dvě knihovny, které se neinstalují do *Node.js*, ale přímo do systému. Prvním je *MongoDB*[8]. Jedná se o multiplatformní dokumentovou databázi. Je to takzvaná *NoSQL* databázi. Na rozdíl od *SQL* databází, které pro ukládání dat využívají tabulky, *MongoDB* využívá již zmíněný formát *JSON* a databázové schéma se generuje dynamicky. Existují i nadstavby, které umožňují definovat schéma databáze, například *Mongoose*[9], o kterém se zmíním později. Databáze je v aplikaci potřeba z důvodů ukládání registračních dat uživatelů a také kvůli pokročilým funkcím zpracování úloh, v databázi jsou uloženy data úloh. A druhá knihovna je knihovna *libxml2-dev*. Jedná se o knihovnu pro práci s *XML* dokumenty. Nabízí pokročilé možnosti práce s *XML*, mimo jiné validace a transformace. Některé později zmíněné *Node.js* balíčky ji vyžadují, protože využívají její funkce pro práci s *XML* a tvoří tedy mezivrstvu pro použití jejich funkcí v *Node.js*.

Nyní již k samotným balíčkům použitým v aplikaci. Aplikace obsahuje systém registrace uživatelů. K těmto potřebám aplikace využívá balíček *passport*[10]. Tento balíček umožňuje snadnou implementaci autentifikace do aplikace. Jeho účelem je autentifikovat dotazy, tedy zjistit, zda je dotaz od přihlášeného uživatele nebo ne. Balíček *path*[11] poskytuje prostředky pro práci s cestami k souborům, například vrátit z celé adresy jméno souboru, cestu k souboru a podobně. Obdobně balíček *fs-extra*[12]. Ten poskytuje rozhraní pro přístup k souborovému systému, jako čtení a zápis do souboru, mazání souboru a podobně. Balíček *Mongoose* již byl zmíněn, jedná se nadstavbu nad *MongoDB*, která umožňuje definovat schéma databáze. Díky tomu je možné specifikovat, jaké datové typy mají jednotlivé hodnoty mít, zda je hodnota daného elementu vyžadována nebo definovat povolené hodnoty výčtem prvků. Tím umožňuje udržovat integritu databáze. Balíček *connect- flash*[13] je určen k zasílání textových zpráv uživateli. Zasílá tzv. *flash* data. Jedná se o data, která se uloží do uživatelovi *session*, jakmile se uživateli zobrazí, jsou zahozeny ze *session*. V tomto případě se používají pro informování uživatele, například zda proběhlo správně přihlášení nebo zobrazují výsledky operací se souborem. Balíček *url*[14], jak již název napovídá, je pomocník pro práci s *URL* adresami. Balíček *multer*[15]. Tento balíček je využit pro nahrávání dat od uživatele na server. Je určen pro zpracování dat formulářů, skládajících se z více částí, které právě umí přiložit k dotazu. Tím umožňuje nahrávání i více souborů od klienta na server. Balíček *request*[16] umožňuje jednoduše vytvářet *HTTP* dotazy. V aplikaci je použit pro stažení souboru z externího umístění, v případě že se validace či transformace provádí na souboru, který je umístěn na zadané *URL* adrese. Balíčky *cookie-parser*[17] a *body-parser*[18] jsou pomocné nástroje, které parsují cookies, respektive tělo zprávy, a nalezené hodnoty ukládají do objektů, aby mohly být snadněji zpracovány programátorem. Obdobně balíček *express-session*[19], který slouží k vytvoření *session. Session* je prostředek, který umožňuje uložit data o klientovi na serveru. Klient pošle svůj identifikátor a server podle toho určí, která jsou data klienta. *Session* se v aplikaci používá při ověřování uživatele. Další je balíček *agenda*[20]. Tento balíček je určen k vytváření a plánování úloh na konkrétní čas či pro spouštění úloh v daných intervalech. K ukládání úloh využívá databázi *MongoDB*, kam uloží data úlohy. Tento balíček je podrobněji popsán v části věnující se plánování úloh.

2.3 Struktura aplikace

V následující části se budeme věnovat struktuře programu. V návrhu architektury, potažmo struktury, programů se využívá návrhových vzorů. Návrhový vzor v softwarovém inženýrství představuje obecné řešení problému. Využívá se při návrhu počítačových aplikací. Nejedná se o knihovnu nebo část zdrojového kódu, nýbrž o jakýsi popis řešení problému nebo šablonu, kterou lze opakovaně použít v určitých situacích. Například objektově orientované návrhové vzory typicky ukazují vztahy a interakce mezi třídami a objekty. Ale neurčují implementaci konkrétní třídy.

Pří návrhu aplikace vyvíjené v rámci této práce byl aplikován návrhový vzor *MVC*. *MVC* znamená *Model-View-Controller*. Princip tohoto návrhového vzoru je takový, že rozděluje aplikaci na tři části, které při modifikaci jedné z nich, mají jen minimální vliv na ostatní. Jsou to tyto tři. Část datovou, anglicky *Model*, což je datový model aplikace, reprezentuje informace, se kterými aplikace pracuje. Další částí je část *View*, která reprezentuje uživatelské rozhraní. Převádí data do podoby vhodné k prezentaci uživateli. A třetí částí je řídící logika, *Controller*, jinak řečeno řadič. Ten reaguje na události, například od uživatele, a zajišťuje změny v modelu nebo pohledu. Průběh aplikace je zjednodušeně takovýto. Uživatel vyvolá akci v uživatelském rozhraní, například klikne

na tlačítko. Řadič dostane oznámení o provedení této akce. Řadič vezme model a podle provedené akce, pokud je to potřeba, ho aktualizuje. Provede další potřebné kroky a nakonec je použit aktualizovaný model a vytvořeno nové uživatelské rozhraní s již aktualizovanými informacemi. A opět se čeká na akci, například opět vstup od uživatele.

Zdrojové kódy aplikace se obvykle rozdělují do více souborů, kvůli přehlednosti. Struktura složek a souborů v rámci vyvíjené aplikace je následující.

```
- app
------ models
---------- user.js
------ routes.js
- config
------ database.js
------ passport.js
- modules
------ XML_transformation_XSLT.js
------ XML_validation.js
------ XML_validation_Schema.js
- node_modules
- views
------ browse_file.ejs
------ file_upload.ejs
------ index.ejs
------ login.ejs
------ signup.ejs
------ schedule_files.ejs
------ profile.ejs
- package.json
- Server.js
```

Příklad 18 - ukázka struktury programu

Nyní si postupně probereme význam a obsah složek a souborů vyjmenovaných v *Ukázka struktury programu*. První je složka *app*. Ve složce *app* se nachází složka *models* a soubor *route.js*. Tento soubor obsahuje popis akcí, jedná se tedy o popis zpracování akcí *Controlleru*. Definuje se v něm, jaká činnost se má provést při zavolání dané akce. Ať už se jedná o akci vyvolanou uživatelem, jako kliknutí na odkaz a vrácení nové stránky nebo vrácení dat asynchronnímu volání. Jinak řečeno, pokud webový server obdrží dotaz, předá ho k vyřízení *Controlleru*, který se podívá do definic v souboru *route.js.* Pokud se zde nenachází definice pro danou akci, vrátí chybu. Pokud existuje, tak se provedou definované akce.

Ve složce *models* se nachází definice datových modelů aplikace. Nachází se zde soubor *user.js*. V tomto souboru je definováno schéma databáze. Pomocí již zmiňovaného nástroje *Mongoose* se zde definuje schéma uložení přihlašovacích informací uživatele do databáze *MongoDB*. A také se zde definují dvě funkce pro vytvoření *hash* kódu z uloženého hesla a funkce pro kontrolu, zda se jedná o správné heslo. Toto je poté využito balíčkem *passport*.

Další složkou je složka *config*. Jak již název napovídá, obsahem jsou soubory, které uchovávají konfiguraci. Obsahem složky jsou soubory *database.js* a *passport.js*. Obsah souboru *database.js* je velice krátký, obsahuje pouze *URL* adresu *MongoDB* serveru, ke kterému se aplikace připojuje kvůli registraci a přihlašování uživatelů. Soubor *passport.js* je rozsáhlejší. Definují se zde funkce pro práci s uživatelem. Zejména založení nového a přihlášení stávajícího uživatele.

Ve složce *modules* jsou uloženy soubory s definicemi akcí, které se dají s dokumenty provádět. V současné době jsou zde tři a definují, jak provádět validaci *XML*, validaci pomocí *XML Schema* a *XSLT* transformaci. Do tohoto adresáře by měly být ukládány definice nových akcí s *XML*, případně jinými dokumenty. V sekci věnované možnostem rozšíření aplikace je toto podrobněji vysvětleno.

Adresář *node_modules* je automaticky vytvořený a ukládají se do něj všechny balíčky určené pro *Node.js*, které jsou nutné pro běh programu. Pokud je v programu někde vyžadován nějaký balíček pomocí příkazu *require*, tak se program podívá do toho adresáře a zde hledá daný balíček. Instalace balíčků probíhá pomocí již zmiňovaného *npm*.

V adresáři *users* se nachází složky s daty jednotlivých uživatelů. Při založení nového uživatele se zde automaticky vygeneruje složka, do které se pak danému uživateli ukládají nahrané soubory, výsledky transformací, případně textové soubory s výpisem výsledku naplánovaných a periodicky spouštěných úloh, o kterých bude řeč dále.

A konečně složka *views*. Jak již název napovídá, z hlediska vzoru *MVC* se jedná o *Views.* Jsou zde uloženy definice uživatelského rozhraní, tedy stránek, které jsou poté zobrazovány uživateli. Stránky zde jsou uloženy s koncovkou *.ejs*. Jedná se o klasické stránky napsané v kódu *HTML* s vloženými skripty napsanými v *JavaScriptu*. Tyto skript obsluhují vstupy a výstupy uživatele. Jediným rozdílem oproti klasickému *HTML* kódu je přítomnost speciálních značek *<%*, *<%=* a *%>*. Tyto značky jsou zpracovány při vytváření stránky balíčkem *EJS.* Tento balíček byl již zmíněn dříve v textu. Při vytváření stránky je stránka projita a části kódu označené již zmíněnými značkami jsou předzpracovány a příkazy v nich provedeny, případně dosazeny hodnoty proměnný. V aplikaci se tohoto využívá k zobrazování krátkých zpráv uživateli. Jako například zprávy při přihlašování nebo výsledky operací s dokumenty.

