Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Webové služby v teorii a praxi

2021

Bc. Ivoš Apolenář

Vedoucí práce: Mgr. Petr Krajča, Ph.D.

Studijní obor: Aplikovaná informatika, prezenční forma

Bibliografické údaje

Autor:	Bc. Ivoš Apolenář
Název práce:	Webové služby v teorii a praxi
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2021
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Petr Krajča, Ph.D.
Počet stran:	95
Přílohy:	1 DVD
Jazyk práce:	český

Bibliograhic info

Author:	Bc. Ivoš Apolenář
Title:	Webservices in Theory and Practice
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2021
Study field:	Applied Computer Science, full-time form
Supervisor:	Mgr. Petr Krajča, Ph.D.
Page count:	95
Supplements:	1 DVD
Thesis language:	czech

Anotace

Tato diplomová práce se zabývá webovými službami využívající protokol SOAP a styl REST. Formálně tyto přístupy popisuje, zejména ve spolupráci s protokolem HTTP. Srovnává jejich rysy a použitelnost, včetně technologií a nástrojů s nimi spojených. Součástí práce je i praktická demonstrace obou přístupů v rámci vybraných programovacích jazyků a technologií formou jednoduché aplikace pro správu publikací. Práce rovněž srovnává efektivitu obou přístupů pomocí experimentů.

Synopsis

This diploma thesis deals with web services which use SOAP protocol and REST style. Formally describes them, especially in cooperation with the HTTP protocol, and compares their features and usability, including the technologies and tools associated with them. The thesis also includes a practical demonstration of both approaches in selected programming languages and technologies in the way of a simple application for managing publications. The thesis also compares the effectiveness of both approaches via experiments.

Klíčová slova: webové služby; SOAP; REST; HTTP

Keywords: web services, SOAP, REST, HTTP

Děkuji Mgr. Petru Krajčovi, Ph.D. za rady a připomínky při vedení této práce. Dále chci poděkovat rodičům a všem, kteří mě po celou dobu studia podporovali.

datum odevzdání práce podpis autora

Obsah

1 Úvod **9**
1.1 Motivace . . . 9
1.2 Obsah práce . . . 9

2 Protokol HTTP **10**
2.1 HTTP komunikace . . . 10
2.2 Požadavek klienta . . . 11
2.2.1 Metody HTTP . . . 11
2.3 Odpověď serveru . . . 12
2.3.1 Stavové kódy . . . 13
2.4 Hlavičky . . . 15
2.5 Bezpečnost . . . 16
2.5.1 HTTPS . . . 16
2.5.2 Autentizační mechanismus . . . 16
2.5.3 Typy autentizace . . . 17
2.6 Shrnutí . . . 18

3 Webové služby **19**
3.1 Obecný popis . . . 19
3.2 SOAP . . . 19
3.2.1 Formát zpráv . . . 20
3.2.2 Styl zpráv . . . 22
3.2.3 Styl kódování . . . 22
3.2.4 SOAP a HTTP . . . 22
3.2.5 Optimalizace zpráv – MTOM . . . 23
3.2.6 WSDL . . . 23
3.2.6.1 Popis . . . 25
3.2.6.2 Styly vazeb . . . 27
3.2.7 Bezpečnost . . . 32
3.3 REST . . . 32
3.3.1 Omezení definující REST . . . 32
3.3.1.1 Klient-server . . . 33
3.3.1.2 Stateless (bezstavovost) . . . 33
3.3.1.3 Cacheable (možnost využití mezipaměti) . . . 33
3.3.1.4 Uniform interface (jednotné rozhraní) . . . 34
3.3.1.5 Layered system (vrstevný systém) . . . 34
3.3.1.6 Code on demand (kód na vyžádání) . . . 35
3.3.2 REST společně s HTTP . . . 35
3.3.3 Bezpečnost . . . 37
3.3.3.1 JWT . . . 37
3.4 Srovnání SOAP a REST . . . 38