V kořenové složce se ještě nachází následující tři soubory. Nachází se zde soubor *package.json*, *script.sh* a *Server.js*. V souboru *package.json* je uveden popis aplikace a mimo jiné výpis balíčků, které jsou potřebné k běhu aplikace. Vše je zde uloženo ve formátu *JSON*. Důvodem, proč do tohoto souboru zapisovat tyto informace je ten, že nástroj *npm* umí tento soubor automaticky zpracovat a spuštěním příkazu *npm install* provede stažení všech potřebných balíčků, což usnadňuje nasazení aplikace. V souboru *script.sh* je zapsán skript, který je spustitelný v *shellu* a tento skript v krátkém časovém intervalu kontroluje, zda program běží. Pokud ne, tak program spustí. V případě nasazení aplikace na produkční server je dobré spouštět aplikaci zavoláním tohoto skriptu, místo přímého spouštění aplikace voláním níže zmíněného souboru *Server.js*. Aplikaci je tedy vhodné spouštět příkazem *sudo sh script.sh*. Tento příkazem se může lišit dle distribuce operačního systému. A nakonec již zmíněný soubor *Server.js*. Toto je hlavní soubor aplikace. Obsahuje definici potřebných balíčků a příkazy ke spuštění webového serveru. Po spuštění začne aplikace naslouchat na portu a vyřizovat příchozí *HTTP* dotazy. Aplikaci lze spouštět přímo zavoláním příkazu *node* s cestou k tomuto souboru. Například *sudo node Server.js*. Aplikaci není doporučeno spouštět přímo, ale využít výše zmiňovaný skript v soubor *script.sh*.

2.4 Zdrojový kód aplikace

Tato část se věnuje zdrojovému kódu aplikace. Je zde popsány architektura a princip aplikace. A prezentováno, jak a proč byla aplikace napsána a klíčové části kódu jsou názorně vysvětleny. Jak již bylo řečeno je zdrojový kód napsán v jazyce *JavaScript* a spouštěn pomocí aplikace *Node.js*.

2.4.1 Inicializační část programu

Hlavní část programu, která inicializuje a spustí aplikaci je umístěna v souboru *Server.js*. V tomto souboru se nejdříve pomocí příkazu *require* nahrají všechny potřebné balíčky.

```
var express = require('express');
var path = require('path');
var fs = require('fs-extra');
var mongoose = require('mongoose');
var passport = require('passport');
var flash    = require('connect-flash');
var url = require('url');
var Agenda           = require('agenda');
var configDB = require('./config/database.js');

var agenda = new Agenda({db: { address: configDB.url,
collection: 'agendaJobs' }});

var app = express();
mongoose.connect(configDB.url);
require('./config/passport')(passport);

app.set('view engine', 'ejs');

app.use(express.static(path.join(__dirname,
'public')));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(bodyParser());
app.use(session({ secret: 'abcdefgh123456' }));
app.use(passport.initialize());
app.use(passport.session());
app.use(flash()); //

require('./app/routes.js')(app, passport, agenda);
```

Příklad 19 - ukázka obsahu souboru Server.js

Předchozí ukázka obsahuje výňatek z kódu obsaženého v souboru *Server.js.* Nejdříve se pomocí volání funkce *require* inicializují všechny potřebné balíčky. Konkrétně se jedná o balíčky *express*, *path*, *fs-extra*, *mongoose*, *passport*, *flash*, *url* a *agenda*. Účel těchto balíčků byl popsán dříve v textu. Pomocí

příkazu *var configDB = require('./config/database.js')* je do objektu *configDB* uložen obsah souboru *database.js*, jehož obsahem je objekt ve formátu *JSON*, který má jedinou vlastnost *url*, což je adresa, na která je umístěna databáze *MongoDB*. Příkazem *var agenda = new Agenda({db: { address: configDB.url, collection: 'agendaJobs' }})*se vytvoří instance balíčku *agenda*, použitého k pokročilému spouštění úloh. Parametrem je objekt ve formátu *JSON*, kde se specifikují informace k připojení k databázi *MongoDB*. Nástroj *agenda* si do této databáze ukládá informace o naplánovaných úlohách. Obdobně je pomocí příkazu *mongoose.connect(configDB.url)* vytvořeno připojení balíčku *mongoose* k databázi, čehož je pak využito při přihlašování uživatelů. Řádek *require('./config/passport')(passport)* inicializuje balíček *passport*, který zajišťuje správu přihlašování uživatelů. Definuje se v něm jak bude prováděno zakládání nových uživatelů a autentifikace stávajících. Řádek *app.set('view engine', 'ejs')* určuje, že jako nástroj při vytváření výsledných stránek, které budou zobrazeny uživateli, budou zpracovány pomocí nástroje *EJS*, který byl popsán výše. Na dalších řádcích se pomocí funkce *use* instance objektu *express* postupně přidávají pomocné balíčky, které jsou poté využity při zpracování dotazů uživatele. Například řádek *app.use(bodyParser())* způsobí to, že objektu reprezentujícímu dotaz uživatel, pojmenujme ho například *req* přidává vlastnost *req.body*. V tomto objektu jsou uloženy informace zaslané uživatelem pomocí metody *POST*. Balíček *body-parse* dotaz při přijetí předzpracuje a naplní objekt *body*. Posledním řádkem je řádek *require('./app/routes.js')(app, passport, agenda)* se nahraje obsah souboru *route.js*, kterým je definice, jak reagovat na konkrétní dotazy, přijaté od uživatele. Jinak řečeno se zde definuje, kam C*ontroller* předá dotaz ke zpracování.

Samotné spuštění webového serveru, tedy naslouchání a vyřizování dotazů je poté provedeno následujícím příkazem.

```
var server = app.listen(8081, function () {
      ...
})
```

Příklad 20 - kód spouštějící webový server

Předchozí kód vytvoří objekt *server*. První parametr je číslo portu, na kterém má server naslouchat a vyřizovat příchozí dotazy a druhým parametrem je funkce, která se zavolá ve chvíli, kde je vytváření instance webového serveru dokončeno. V aplikace se v této části, po spuštění serveru, provádí kontrola, zda nejsou v databázi uloženy nějaké úlohy, naplánované na daný čas, případně spouštěné periodicky. Pokud takové úlohy existují, jsou pomocí dat uložených v databázi opět naplánovány. V případě, kdy webový server z nějakého důvodu přestane fungovat a je nutné ho spustit znovu, se tímto zaručí, že úlohy budou i po tomto novém spuštění opět správně naplánovány a spuštěny tak, jak je uživatel dříve zadal. Opětovné naplánování úloh zajišťuje volání funkce *schedule_task_from_backup.* Tato funkce je velice podobná dále popsané funkci *schedule_task*, je pouze lehce upravena tak, aby ke spuštění úloh uměla použít data uložená v databázi. Na rozdíl od funkce *schedule_task*, která zpracovává data zadaná přímo uživatelem. Průběh funkce *schedule_task* bude popsán dále v text, v části věnované souboru *route.js*.

Budeme postupovat postupně, další soubor je umístěn ve složce *models*, která je podsložkou složky *app.* Obsahem je definice schématu databáze pomocí nástroje *mongoose*. Definuje se zde, jak bude vypadat záznam v databázi, ve kterém budou uloženy údaje o uživateli.

```
var userSchema = mongoose.Schema({
     local            : {
          username     : String,
          password     : String,
     }
});
```

Příklad 21 - schéma databáze

Předchozí příklad ukazuje, jak takový zápis definice pomocí nástroje *mongoose* vypadá. Uvedený zápis říká, že databáze bude obsahovat objekty, které se budou nazývat *local* a každý objekt *local* bude mít vlastnosti *username* a *password*, které jsou typu *String*, tedy textové řetězce. Mimo definici schématu obsahuje soubor ještě definice dvou funkcí. Jedná se o funkce pro práci s heslem uživatele. Jedna funkce vrací hash kód hesla uživatele a druhá kontroluje, zda jsou zadané kódy stejné a tím ověřuje, zda se uživatel přihlašuje správným heslem.

2.4.2 Zpracování dotazů uživatele

V této části je popsáno zpracování dotazů od uživatele. Definice zpracování se nachází v souboru *route.js*. V tomto souboru jsou zapsány jména akcí, které poté používá *Controller*. Definuje se zde, jak budou jednotlivé akce obslouženy. Nejlépe to bude ukázáno na příkladu.

```
app.get('/profile', isLoggedIn, function(req, res) {
        res.render('profile.ejs', {
            user : req.user
        });
});
```

Příklad 22 - zápis zpracování akce

Předchozí příklad ukazuje zápis definice zpracování dotazu s názvem *„/profile“*. Pokud to vezmeme postupně, tak *app* je instance balíčku *express*, který, jak již bylo řečeno, zajišťuje běh webového serveru. Funkce *get* poté říká, že pokud se definice vztahuje na dotaz, který bude zaslán metodou *GET* protokolu *HTTP.* První argumentem je takzvaná cesta, jinak řečeno identifikátor akce. Následující parametry jsou takzvané *callback* funkce. Jsou volány po přijetí dotazu a předzpracovávají ho. Například funkce *isLoggedIn* vypadá následovně.

```
function isLoggedIn(req, res, next) {

    if (req.isAuthenticated())
        return next();

    res.redirect('/');
}
```

Příklad 23 - funkce isLoggedIn

Tato funkce má tři argumenty. První je objekt, který má význam dotazu, v příkladu je pojmenován *req*. Obsahuje v sobě informace o dotazu, který zaslal uživatel. Druhý parametr je objekt

odpovědi uživateli. Tedy jaká odpověď se má uživateli zaslat. Funkce *req.isAuthenticated()* vrací *true*, pokud je uživatel přihlášen. Pokud je uživatel přihlášen, pomocí *next()* se předají data ke zpracování další *callback* funkci, uvedené v definici *app.get(..),* konkrétně v příkladu *zápis zpracování akce* se jedná o anonymní funkci, která jediné co udělá, tak je že zavolá funkci *render* objektu *res.* Tento příkaz vytvoří stránku, která bude zaslána uživateli, provede takzvaný rendering, tedy vykreslení stránky, v tomto případě spíše vypsání, protože stránka je zobrazena až pomocí prohlížeče uživatele. Protože je stránka zapsaná v kódu *HTML* s prvky *EJS*, tak renderování provede nahrazení speciálních hodnot *EJS* a zaslání stránky uživateli. Obdobně probíhá zpracování dotazů, zaslaných metodou *POST*. K definování této akce slouží funkce *post* objektu instance balíčku *express*. Takže obdobný zápis k *app.get()* by byl *app.post()*. Parametry jsou stejné. V souboru *route.js* je tedy definováno, jak zpracovat dotazy uživatele. Není nutné vypisovat všechny ukázky, jak jsou dotazy zpracovány. Ukažme si příklad funkce, zpracovávající *XML* dokumenty.

```
app.post('/process_file', isLoggedIn, function(req, res) {
      var file = req.body.file;
      var file_trans = req.body.file_trans;
      var volba = req.body.operace;

      if (!file) {
      …
      }

      switch(volba) {
            case 'XML_validace':
                        xml_valid.processFile_XML(file,
                        req.user.local.username, '', function(err,
                        err_message){
                        …
                        });
                  break;
            default:
            …
      }});
```

Příklad 24 - zpracování souboru

Nyní si krátce vysvětleme, co kód v předchozí ukázce provádí. Jedná se o reakci na dotaz *„/process_file“*. Tento dotaz je vyvolán uživatelem ve chvíli, kdy zadá zpracování konkrétního XML dokumentu. Jedná se o dotaz zaslaný metodou *POST.* Nejdříve se na prvních třech řádkách přečtou informace, které zadal uživatel. Proměnná *file* značí jméno souboru, který chceme zpracovávat. Proměnné *file_trans* není využívána vždy, ale pouze pokud je k operaci potřeba druhý soubor, tedy pokud například provádíme transformaci souboru, tak v této proměnné bude název dokumentu, který obsahuje popis, jak transformaci provést. Pomocí podmínky *if (!file)* se ověřuje, zda byl zadán název souboru, který má být zpracován. Pokud ne, je uživateli zasláno chybové hlášení. A poslední částí je část začínající příkazem *switch*. V hodnotě *volba* je uloženo, jaká operace se má s dokumentem provést. Podle toho se vybere, jak se má dál pokračovat. Pokud se jedná, jako v uvedeném příkladu, o validaci *XML*, zavolá se příkaz *xml_valid.processFile_XML.* Tento příkaz je *definován var xml_valid = require('./../modules/XML_validation.js').* Tedy co a jak konkrétně provést s dokumentem je

definováno v tomto konkrétním případě v souboru *XML_validation.js*, který je umístěn v adresáři *modules*. Nyní trochu přeskočíme a zmíním se o adresáři *modules*. Jak již bylo řečeno, aplikace je navržena jako modulární. Je možné jí rozšiřovat. Jednou z možností rozšíření, je přidání nových formátů, se kterými aplikace bude umět pracovat. A právě do adresáře *modules* se ukládají soubory, ve kterých je uveden zdrojový kód popisující jak a čím dokument zpracovat.

```
var fs = require('fs-extra');
var xsd = require('libxmljs');

var processFile_XML = function (file, user, option, callback){
try
{
      var file_path = __dirname + '/../users/' + user +
      '/files/upload/' + file;

      var xmlContent = fs.readFileSync(file_path, 'utf-8');
      var xmlDoc = xsd.parseXml(xmlContent);

      callback(false, null);
}
catch (e)
{
      callback(true, e);
}
}

exports.processFile_XML = processFile_XML;
```

Příklad 25 - validace XML dokumentu

Předchozí příklad názorně ukazuje, jak je zapsána a uložena funkce, zpracovávající *XML* dokument. Nejdříve jsou načteny potřebné balíčky, které jsou využívány při zpracování dokumentu. V tomto případě se jedná o balíček *fs-extra*, který poskytuje funkce k práci se soubory, a balíček *libxmljs*[21], který obsahuje funkce pro práci s *XML*. V parametru funkce se zadává pouze název souboru, nejdříve je tedy sestavena cesta k souboru, která je závislá na jméně uživatele a souboru. Obsah souboru je načten do proměnné *xmlContent* a ten je parsován knihovnou *libxmljs*. V tomto případě se jedná o jednoduchou kontrolu, zda je *XML* správné. Pokud se v dokumentu vyskytne chyba, nastane vyhození výjimky, která je zachycena blokem *try-catch* a předána do *callback* funkce. V opačném případě je také zavolána *callback* funkce, ale s příznakem že vše proběhlo v pořádku.

Popis modulární architektury a postupu rozšiřování funkcionality je popsán v kapitole věnované modulární architektuře. Nyní stačí vědět, že do souboru *route.js* se pomocí *require* vkládají odkazy na soubory, obsahující definice, jak dokumenty zpracovávat, a že k uložení těchto souborů je určen adresář *modules*.

V souboru *route.js* se nachází kromě definic zpracování dotazů i další pomocné funkce. První je *isValidDate*. Jak název napovídá, tato funkce kontroluje, zda objekt zadaný v parametru reprezentuje hodnotu kalendářního data či nikoliv. Je zde kvůli kontrole údajů zadaných uživatelem, aby nemohly

vznikat nesmyslné vstupy funkcí. Další je funkce *getFiles*. Tato funkce vrací výpis obsahu adresáře, zadaného v parametru. Pomocí této funkce se uživateli zobrazuje, jaké soubory má nahrané na serveru. A konečně poslední funkcí je *download_file_wget.* Tato funkce jako parametr přijímá *URL* adresu, na které je umístěn nějaký soubor a ten poté stáhne do složky uživatele. Aplikace umožňuje spouštět naplánované a periodické úlohy na dokumentu, který není lokálně uložen, ale je dostupný online. Pomocí této funkce se stáhne do dočasného uložiště. Poté co jsou operace provedeny, je tento soubor opět smazán.

2.4.3 Konfigurace, zpracování dokumentů a balíčků

V této části je popsáno, kde a jak dochází ke konfiguraci připojení k databázi, kde se nachází funkce, zpracovávající dokumenty a kam se ukládají balíčky, stažené pomocí *npm*. Začněme složkou *config*. V této složce jsou umístěny dva soubory. Jak již název složky napovídá, obsahují určitý způsob konfigurace. První souborem je *database.js*. Obsah tohoto souboru je velice krátký.

```
module.exports = {
      'url': 'mongodb://localhost:27017/XMLcheckDB'
};
```

Příklad 26 - soubor database.js

Jediným účelem souboru tedy je, že vytvoří *JSON* objekt, jehož vlastností je *url*, ve které je uložena adresa a název *MongoDB* serveru. Zkráceně adresa říká, že *MongoDB* je umístěn na stejné adrese, jako běží aplikace, tzv. *localhost*. Dále, že databáze naslouchá na portu 27017 a jméno databáze, ke které se chceme připojit je *XMLcheckDB*.

Druhým souborem je soubor *passport.js*. Tento soubor konfiguruje balíček *passport*, který poté zajišťuje přihlašování a ověřování uživatelů. Nastavuje se zde strategie, jakou se budou uživatelé přihlašovat. Myšleno ve smyslu, zda se budou přihlašovat jménem a heslem nebo zda pro přihlášení využijí například účet na *Google+*. Balíček *passport* umožňuje více druhů přihlášení, kromě již zmíněného přihlášení přes *Google* umožňuje také použít účet na *Facebooku* nebo *Twitteru*. V rámci této aplikace bylo implementováno přihlášení přes uživatelské jméno a heslo. V souboru se nachází dvě funkce pro serializaci a deserializaci. Tyto funkce jsou využity k uložení dat uživatele do *session* a k jejich opětovnému načtení při přijmutí dotazu. Tím se provádí ověření přihlášení uživatele. Dále se zde nachází dvě funkce, které zajišťují založení nového a přihlášení stávajícího uživatele. Přihlášení probíhá tak, že po odeslání dat uživatelem provede balíček *passport* zpracování dotazu, extrahuje uživatelské jméno a heslo. Poté se provede dotaz do databáze, zda se zde nachází uživatel s takovým zadaným přihlašovacím jménem, pokud ne, vrátí se chyba, že nebyl nalezen zadaný uživatel. Pokud takový uživatel existuje, ověří se ještě heslo, pomocí již zmíněné funkce *validPassword*, která porovná hesla a pokud si odpovídají, je uživatel přihlášen. Založení nového uživatele také začíná dotazem do databáze, zda již uživatel se stejným jménem neexistuje. Pokud ne, vytvoří se nový objekt podle definice v *model.js*. Uloží se do něj přihlašovací jméno a zahashované heslo. V dalším kroku se pro uživatele ve složce *users* založí složka pojmenovaná jménem uživatele. V této složce jsou poté uloženy soubory, které uživatel nahrál, stejně jako záznamy o naplánovaných úlohách a výsledky transformací *XML* dokumentů.

Následuje složka *modules*. V této složce jsou uloženy soubory, ve kterých jsou funkce, zpracovávající vlastní *XML* dokumenty. V současnosti se zde nachází tyto soubory. Soubor *XML_validation.js*, který obsahuje funkci validující zadaný dokument, funkce se nazývá *processFile*_XML. Dále soubor *XML_validation_Schema.js,* ve kterém se nachází funkce *processFile_XSD*, která validuje zadaný dokument oproti definici v *XML Schema*. A nakonec soubor *XML_transformation_XSLT.js*. V tomto souboru je funkce *processFile_XSLT,* která provádí transformaci dokumentu pomocí *XSLT* předpisu. Parametr těchto funkcí jsou prakticky stejné, vstupem je jméno souboru, případně souborů, dále identifikátor uživatele a nakonec *callback* funkce, která se vyvolá po konci provádění operací. V této části nebudeme zacházet do podrobností. V kapitole o rozšiřitelnosti aplikace bude struktura a vytváření těchto souborů popsáno.

O složce *node_modules* už byla řeč, pomocí balíčkovacího správce *npm* se zde ukládají jednotlivé balíčky, potřebné k běhu programu. Ve složce *users* jsou ukládány data uživatelů. Při registraci se zde pro uživatele vytvoří složka, ve které se dále vytvoří složka *files* a v ní složky *uploads*, *results* a *logs*. Do složky *uploads* se poté ukládají soubory, které uživatel nahrál pomocí webového rozhraní, do *results* se ukládají výsledky transformací souborů a do *logs* textové soubory s výpisem průběhu naplánovaných a periodických úloh. Velikost v současné době není programově omezena, jediným limitem je tady velikost disku, který má server k dispozici.

2.4.4 Uživatelské rozhraní

A zbývá nám poslední složka, *views*. Zde jsou uloženy všechny pohledy, neboli stránky, které poté tvoří uživatelské rozhraní. Zobrazují informace uživateli a naopak uživatel pomocí těchto stránek posílá data serveru. Nachází se zde soubory *browse_file.ejs*, *file_upload.ejs*, *index.ejs*, *login.ejs*, *profile.ejs*, *schedule_files.ejs* a *signup.ejs*. I když mají koncovku .ejs, jsou tyto stránky jsou napsané v kódu *HTML*. Ovšem před odesláním stránky k zobrazení uživateli, se nahradí příkazy a proměnné ve speciálních značkách hodnotami. O toto se stará *EJS*, které bylo zmíněno dříve.

```
<% if (message.length > 0) { %>
      <div class="alert alert-danger"><%= message %></div>
<% } %>
```

Příklad 27 - značka pro EJS

Uvedený příklad ukazuje použití *EJS* při vykreslování stránek. Využívá tzv. *flash* zpráv, jedná se o informaci přiloženou do *session*, která se smaže ze *session* po zobrazení uživateli. Při zpracování stránky se *EJS* podívá, zda se zde nenachází speciální značky *<%, <%=* a *%>.* Pokud ano, tak vyhodnotí výrazy v nich a nahradí je. V našem konkrétním případě pokud je délka proměnné *message* delší než nula, tedy obsahuje nějaký text, tak vytvoří speciální *div*, do kterého tuto zprávu vypíše a ona se poté zobrazí uživateli.

Jednotlivé stránky mají různý účel. První stránkou, která je zobrazena uživateli je *index.js*. Na této stránce je pouze název aplikace a dvě tlačítka, jedno pro přihlášení, které uživatele přesměruje na stránku *login.ejs*, a druhé, které přesměruje na stránku *signup.ejs*, kde je možné provést registraci nového uživatele. Stránka *profile.ejs* tvoří rozcestník, kde je uživateli nabídnuto přesměrování na stránku *file_upload.ejs*, *browse_file.ejs* nebo *schedule_files.ejs*. Stránka *file_upload.js* definuje rozhraní, pomocí kterého uživatel nahrává soubory na server do složky *uploads*.