3.5 Shrnutí . . . 39

4 **Technologie a nástroje pro webové služby** **41**
4.1 SOAP . . . 41
4.1.1 UDDI . . . 41
4.1.2 WS specifikace . . . 42
4.1.3 SoapUI . . . 43
4.2 REST . . . 44
4.2.1 Jazyky pro modelování API . . . 44
4.2.2 Nástroje pro práci s API . . . 47
4.2.3 OData . . . 47
4.2.4 Postman . . . 48
4.3 Mock server . . . 48
4.4 Apache JMeter . . . 49
4.5 Shrnutí . . . 49

5 **Technologie pro webové služby použité v praktické části** **50**
5.1 Java . . . 50
5.1.1 JAX-WS . . . 51
5.1.1.1 Server . . . 51
5.1.1.2 Klient . . . 52
5.1.2 JAX-RS . . . 52
5.1.2.1 Server . . . 52
5.1.2.2 Klient . . . 54
5.2 PHP . . . 54
5.2.1 SOAP extenze . . . 55
5.2.1.1 Server . . . 55
5.2.1.2 Klient . . . 56
5.2.2 Slim framework . . . 57
5.2.3 Guzzle . . . 58
5.3 Python . . . 58
5.3.1 Spyne . . . 58
5.3.2 Zeep . . . 59
5.3.3 Flask . . . 59
5.3.4 Requests . . . 61
5.4 JavaScript/Node.js . . . 61
5.4.1 Modul Soap . . . 61
5.4.1.1 Server . . . 61
5.4.1.2 Klient . . . 62
5.4.2 Express . . . 62
5.4.3 Axios . . . 63
5.5 Shrnutí . . . 63

6 Praktická část **65**
6.1 Aplikace . . . 65
6.1.1 Rozhraní webových služeb . . . 65
6.1.1.1 SOAP API . . . 66
6.1.1.2 REST API . . . 66
6.1.2 Server . . . 68
6.1.2.1 Databáze . . . 68
6.1.3 Klient . . . 68
6.2 Implementace . . . 69
6.2.1 Java . . . 70
6.2.2 PHP . . . 70
6.2.3 Python . . . 71
6.2.4 JavaScript/Node.js . . . 71
6.3 Zhodnocení implementací . . . 72
6.3.1 Java . . . 72
6.3.2 PHP . . . 73
6.3.3 Python . . . 73
6.3.4 JavaScript/Node.js . . . 74
6.4 Shrnutí . . . 74

7 Experimenty **75**
7.1 Testovací prostředí . . . 75
7.1.1 Konfigurace . . . 75
7.2 Testovací scénář . . . 77
7.3 Výsledky . . . 78
7.3.1 Jednotlivé implementace . . . 78
7.3.2 Webové služby . . . 83
7.4 Shrnutí . . . 85

8 SOAP proti REST **86**
8.1 Standard proti stylu . . . 86
8.2 Návrh aplikace . . . 86
8.3 Implementace . . . 86
8.4 Využitelnost . . . 87

Závěr **89**

Conclusions **90**

A Obsah přiloženého DVD **91**

Literatura **92**

Seznam obrázků

1 HTTP autentizace, zdroj: [10] 17
2 Formát SOAP zprávy, zdroj: [16] 20
3 Formát multipart SOAP zprávy, zdroj: [16] 24
4 WSDL – typy operací 27
5 Úrovně RESTful implementací, zdroj: [29] 36
6 Vzájemný vztah SOAP-WSDL-UDDI zdroj: [34] 42
7 Schéma databáze pro testovací aplikace 68
8 Ukázka klientské aplikace 69
9 Master–Slave princip využitý pro experimenty [59] 75
10 Experimenty – ukázka možného výstupu po dokončení testu pomocí aplikace Apache JMeter 78
11 Experimenty – srovnání implementací: čas vrácení odpovědi (ms) 81
12 Experimenty – srovnání implementací: přenesené bajty 81
13 Experimenty – srovnání implementací: časy trvání testů (mm:ss) . 82
14 Experimenty – srovnání implementací: průměrné vytížení CPU (%) 82
15 Experimenty – srovnání dle typu webové služby: čas vrácení odpovědi (ms) 84
16 Experimenty – srovnání dle typu webové služby: přenesené bajty . 84
17 Experimenty – srovnání obecné rychlosti odpovědi v průběhu testů dle typu webové služby pomocí průměrování a vzorkování 85
18 Srovnání popularity SOAP a REST dle Stack Overflow, zdroj: [60] 88