Kromě registrace a přihlášení, které nejsou z uživatelského hlediska ničím zajímavé, se zde nachází hlavní rozcestník, ze kterého je možné pokračovat do dalších částí webových stránek. Na následujícím obrázku je vidět, jak vypadá hlavní rozcestník aplikace.

Obrázek 1 - Hlavní stránka

Na předchozím snímku je ukázka hlavní stránky. Uživatel zde může přecházet mezi stránkami *Nahrávání souborů*, *Práce se soubory* a *Plánování úloh*. Zároveň jde zde krátký náhled k jednotlivým stránkám. V části *Nahrávání souborů* uživatel nahrává své datové soubory, se kterými chce poté přímo pracovat. Soubory jsou uloženy na serveru. Dále je zde krátký náhled, kde je vidět, jaké soubory má uživatel již nahrané.

V druhé části nazvané *Práce se soubory* poté uživatel může provádět operace s těmito soubory. V současné době se jedná o validaci a transformaci pomocí *XML Schema*, respektive *XSLT*. Opět je zde na hlavní stránce vidět náhled na uložené soubory, v této části se jedná o výsledky transformací, které jsou také uloženy na serveru, a je možné s nimi dále pracovat. Po vstupu do této části je uživateli umožněno v reálném čase spouštět validace a transformace dokumentů, ať již uložených na serveru v části popsané v předchozím odstavci, tak také soubory dostupné přes URL adresu, tedy uložených na vzdáleném uložišti. Také je zde možné dané úlohy provádět dávkově. Tedy zadat více vstupních souborů, nad kterými bude daná operace provedena hromadně.

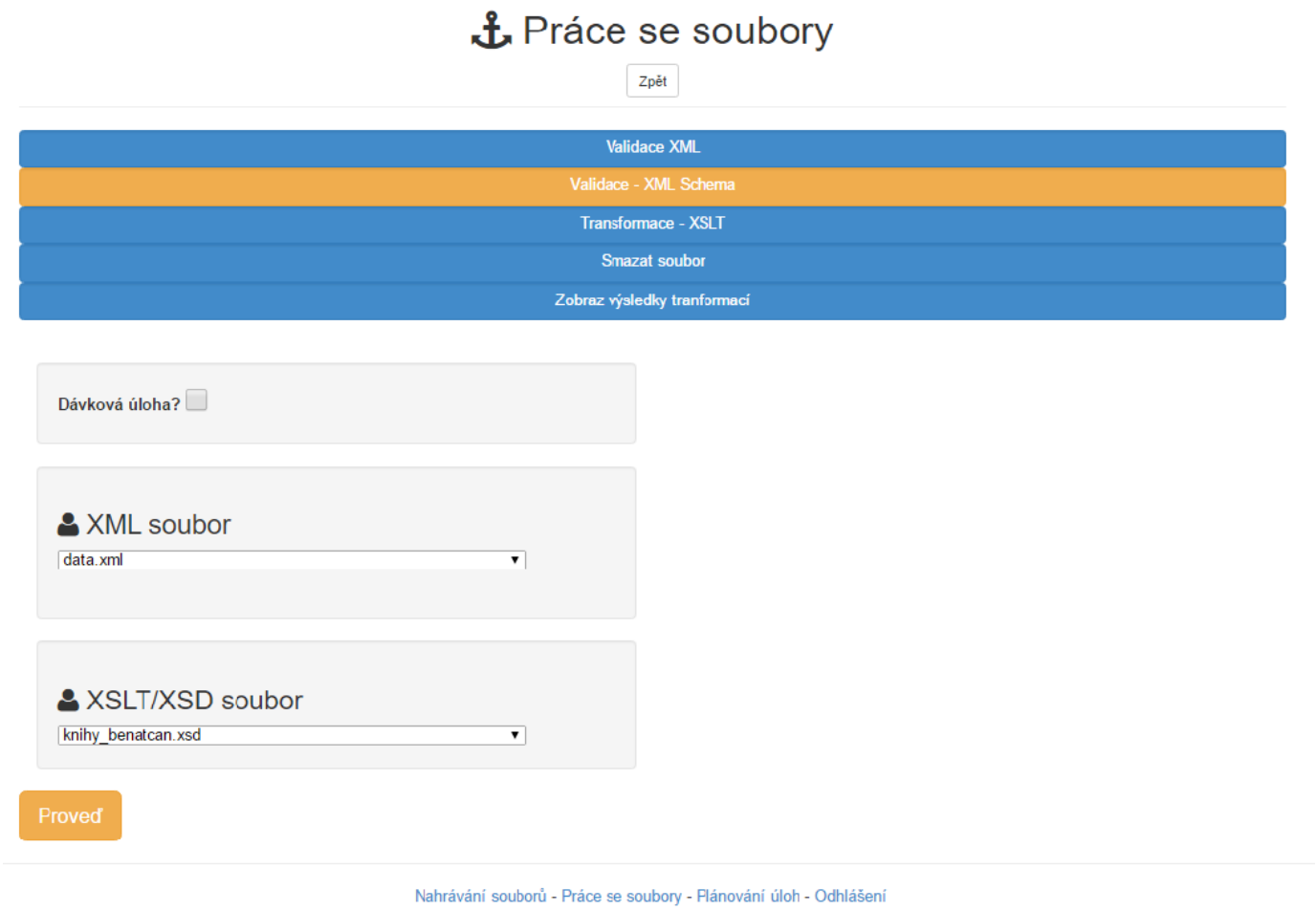
A konečně poslední část, *Plánování úloh*. Tato část je podrobněji shrnuta v následující kapitole. Uživatel zde může plánovat své úlohy, zadávat zpožděné spuštění v daný čas, případně plánovat spuštění v zadaných intervalech. I zde se nachází náhled, v tomto případě je zde výpis naplánovaných úloh.

Na závěr je ještě nutné dodat, že v současné verzi není implementována administrace uživatelů pomocí administrátora, tedy není zde nadřazený uživatel, který by měl pomocí rozhraní přístup k ostatním uživatelským účtům. Tato funkce není v současné době prakticky použitelná v aplikaci, ovšem její přidání je možné. Jak již bylo řečeno, uživatelská data jsou uloženy v databázi a je s nimi tedy možno poměrně jednoduše pracovat.

Obrázek 2 - Ukázka stránky Nahrávání souborů

Stránka fyzicky uložená jako *file_uploads.ejs* je určení k nahrávání souborů, proto i její nadpis je *Nahrávání souborů*. Uživatel na této stránce pomocí jednoduchého dialogu nahrává své soubory na server. Je možné vybrat a nahrát až deset souborů najednou. S nahranými soubory je poté možné provádět akce na stránkách *Práce se soubory* a *Plánování úloh*, které jsou popsány dále.

Stránka *browse_files.ejs* tvoří rozhraní pro práci se soubory umístěnými na serveru. Nadpisem stránky je *Práce se soubory*. Je zde možné vybrat ze seznamu konkrétní nahraný soubor a provádět s ním operace. V současné době obsahuje operace validace, validace pomocí *XML Schema* a transformace pomocí *XSLT*. Uživatel jednoduše vybere, jakou akci chce provést z výběru v horní části stránky, poté zadá vstupní parametry, zda se jedná o dávkovou úlohu, tedy zda vstupem bude více souborů, jaký soubor, případně soubory, budou vstupem a úlohu stiskem tlačítka spustí. Výsledek vidí během chvíle. V případě transformace se výsledek uloží na server a automaticky se nabídne jeho stažení. Výsledky těchto transformací je zde možné prohlížet, ukládají se do složky *results*. A také je zde možné mazat uložené soubory. Náhled stránky je níže.



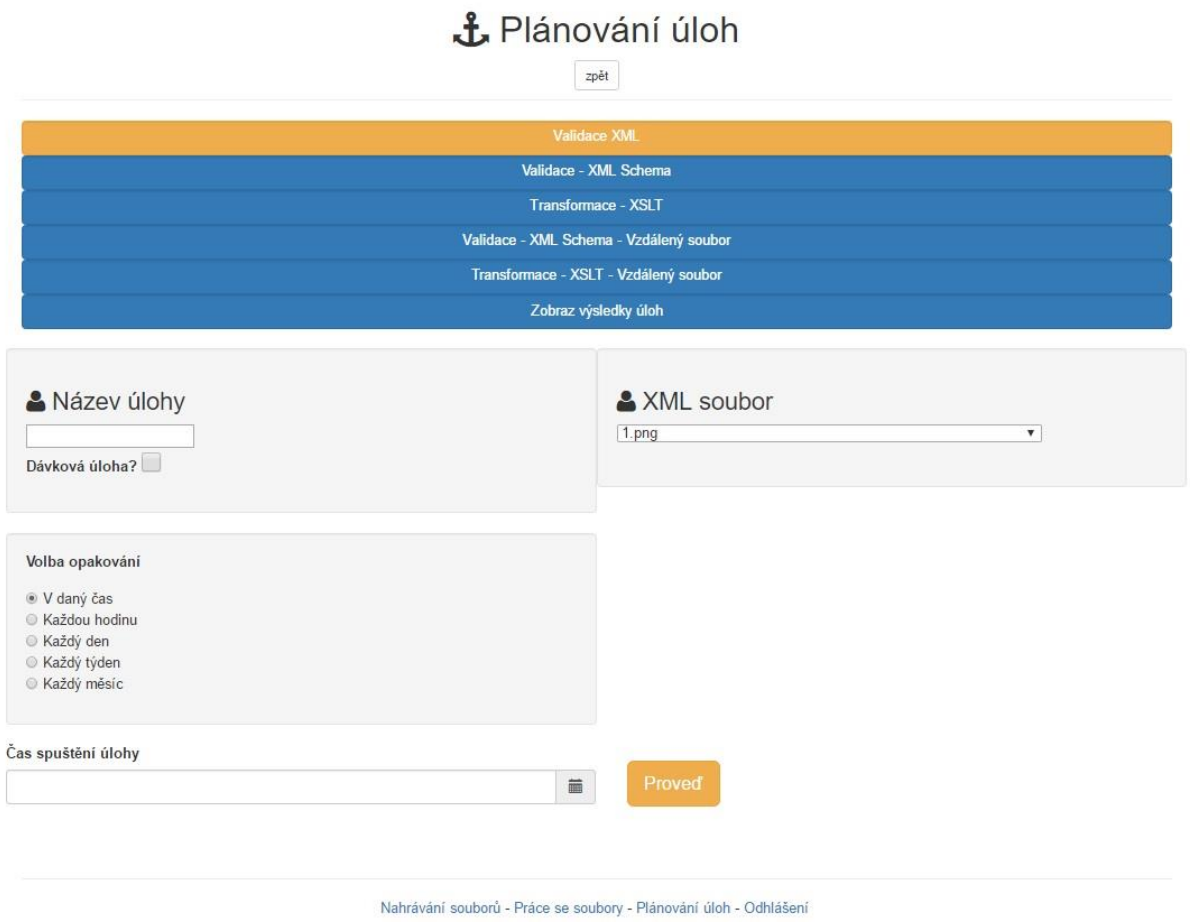
Obrázek 3 - Náhled stránky Práce se soubory

Poslední je stránka *schedule_files.ejs*, nazvaná *Plánování úloh*. Tato stránka tvoří rozhraní pro pokročilou práci s úlohami. Je zde možné naplánovat spuštění validace nebo transformace v přesně určený čas a spouštění úloh v pravidelných intervalech. V současné době je možné naplánovat úlohy na spouštění každou hodinu, den, týden nebo měsíc. V kapitole o rozšiřitelnosti je popsáno přidání jiného intervalu spouštění, stejně jako popsána stránka samotná.

2.5 Pokročilé možnosti spouštění úloh

Jedním ze zmiňovaných přínosů aplikace je možnost spouštět úlohy pokročilým způsobem. Za normální označme spuštění úloh takovým způsobem, že uživatel řekne, jaký dokument chce zpracovat, pokud je to potřeba, zadá i schéma pro validaci případně transformaci, a po odeslání těchto informací, jsou tyto akce hned vykonány a výsledek vrácen uživateli. Naopak pokročilým zpracováním se v této práci myslí spouštění úloh v daný čas, spouštění úloh v pravidelných intervalech nebo spouštění úloh v dávkách.

Aplikace má kromě stránky s přihlášením, založením uživatele a rozcestníku tři hlavní části Jendou z nich je stránka s uživatelským rozhraním pro správu naplánovaných úloh. Uživateli je zde umožněno pomocí jednoduché webové stránky zadávat a sledovat své naplánované úlohy.



Obrázek 4 - Snímek stránky Plánování úloh

Nyní si postupně rozeberme stránku, která je zobrazena na *Obrázku 4.* Tato stránka je funkčně rozdělena na dvě části. V levé části se nachází část, kde se zadávají nové úlohy. Jsou zde různá nastavení, kterými uživatel může nastavit parametry spouštění úloh. V pravé části se nachází panel, s výběrem jednotlivých vstupních souborů.

Jakou konkrétní operaci, chce uživatel provést, se volí v horní části stránky. Dle operace se mohou mírně měnit parametry zadávané níže. V současné době aplikace uživateli nabízí možnost dokument validovat, validovat pomocí *XML Schema* a transformovat pomocí *XSLT*. Validace pomocí *XML Schema* a transformace *XSLT* je možné vykonávat i na dokumentech dostupných přes *URL*. Jak již bylo řečeno, je možnost zadat více lokálních souborů i více souborů dostupných z URL, se kterými se provede daná operace. Ale je možné zadat pouze jeden předpis, podle kterého se bude validovat, respektive transformovat. Výčet operací může být rozšířen, aplikace je navrhována jako modulární. V kapitole věnované modulárnosti je přidání nových operací popsáno.

Dále začněme levou částí. Nahoře se nachází pole, pro zadání jména úlohy. Jméno úlohy je povinné a musí být unikátní, pokud není, teda již existuje úloha se stejným jménem, stránka navrátí chybu a je nutné zadat jiný název úlohy. Ihned pod polem pro zadání názvu úlohy je zaškrtávací políčko,

kterým uživatel nastavuje, zda se jedná o dávkovou úlohu. Pokud se o dávkovou úlohu nejedná, je uživateli umožněno vložit pouze jeden vstupní soubor, se kterým budou prováděny operace. A posledním nastavením, které je nutné zadat je časový údaj, kdy a jak budou operace provedeny. V principu jsou dvě možnosti. Uživatel může operace naplánovat na daný čas. Pro tento případ je zde k dispozici komponenta pro zadání přesného data i s časem s přesností na minuty. Poté se jednou v daný čas operace provedou a výsledek bude uložen do souboru. Druhou možností je provádět operace periodicky. V tomto případě uživatel zadává čas, kdy mají být operace poprvé provedeny a periodu. Tedy za jakou dobu se má operace znovu provést. V současné době aplikace umožňuje nastavit opakování každou hodinu, den, týden či měsíc. Opět v kapitole o modulárnosti je ukázáno, jak přidat další periody spouštění. Poslední věc, která se na pravé straně stránky nachází, je tlačítko *Proveď*. Stisknutím se provede přidání úlohy do prováděných úloh a úloha začne vykonávat dle zadaných parametrů.

Na pravé straně se nachází rozbalovací seznam obsahující výpis všech nahraných souborů. Uživatel zde zvolí, s jakým dokumentem chce následující operace provést. Pokud je vybrána volba dávkových úloh, je zde také umístěno tlačítko *Přidat ke zpracování*. Kliknutím na toto tlačítko se daný soubor přidá do seznamu souborů, se kterými poté bude provedena dále zvolená operace. Aplikace také nabízí možnost provést operace na souboru umístěném na vzdáleném uložišti. Pokud je zvolena tato možnost, pak se zde místo rozbalovacího seznamu nachází přímo textové pole, kam uživatel zapíše celou *URL* adresu, kde se nachází soubor, který chce zpracovat. Opět je zde možnost zadat více umístění, pokud je zvolena dávková úloha. Dále pokud se jedná o operaci, která vyžaduje jako vstup druhý soubor, tedy například validace pomocí *XML Schema* nebo transformace *XSLT*, je zde další rozbalovací seznam, který obsahuje stejné soubory, jako předešlý seznam. Zde uživatel zvolí, jakým schématem chce výše zvolený dokument validovat, respektive transformovat.

Nyní se konečně dostáváme k *Zobrazení výsledků úloh*. Jedná se o jednu volbu zpracování. Uživatel tímto tlačítkem může zobrazit okno, které obsahuje rozbalovací seznam, náhled a dvě tlačítka *Stáhnout* a *Smazat úlohu*. Toto okno obsahuje výpis všech operací daného uživatele. Jsou zde uloženy výpisy z běžících periodických úloh, stejně jako úlohy naplánované na určitý čas, pokud již proběhly i pokud ne. Výpisy z průběhu jsou uloženy v adresáři *logs* ve složce uživatele. Jedná se o textové soubory, do kterých se zapisují výstupy z operací spolu s detaily úlohy. Přepnutím na jinou položku v rozbalovacím seznamu se zašle dotaz na server a ten pošle zpět výpis souboru, který je uživateli zobrazen v textovém poli napravo.

Pokud uživatel chce, může si uložený výsledek zadané úlohy stáhnout a procházet na svém zařízení. Slouží k tomu tlačítko *Stáhnout*, po kliknutí je uživateli nabídnuto stažení. Druhé tlačítko *Smazat úlohu*, jak již název napovídá, slouží k vymazání a přerušení běhu vybrané úlohy. Při tomto kroku jsou vymazána všechna data o úloze, tedy i uložený výpis průběhu. Pokud chce uživatel soubory zachovat, musí je stáhnout na své zařízení.

2.6 Modulární struktura aplikace

V následující kapitole je popsána modulární architektura vyvíjené aplikace. V předchozích kapitolách bylo řečeno, že aplikace pro zpracování *XML* dokumentů již existují, byly proto předneseny jisté požadavky na aplikaci. Například správa uživatelů, která byla popsána výše. Nebo možnosti pokročilého spouštěné úloh, popsané v předchozí kapitole. Z hlediska budoucího použití je ovšem také důležité,

aby aplikace stále splňovala požadavky uživatelů. Tyto požadavky se pochopitelně v čase mění. Ať již se jedná o nové formáty, které je potřeba umět zpracovat, či o jiné možnosti průběhu operací s dokumenty. Například jiná perioda spuštění, jiná reakce na výskyt chyby nebo jen jiný druh výstupu. Tato práce se zabývá zejména formátem *XML*, ale samozřejmě existuje více formátů pro popis dat. Zmiňme například formát *JSON,* který je hojně využíván například pro serializaci dat. A velice pravděpodobně budou dál vycházet nové formáty.

Z různých důvodů, ať již vyjmenovaných výše nebo dalších, je zřejmé, že je velice výhodné navrhnout aplikaci tak, aby se dala snadno upravovat. Může se jednat o pohled z hlediska architektury. Snaha oddělit složky tak, aby zásah do jedné neznamenal nutně úpravy ve vrstvě druhé. Například architektura *MVC* odděluje logickou a zobrazovací část. V této práci je pojem modularita chápán z pohledu rozšíření funkcionality dané aplikace. Dalším pohledem je, jak toto rozšiřování provádět. Buď se jedná o rozšíření funkcionality takové, které zvládne běžný uživatel. Například nahrání nového vzhledu do aplikace nebo určitého rozšíření do webového prohlížeče, který například umožní zobrazování dokumentů *PDF*, blokování vyskakovacích oken a podobně. Takovéto řešení rozšiřování aplikace pomocí nahrání určitých souborů. Takovéto řešení je uživatelsky zaměřeno. Vše zvládne běžný uživatel bez odborných znalostí. Toto řešení ovšem klade větší nároky na návrh a vývoj aplikace.

Aplikace vyvinutá v rámci této práce se zaměřuje na jinou rozšiřitelnost. Cílem není vytvořit pokročilé uživatelské rozhraní, přes které by se rozšiřování a úpravy aplikace prováděli. Je zaměřen na pokročilejší uživatele, kteří mají znalosti v oblasti programování a informačních technologií. Systém je navržen tak, aby si uživatel funkčnost rozšířil dopsáním krátkých částí kódu. Aplikace sama o sobě zajišťuje správu uživatelů, umožňuje jim se jednoduše registrovat a nahrávat své soubory. Dále zajišťuje komunikaci se serverem, rozhraní pro práci s nahranými soubory a grafické rozhraní pro spouštění úloh. Rozšiřitelnost aplikace se zaměřuje na rozšíření z pohledu podporovaných formátů, případně možností zpracování úloh. V následující části bude pomocí příkladů popsáno, jak je možné toto do aplikace přidat.

2.6.1 Rozšíření podporovaných formátů

V této kapitole bude popsáno přidání nových formátů do aplikace. Zároveň na tom budou vysvětleny části kódu a popsán princip fungování aplikace. Pro přidání nových funkcí je potřeba učinit dva kroky. Přidat nová ovládací prvky do uživatelského rozhraní. Druhým krokem je napsání části kódu, který bude novou funkcionalitu vykonávat a zařadit ho do funkce zpracovávající úlohy od uživatele.

Jako vzorový příklad si vezměme přidání validačního formátu *Relax NG*, čímž umožníme validaci XML dokumentů tímto formátem. Prvním krokem by mělo být nalezení odpovídající knihovny, kterou chceme použít k validaci. Na internetu je veliké množství volně dostupných knihoven, respektive balíčků, které lze v aplikacích volně použít. Velice oblíbený je server https://github.com/, který online udržuje repozitáře různých aplikací a funguje obdobně jako verzovací nástroje typu *Subversion*. Udržuje informace o změnách a vydaných verzích. Pro naše účely byl použit nástroj *xmllint*[22], který je spouštěn z příkazové řádky *shell*. Jedná se nástroj, jehož popis se nachází na adrese *http://xmlsoft.org/xmllint.html*. *Node.js* umožňuje spouštět nástroje příkazové řádky. Slouží k tomu již zmíněný balíček *child_proces*, syntaxe vypadá následovně.

```
var execCmd = require('child_process').exec;
execCmd (cmd, function(error, stdout, stderr) {
      // standartní výstup je v stdout
});
```

Příklad 28 - volání funkce příkazové řádky z Node.js

Bylo tedy rozhodnuto použít tuto knihovnu. Dle verze a distribuce operačního systému *Linux*, ve kterém aplikace běží, se může lišit postup instalace knihovny. V našem případě použijeme příkaz *sudo apt-get install libxml2*. Tímto příkazem bude knihovna *libxml2* nainstalovaná a my můžeme z příkazové řádky volat její funkce. Takto volaná funkce běží se stejnými právy jako aplikace, která ji spustila.

Způsob, jakým bude provedena samotná validace dokumentu, je tedy připraven. Nyní již zbývá pouze upravit samotnou aplikaci pro použití této funkce. Začněme úpravou grafického rozhraní. Budeme předpokládat, že chceme přidat možnost validace pomocí *Relax NG* jak do plánovaných úloh, tak také do úloh, které uživatel zadá k okamžitému provedení na stránce *Práce se soubory*. Musíme tedy upravit dvě uživatelská rozhraní. Jedná se o soubory *browse_file.ejs* a *schedule_files.ejs*. Začněme souborem browse_file.ejs. Zde musíme přidat možnost validace do výběru pomocí přepínačů. Tyto přepínač jsou zapsány v kódu na řádku přibližně 450. Přidáme do značky *div*, obsahující tyto volby, následující řádek.

```
<div class="controls form-inline" data-toggle="buttons">
      <label id="idLabel1" class="form-control btn btn-warning">
            <input class="form-control" type="radio" name="operace"
            value="XML_validace" checked
            onchange='setVisibility(this)'> Validace XML</label>
      …
      <label id="idLabel5" class="form-control btn btn-primary">
            <input class="form-control" type="radio" name="operace"
            value="RelaxG_validace"
            onchange='setVisibility(this)'>Validace - Relax
            NG</label>
      <label id="idLabel6" class="form-control btn btn-primary">
            <input class="form-control" type="radio" name="operace"
            value="delete" onchange='setVisibility(this)'> Smazat
            soubor</label>
</div>
```

Příklad 29 - přidání možnosti validace pomocí Relax NG

Přidáním zvýrazněného řádku jsme umožnili uživateli vybrat tuto možnost. Z úpravy tohoto souboru je to vše, po stisknutí tlačítka odeslání formuláře se informace zašlou na server. V hodnotě proměnné operace, která bude odeslána na server, se nyní může objevit nová hodnota *RelaxG_validace*, musíme tuto možnost zařadit do zpracování na serveru, to bude vysvětleno dále. Obdobně upravíme i soubor *schedule_files.ejs*. Zde se na řádku přibližně 550 nachází obdobný *div*, jako

v předchozím případě. Opět přidáme stejný řádek, jako do předchozího souboru a je to vše, co je nutné udělat v grafické části.

Nyní přejdeme k úpravě kódu, který je vykonávaný na serverové části. Nejdříve ze všeho vytvoří ve složce *modules* nový soubor, pojmenujme ho například *XML_validation_RelaxNG.js*. Obsahem tohoto souboru bude definice funkce, která bude mít parametry název dokumentu, název schématu, jméno uživatele a funkci, která se zavolá po provedení. Obsah souboru bude vypadat následovně.

```
var exec = require('child_process').exec;

var processFile_RNG  = function (file, file_trans, user, callback){
      var file_path = __dirname + '/../users/' + user +
      '/files/upload/' + file;
      var file_schema_path = __dirname + '/../users/' + user +
      '/files/upload/' + file_trans;

      try {
            var cmd = "xmllint --relaxng " + file_schema_path + " " +
            file_path;
            exec(cmd, function(error, stdout, stderr){
                  if (error){
                        callback(false, error);
                  }
                  else if (stderr){

                        callback(false, stderr);
                  }
                  else{
                        callback(true, stderr);
                  }
            });
      }
      catch (e){
            callback(false, e);
      }
}

exports.processFile_RNG  = processFile_RNG;
```

Příklad 30 - funkce validující pomocí Relax NG

Průběh je následující. Nejdříve vložíme referenci na balíček *child_process*. Tento balíček obsahuje funkce, pro volání příkazů příkazové řádky a tím spouštět binární soubory. Dále vytvoříme

funkci, pojmenovanou *processFile_RNG*. V této funkci se nejdříve z názvů souborů složí cesta k nim. Poté složíme řetězec, který bude reprezentovat příkaz včetně parametrů, který se pak zavolá v příkazové řádce. Vstupem do funkce *exec* je kromě řetězce s příkazem, je ještě callback funkce. Její výstupy zpracujeme a dle nich vrátíme informaci, zda je soubor validní.

Nyní musíme upravit soubor *route.js*. Nachází se v něm zpracování dotazů uživatele. Nejdříve musíme přidat referenci na námi vytvořenou funkci *processFile_RNG.* To se provede přidáním následující řádky na začátek souboru.

```
var xml_transform_RelaxNG =
require('./../modules/XML_transformation_XSLT.js');
```

Příklad 31 - přidání reference do route.js

Tímto příkazem se vytvoří objekt *xml_transform_RelaxNG*, který obsahuje funkci *processFile_RNG*, která má výše definované parametry. Nyní nalezneme řádek *app.post('/process_file', isLoggedIn, function(req, res)*. Zde začíná část kódu, vykonávající zpracování požadavku na určitou operaci s dokumentem od uživatele. Do přepínače *switch(volba)* přidáme nový *case*, který bude zpracovávat naší novou operaci, kterou jsme v grafickém rozraní pojmenovali *RelaxG_validace.* Přidáme následující řádky.

```
case 'RelaxNG_validace':
     xml_transform_RelaxNG.processFile_RNG(file, file_trans,
     req.user.local.username, function(err, err_message){
               if (err)
               {
                    res.render('browse_file.ejs', {
                         user : req.user,
                         message : 'Chyba při zpracování
                         souboru! ' + err_message
                         });
               }
               else
               {
                    res.render('browse_file.ejs', {
                         user : req.user,
                         message : "Validace souboru OK!"
                         });
                    }
               });
```

Příklad 32 - přidání zpracování požadavku na validaci Relax NG

Krátce popišme, co se v uvedené části děje. Přidali jsme možnost přepínače *RelaxNG_validace*. Pokud je proměnná *volba* rovna této hodnotě, vykoná se následující kód. Zavolá se funkce *processFile_RNG* objektu *xml_transform_RelaxNG*, který jsme si definovali pomocí *require*. Funkci se předají jako parametr hodnoty získané od uživatele. V hodnotě *file* je uložen název dokumentu, který chceme validovat. Jedná se o soubor umístěný ve složce *files* v adresáři uživatele. V hodnotě *file_trans* je uloženo jméno souboru, ve kterém se nachází schéma. Opět se jedná o soubor ve složce *files*. Hodnota *req.user.local.username*, je jméno uživatele, které je získané z databáze pomocí *ID* uživatele, které se předává v *session*. Obsahuje údaje o uživateli získané z databáze. Proto je zde *local*, vychází to ze schématu databáze v souboru *database.js*. Posledním parametrem je anonymní funkce, která se vykoná po konci průběhu funkce *processFile_RNG*. Funkce zjistí, zda se vyskytla při průběhu *processFile_RNG*, nějaká chyba a dle toho vrátí výsledek uživateli.

Nyní zbývá dopsat kód pouze do části zpracovávající naplánované úlohy. Pro tento úkol je v *route.js* vytvořena funkce *schedule_task*. V ní je potřeba obdobně přidat zpracování volby *RelaxNG_validace*. Je zde přepínač *switch (operace)*. Přidáme tyto řádky.

```
case "RelaxNG_validace":
{
      fs.writeFile(log_path, "Validace dle schéma. Název úlohy: " +
      task_name + ". Zadána dne: " + datestring + ". Spouštěna: " +
      period_string + ". Soubor xml: " + xml_file + ". Soubor XSD: " +
      xml_file2 + "\n");

      agenda.define(task_name, function(job, done){
            var datum_format = new Date();

            var datestring =  datum_format.getDate()  + "." +
            (datum_format.getMonth()+1) + "-" +
            datum_format.getFullYear() + " " + datum_format.getHours() +
            ":" + (datum_format.getMinutes() < 10 ? "0" +
            datum_format.getMinutes() : datum_format.getMinutes() );

            xml_valid_Schema.processFile_XSD(xml_file, xml_file2,
            user.local.username, '',  function(err){
                  if (err){
                        var string_message = '' + datestring  + " - Chyba
                        validace souboru : " + err + "\n";

                        fs.appendFile(log_path, string_message,
                              function (err2){
                                    done();
                              });
                  }
                  else{
                        var string_message = '' + datestring  + ": " +
                        task_name + " - Validace souboru: OK. \n";
```

```
                    fs.appendFile(log_path, string_message,
                          function (err2) {
                                done();

                          });
                }
                });
          });

          if (period == "just_in_time"){
                agenda.schedule(date, task_name,
                JSON.stringify(data));
          }
          else{
                agenda.every(schedule_string, task_name,
                JSON.stringify(data));
          }

          res.render('schedule_files.ejs', { message: "Přidání
          úlohy proběhlo úspěšně!", user : user });

          break;
    }
```

Příklad 33 - přidání zpracování Relax NG do pokročilých úloh

Na ukázce je vidět naplánování požadavku uživatele na validaci pomocí *Relax NG*. Příkaz *fs.writeFile* zapíše do souboru, zadaného jako první parametr, úvodní informace o úloze. Jak se úloha jmenuje, kdy byla zadána, jaký je časový průběh. Pro spouštění úloh v daný čas či periodicky se využívá balíček *agenda*. Tento balíček umožňuje naplánovat úlohy na určitý čas, či s určitou periodou opakování. Funkce *define* vytvoří novou úlohu se jménem zadaným jako první parametr. Druhým parametrem je funkce, která se má vykonat. Ve funkci si nejdříve vytvoříme z časového údaje čitelný textový řetězec, který potom vepíšeme jako časový údaj do zápisu o průběhu úlohy. Poté zavoláme validaci dokumentu, obdobně jako v předchozím případě, kde jsme přidávali zpracování validace pomocí *Relax NG* do úloh přímo zadaných uživatelem. Poté už zbývá jenom říct, kdy a jak úlohu spouštět. Pokud se jedná o úlohu naplánovanou na daný čas, zavolá se funkce *agenda.schedule*, která má první parametr typu *Date*, což je objekt datumu v jazyce *JavaScript*. Druhý parametr je jméno úlohy, kterou jsme se o pár řádků výš definovali. Poslední parametr je volitelný a jsou to data, která budou uložena spolu s daty dané úlohy do databáze. Ukládají se zde kompletní data, která uživatel zadal. Tímto příkazem se také provádění úlohy, respektive se čeká, až nastane čas, kdy má být spuštěna. V čas jaký zadal uživatel. Pokud se jedná o úlohu spouštěnou periodicky, volá se funkce *agenda.every()*. První parametrem je řetězec, který reprezentuje, jaké opakování uživatel zadal. Bude popsán v následující části, kde bude vysvětleno přidání nových možností periodického spouštění úloh. Zbytek parametrů je stejný, jako v případě *agenda.schedule()*. Po zadání úlohy následuje už pouze vrácení výsledku uživateli, že úloha byla úspěšně přidána. Poslední věc, kterou je nutné udělat, aby vše probíhalo korektně, je přidat definic zpracování do funkce *schedule_task_from_backup* v souboru *Server.js*. tato funkce se stará o obnovení naplánované úlohy v případě pádu aplikace. Do stejné části kódu jako v případě funkce *schedule_task* přidáme stejný kód, jako jsme přidali do *schedule_task*.

Pouze s tím rozdílem, že vynecháme první řádek *fs.writeFile* a řádek *res.render*. První příkaz by nám přemazal soubor s výsledky dosavadního průběhu dané úlohy, protože tento soubor byl již dříve založen. A *res.render* vynecháme z logických důvodů, jedná se o obnovení úlohy, ne o reakci na dotaz uživatele, tedy není proč ani komu zasílat zpět odpověď.

Tímto tedy bylo ukázáno, jak přidat nový formát. Navíc na uvedeném příkladu je ukázáno, že i pokud není k dispozici vhodný balíček pro *Node.js*, pořád je možné využít knihovny a funkce příkazové řádky. Tedy neměl by být problém provádět všechny operace, pro které existuje knihovna pro operační systém *Linux*.

2.6.2 Rozšíření možností spouštění naplánovaných úloh

V předchozí části bylo ukázáno, jak rozšířit funkcionalitu přidáním podporovaných formátů. Jedním z bodů práce je i již zmíněné pokročilé zpracovávání úloh, tedy plánování a opakované spouštění úloh. I tato část je navržena jako modulární s možností přidání nových nebo úpravy současných voleb spouštění. V popisu pokročilých možností spouštění bylo řečeno, že aplikace v současné době podporuje spouštění úloh v daný čas, každou hodinu, den, týden a měsíc. V této části bude ukázáno, jak přidat například spouštění jednou za dva dny, čtyři hodiny a šest minut.

Nejdříve ze všeho se podívejme, jak je vůbec realizováno spouštění úloh dle zadaných parametrů. Aplikace k tomu využívá balíček *agenda*. Tento balíček umí definovat úlohu a poté ji spustit v daný čas, případně spouštět v pravidelných intervalech. Jak se definuje funkce, bylo ukázáno v předchozí kapitole. Zkráceně pomocí funkce balíčku *define* balíčku *agenda* vytvoříme úlohu s daným jménem. Spouštění poté probíhá přes funkce *every* nebo *schedule*. Funkce *schedule* nastaví spuštění úlohy v daný čas. Jako parametry přijímá objekt typu *Date*. Což je *JavaScript* objekt pro práci s datumy. Umí vyjádřit přesný čas. Stačí tedy předat tento objekt s daným časem a úloha bude v tento čas vykonaná. Druhý parametr je název úlohy, který jsme si sami definovali, a kterou chceme v daný čas vykonat. A konečně třetí parametr je volitelný. Zadávají se zde data, které jsou poté uloženy společně s úlohou do databáze. V rámci této aplikace je této vlastnosti využito k ukládání všech dat, které uživatel zaslal při zadávání úlohy. Tyto data jsou pak použita v pro obnovení úlohy v případě pádu aplikace, případně serveru. Druhá je funkce *every*, která slouží k pravidelnému spouštění úloh. Prvním argumentem již není objekt *Date*. Nýbrž textový řetězec daného tvaru, který popisuje spouštění úlohy. Tento řetězec může mít formát, používaný v nástroji *Cron.* Tento nástroj běží jako služba na pozadí a automaticky v určitý čas spouští příkazy či skripty. Krom toho umožňuje i periodické spouštění úloh. Zápis se provádí pomocí pěti polí. Které mají postupně význam minut, hodin, dní v měsíci, měsíce a dnu v týdnu. Každé pole může obsahovat hvězdičku, poté se na něj nebere ohled, provádí se vždy. Jinak může obsahovat konkrétní číslo, více čísel oddělených čárkou. Složitější implementace používají například lomítko, pro vyjádření násobků. Například výraz, kterým bychom chtěli spouštět úlohu každých pět minut, by vypadal takto *„*/5 * * * *“.* Tato syntaxe je poměrně složitá, hlavně v případě, kdy chceme nastavit složitější periodickou úlohu. Proto *agenda* nabízí ještě druhý způsob zápisu, takzvaný pro lidi čitelný zápis. Tento zápis umožňuje přehledněji zapsat definici pakování úlohy. Například pokud chceme spouštět úlohu každých 15 min, řetězec bude vypadat „15 minutes“. V našem případě, pokud chceme opakovat úlohu jednou za dva dny, čtyři hodiny a šest minut bude zápis vypadat takto *„2 days, 4 hours and 6 minutes*“. Podrobnosti a dokumentace nástroje agenda je k dispozici na uvedené adrese[20].

Opět jako v případě rozšiřování o nové formáty musíme upravit jak grafické rozhraní, tak vlastní logiku aplikace na serveru. Nejdříve upravme grafické rozhraní. Zadávání opakovaných úloh se provádí na stránce *schedule_files.ejs*. Zde se nachází část *Volba opakování*, ve které přidáme nový přepínač.

```
<div class="well" >
      <label>Volba opakování</label>
      <div>
      <input type="radio" name="period" value="just_in_time" checked
      onchange='showDatum()'> V daný čas    <br>
      <input type="radio" name="period" value="hourly"
      onchange='hideDatum()'> Každou hodinu <br>
      <input type="radio" name="period" value="daily"
      onchange='hideDatum()'> Každý den     <br>
      <input type="radio" name="period" value="weekly"
      onchange='hideDatum()'> Každý týden   <br>
      <input type="radio" name="period" value="monthly"
      onchange='hideDatum()'> Každý měsíc   <br>
      <input type="radio" name="period" value="ukazka_perioda"
      onchange='hideDatum()'> Jednou za 2d, 4h a 6m    <br>
      </div>
</div>
```

Příklad 34 - přidání nové možnosti periodického zpracování

Tímto jsme přidali novou možnost přepínače *„Jednou za 2d, 4h a 6m”.* To je vše, co je nutné udělat v uživatelském rozhraní. Zbývá tedy upravit serverovou část aplikace.

Zpracování naplánovaných úloh probíhá ve funkci *schedule_task*, která je umístěna v souboru *route.js*. Nás zajímá řádek *switch (period).* Na tomto řádu začíná přepínač, který určuje, jak bude úloha vykonána. V našem konkrétním případě tedy přidáme následující řádky kódu.

```
case "ukazka_perioda":
      {
            period_string = "Každý 2d, 4h a 6m";
            schedule_string = "2 days, 4 hours and 6
            minutes";
            break;
      }
```

Příklad 35 - přidání zpracování periodické úlohy

Proč je v první řádku *„ukazka_perioda“* je snad jasné, pojmenovali jsme si tak náš přidaný přepínač. Proměnná *period_string* je proměnná, jejíž obsah bude vypsán do zápisu o průběhu úlohy, má význam popisu dané periody. V zápise se poté objeví „Spouštěna: Každý 2d, 4h a 6m.“. A to je opět

vše, co je nutné upravit. Aplikace nyní umožňuje spouštět úlohy v pravidelném intervalu jednou za dva dny, čtyři hodiny a šest minut.

V kapitole *Modulární struktura aplikace* byl popsán postup rozšíření funkcionalit aplikace. Toto rozšiřování sice vyžaduje přímý zásah do kódu, ale z ukázek je zřejmé, že se jedná o krátké části kódu, které by měl i mírně pokročilý programátor zvládnout dopsat. Víceméně se jedná o pouhé zkopírování částí kódu a nahrazení identifikátorů funkcí a proměnných novými. Nejnáročnější část je vytvoření nové funkce pro zpracování dokumentu v adresáři *modules*. Je potřeba najít odpovídající knihovnu a napsat obsluhu této funkce. I pokud pro *Node.js* není k dispozici vhodný balíček k provedení daných operací s dokumentem, je možné použít přímo knihovny nainstalované na operačním systému, kde *Node.js* běží.

Závěr

Cílem této práce bylo seznámit se s jazyky pro popis dat a možnostmi jejich validace a transformace. A poté na základě získaných znalostí vytvořit aplikaci, která bude umět provádět operace s těmito strukturami. V teoretické části se práce věnuje tomu, co to jazyky pro popis dat jsou, k čemu slouží a jaké jsou jejich prostředky pro validaci a transformaci. Jazyky pro popis dat slouží k popisu struktury a významu dat. Přidávají k uloženým datům další informace, které poté mohou být použity při interpretování dat. Byla zmíněna řada formátů, které byly za tímto účelem vytvořeny. Například *HTML*, *CSS*, *JSON*, *YAML* nebo *XML*. Jako hlavní formát, na kterém se demonstruje princip a užití těchto jazyků byl vybrán obecný značkovací jazyk *XML*. Jazyk *XML* byl vybrán z důvodů všeobecného rozšíření a dostupnosti nástrojů, pro validaci a transformaci dokumentů, zapsaných v tomto formátu. V práci byla krátce popsána historie jazyka a vysvětlena struktura, popsán postup tvorby konkrétních jazyků, vycházejících z formátu *XML*, a také možnosti validace a transformace těchto struktur. Dalším cílem této práce bylo navrhnout a vytvořit aplikaci, která bude umět provádět validace a transformace těchto dokumentů. Pro účel validace dokumentů byly popsány jazyky *DTD* a *XML Schema*. Oba dva jazyky slouží k popisu struktury a typu obsahu elementů daného *XML* jazyka. Jazyk *XML Schema* umožňuje oproti *DTD* přesněji specifikovat, jaký obsah je očekáván v daném elementu, například určit rozsah číselné hodnoty. Opět bylo na příkladech ukázáno, jak sestrojit k danému *XML* dokumentu definici. Dále byla popsána transformace *XML* dokumentů pomocí jazyka *XSLT*. Pomocí transformace lze například převést dokument na jiný formát a tím usnadnit přenos dat mezi různými systémy. V práci byl jazyk *XSLT* popsán a bylo názorně ukázáno, jak vytvořit předpis pro transformace dokumentů. Popisem informací, zmíněných v tomto odstavci se zabývá kapitola *Jazyky pro popis da.*

Získané znalosti byly poté použity k tvorbě aplikace, která bude provádět popsané operace. V rámci práce byla tato aplikace popsána jak teoreticky, tak reálně vytvořena. Aplikace umožňuje správu uživatelů pomocí jednoduché registrace. Uživatelům je poté umožněno spravovat své nahrané soubory a provádět jejich validace a transformace. Je také implementováno pokročilé spouštění úloh. Uživatel může úlohy naplánovat na zvolený čas, případně nastavit jejich opakované spouštění v pravidelných časových intervalech. Řešení bylo navrženo modulárně, je tedy možné funkčnost rozšířit o další formáty dokumentů, případně přidat nové možnosti spouštění. Postup rozšíření aplikace byl zdokumentován a názorně předveden na příkladech. Vlastní aplikace pro transformaci a validaci těchto datových struktur je popsána v kapitole *Vlastní aplikace pro validaci a transformaci*. V kapitole *Návrh*

aplikace bylo popsáno, jaké jsou požadavky na aplikaci. V návaznosti na to bylo v následující kapitole *Zvolené technologie* vysvětleno, jaké programovací prostředky byly vybrány a použity při tvorbě aplikace. Navržená aplikace je dostupná na webové stránce, z toho důvodu se aplikace dělí na část klientskou a serverovou. Aplikace by také měla běžet bez potřeb instalace speciálních balíčků, měla by klást minimální nároky na uživatele. Z těchto důvodů byly vybrány následující technologie. Na klientské části byl použit jazyk *HTML* a *JavaScript*. Jazyk *HTML* není potřeba představovat, je hlavním jazykem pro tvorbu webových stránek. Jazyk *JavaScript* naopak umožňuje provádět skriptování na straně klienta a tím například umožnit interakci s uživatelem. Je implementován ve většině moderních prohlížečů. Na straně serveru byla použita technologie *Node.js*, díky čemuž bylo možné použít programovací jazyk *JavaScript* i na straně serveru, čímž se usnadnil vývoj aplikace. Dále umožňuje snadné horizontální škálování výkonu, je tedy možné zvýšit výkonost aplikace spuštěním na více jádrových serverech. V současné chvíli se může zdát výkon dostatečný, ale v případě nasazení a využívání v reálných situacích, je možné, že služeb aplikace bude využívat větší počet lidí, kteří mohou mít naplánováno značné množství operací, a tedy server může být značně vytížen. S tím také souvisí podpora běhu v *cloudu*. Velcí poskytovatelé cloudových služeb jako *Google* nebo *Microsoft* nabízejí možnost jednoduše hostovat aplikace napsané v *Node.js* a tím se dále usnadňuje jak nasazení aplikace, tak její správa. Pro *Node.js* existuje řada knihoven neboli balíčků. Balíčky, které byly použity při tvorbě samotné aplikace, jsou také popsány v kapitole *Zvolené technologie*. V kapitolách *Struktura aplikace* a *Zdrojový kód aplikace* byla popsána struktura a aplikace a ukázány použité postupy pomocí krátkých ukázek kódu. Bylo zde ukázáno, jak probíhá inicializace aplikace a následné zpracovávání dotazů uživatele. O běh webového serveru, který dotazy přijímá se stará balíček *Express*. Byly zde také popsány všechny části aplikace. Aplikace obsahuje jednoduché uživatelské rozhraní. Uživateli je umožněno se registrovat a poté se přihlásit a pracovat s dokumenty. Dokumenty nahrává do své složky, umístěné na serveru a může k nim tedy přistupovat z různých míst. S uloženými dokumenty je možné provádět operace, které byly popsány v teoretické části. Práce popisuje, jaké byly použity programové prostředky pro práci s dokumenty, a je vysvětleno, jak je provádění těchto operací implementováno v aplikaci.

Aplikací, umožňující online provádění zmíněných operací s *XML* dokumenty existuje celá řada. Jedním z přínosů vytvořené aplikace je zmíněné ukládání dat na serveru a tedy možnost k nim přistupovat z různých míst. Dalším přínosem je poté implementace pokročilého zpracování úloh. Tato implementace je popsána v kapitole *Pokročilé spouštění úloh*. Na snímku stránky aplikace je názorně vysvětleno, jak se provádí zadání takové úlohy ke zpracování a kde se uchovávají informace o spuštěných a proběhlých úlohách. Takovou úlohou může být validace nebo transformace, v případě validace se jedná o kontrolu „*well-formed*“ dokumentu nebo validace pomocí *XML Schema*. Transformace je prováděna pomocí jazyka *XSLT*. Aplikace umožňuje naplánovat úlohu na zadaný čas nebo ji spouštět v pravidelných intervalech. V současné době aplikace umožňuje spouštět úlohy každou hodinu, den, týden a měsíc. Toto jsou předdefinované intervaly spouštění, jak je ukázáno v dalších kapitolách, je možné velice snadno funkčnost rozšířit a přidat libovolné možnosti intervalů spouštění. Další možností pokročilého spouštění úloh je možnost zadat dávkovou úlohu. Je možné zadat více dokumentů, nad kterými bude daná operace postupně provedena. Aplikace také umožňuje provádět tyto operace na dokumentech, které jsou umístěny na vzdáleném uložišti a jsou dostupné na zadané *URL* adrese. Výstup z těchto operací je potom uložen v adresáři uživatele v textových souborech. Ten si poté může výsledky zpětně prohlédnout a pracovat s nimi dále. Ke spouštění úloh v daný čas nebo v zadaných intervalech je použit balíček a*genda*. Tento balíček umožňuje definovat

úlohu a spustit ji dle zadaných parametrů. *Agenda* ukládá data úloh do databáze, kde jsou uchovány i v případě pádu aplikace. V rámci aplikace byla vytvořena funkce, která uložená data vezme a použije je k znovu spuštění naplánovaných úloh v případě pádu aplikace či serveru.

Jedním z hlavních cílů práce bylo, aby byla aplikace navržena jako modulární a tedy umožňovala rozšíření funkcionality přidáním nových podporovaných formátů, případně nových možností pokročilého zpracování úloh. V kapitole *Rozšíření podporovaných formátů* bylo popsáno přidání validačního formátu *Relax NG*. Bylo zde ukázáno, jak přidat tuto možnost jak do uživatelského rozhraní, tak i jak provést přidání zpracovávání dané úlohy na serverové části. V rámci ukázek přidání nového formátu, který umí aplikace zpracovat, bylo zároveň vysvětleno, jak takové zpracování úloh probíhá. V kapitole *Rozšíření možností spouštění naplánovaných úloh* se poté popisuje, jak přidat novou možnost spouštění aplikace, konkrétně se jednalo o přidání specifického intervalu spouštění. Stejně jako v případě rozšiřování podporovaných formátů, i zde bylo pomocí ukázek kódu aplikace popsáno, jak je prováděno toto plánování úloh. Zmíněné dvě kapitolu vysvětlují, jak přidat novou funkcionalitu. Jak je zřejmé, přidání vyžaduje zásah do aplikace a základní znalosti programování. Aplikace nebyla navržena a ani zamýšlena tak, aby poskytovala jisté uživatelské rozhraní pro tento účel. Ovšem, jak je na uvedených příkladech ukázáno, jedná se o dopsání krátkých částí kódu, které zvládne i mírně pokročilý programátor. Z tohoto důvodu je i uživatelské rozhraní aplikace navrženo co nejjednodušeji. Tím je zjednodušeno přidávání nových ovládacích prvků a celkově celá rozšiřitelnost. Aplikace demonstruje možnosti využití jazyků pro popis dat a implementaci jednoduše rozšiřitelné aplikace. Jádro aplikace zařizuje běh webového serveru, poskytuje uživateli rozhraní, pro přístup do aplikace, a zpracovává jeho požadavky. V případě potřeby rozšíření se pouze doplní ovládací prvky do uživatelského rozhraní a nové možnosti zpracování do funkcí na serveru. Vše toto bylo v této práci zdokumentováno a je tedy možné aplikaci rozšířit a tím umožnit uživateli práci s vybraným formátem, případně nastavit možnosti spouštění dle požadavků uživatele. Aplikace popsaná v této práci byla vytvořena a je dostupná na adrese *http://xmlcheck.nti.tul.cz:8081/.* Je možné se volně registrovat a využívat ji.

Seznam použité literatury

[1] HAROLD, Elliotte Rusty a W MEANS. XML in a nutshell. 3rd ed. Sebastopol, CA: O'Reilly, c2004, xix, 689 p. In a nutshell (O'Reilly & Associates). ISBN 0596007647.

[2] VAN DER VLIST, Eric. XML schema. Beijing: O'Reilly, c2002. ISBN 0596002521.

[3] DOUG TIDWELL. XSLT. 2nd ed. Farnham: O'Reilly, 2007. ISBN 9780596527211.

[4] Node.js runtime. Node.js [online]. Node.js Foundation, 2016 [cit. 2016-09-22]. Dostupné z: https://nodejs.org/.

[5] Bootstrap framework. Bootstrap [online]. Bootstrap Authors and Twitter, Inc., 2016 [cit. 2016-09-22]. Dostupné z: https://github.com/twbs/bootstrap.

[6] Embedded JavaScript templates for node. Embedded JavaScript [online]. TJ Holowaychuk, 2010 [cit. 2016-09-22]. Dostupné z: https://github.com/tj/ejs.

[7] Express web framework. Express [online]. Node.js Foundation, 2016 [cit. 2016-09-22]. Dostupné z: http://expressjs.com/.

[8] MongoDB for giant ideas. MongoDB [online]. MongoDB, Inc., 2016 [cit. 2016-09-22]. Dostupné z: https://www.mongodb.com/.

[9] Mongoose mongodb modeling for nodejs. Mongoose [online]. LearnBoost, 2010 [cit. 2016-09-22]. Dostupné z: http://mongoosejs.com/.

[10] Passport authentication for Node.js. Passport [online]. Jared Hanson, 2016 [cit. 2016-09-22]. Dostupné z: http://passportjs.org/.

[11] Path module. Path [online]. Node.js Foundation, 2016 [cit. 2016-09-22]. Dostupné z: https://nodejs.org/docs/latest/api/path.html#path_path.

[12] Node.js: fs-extra. Fs-extra [online]. JP Richardson, 2016 [cit. 2016-09-22]. Dostupné z: https://github.com/jprichardson/node-fs-extra.

[13] Connect-flash middleware. Connect-flash [online]. Jared Hanson, 2013 [cit. 2016-09-22]. Dostupné z: https://github.com/jaredhanson/connect-flash.

[14] Node-url. Node-url [online]. Node.js Foundation, 2016 [cit. 2016-09-22]. Dostupné z: https://github.com/defunctzombie/node-url.

[15] Multer middleware. Multer [online]. Node.js Foundation, 2016 [cit. 2016-09-22]. Dostupné z: https://github.com/expressjs/multer.

[16] Request - Simplified HTTP client. Request [online]. 2016 [cit. 2016-09-22]. Dostupné z: https://github.com/request/request.

[17] Cookie-parser. Cookie-parser [online]. 2016 [cit. 2016-09-22]. Dostupné z: https://github.com/expressjs/cookie-parser.

[18] Body-parser. Body-parser [online]. 2016 [cit. 2016-09-22]. Dostupné z: https://github.com/expressjs/body-parser.

[19] Express-session middleware. Express-session [online]. 2016 [cit. 2016-09-22]. Dostupné z: https://github.com/expressjs/session.

[20] Agenda light-weight job scheduling library. Agenda [online]. Ryan Schmukler, 2013 [cit. 2016-09-22]. Dostupné z: https://github.com/rschmukler/agenda.

[21] Libxmljs. Libxmljs [online]. 2016 [cit. 2016-09-22]. Dostupné z: https://github.com/libxmljs/libxmljs.

[22] Xmllint - command line XML tool. Xmllint [online]. 2016 [cit. 2016-09-22]. Dostupné z: http://xmlsoft.org/xmllint.html.