Seznam tabulek

1 Popis vybraných stavových HTTP kódů 14
2 Stručné srovnání SOAP – REST, zdroj: [32, 33] 40
3 Experimenty – srovnání implementací: čas vrácení odpovědi (ms) 79
4 Experimenty – srovnání implementací: přenesené bajty 80
5 Experimenty – srovnání implementací: časy trvání testů (mm:ss) . 80
6 Experimenty – srovnání implementací: průměrné vytížení CPU (%) 80
7 Experimenty – srovnání SOAP a REST 83

Seznam zdrojových kódů

1 Ukázka HTTP/1.1 požadavku 11
2 Ukázka HTTP/1.1 odpovědi 13
3 SOAP požadavek pomocí HTTP, zdroj: [13] 23
4 SOAP odpověď pomocí HTTP, zdroj: [13] 25
5 WSDL – RPC/encoded styl, zdroj: [24] 29
6 WSDL – RPC/literal styl, zdroj: [24] 29
7 WSDL – Document/literal styl, zdroj: [24] 30

8 WSDL – Document/literal wrapped styl, zdroj: [24] 31
9 OpenAPI – definice REST API ve formátu YAML 45
10 API Blueprint – ukázka definice REST API 46
11 RAML – ukázka definice REST API 46
12 JAX-WS – implementace koncového bodu služby, zdroj: [47] . . . 52
13 JAX-WS – implementace klienta 52
14 JAX-RS – implementace zdroje 53
15 JAX-RS – generický error handler na straně serveru 53
16 JAX-RS – filtrace požadavku pro autentizaci na straně serveru . . 54
17 JAX-RS – implementace klienta 55
18 PHP SOAP – implementace koncového bodu služby 56
19 PHP SOAP – implementace klienta 56
20 Slim framework – implementace zdroje 57
21 Slim framework – middleware kontrolující autentizaci požadavku . 57
22 Slim framework – generický error handler 58
23 Guzzle – implementace klienta 58
24 Spyne – implementace koncového bodu služby 59
25 Zeep – implementace klienta 59
26 Flask – implementace zdroje 60
27 Flask – middleware kontrolující autentizaci požadavku 60
28 Flask – generický error handler 60
29 Requests – implementace klienta 61
30 Node.js SOAP – implementace koncového bodu 62
31 Node.js SOAP – implementace klienta 62
32 Express – implementace zdroje 63
33 Express – middleware kontrolující autentizaci požadavku 63
34 Express – generický error handler 64
35 Axios – implementace klienta 64

1 Úvod

1.1 Motivace

V dnešní době webové služby patří mezi základní stavební kameny téměř každého softwaru. Ustupuje se od monolitických architektur a dává se přednost architektuře založené na malých a jednoúčelových komponentách, kterým se říká mikroslužby [1]. Ty spolu navzájem komunikují často skrze webové služby. Tento přístup v sobě, při správném uchopení, skýtá řadu výhod. Každá z mikroslužeb může být implementována v programovacím jazyce a prostředí nejvhodnějším pro daný účel. Jednodušeji se spravuje, a proto se často nasazuje do cloudových řešení. V cloudu se mikroslužby zpravidla shlukují do clusterů, vystupujících jako jedna služba. To přináší mimo jiné horizontální škálovatelnost umožňující regulaci výkonu, odolnost proti výpadkům či plynulý a rychlý přechod na nové verze. A to vše s vysokou mírou automatizace.

Architektura mikroslužeb ovšem není jenom o výhodách. Již na první pohled je zřejmé, že přináší i řadu problémů. Jde například o složitější celkovou údržbu systému či vyšší provozní náklady. Musí se spravovat a rozvíjet rozhraní pro vzájemnou komunikaci. Komunikace navíc způsobuje další režii s přenosem a výměnou informací napříč službami. Proto je důležité se snažit komunikaci co nejvíce zefektivnit a hledat co nejvýhodnější řešení v rámci programovacího jazyka, platformy či nástrojů pro vývoj pomocí webových služeb.

1.2 Obsah práce

Úkolem tohoto dokumentu je seznámení s přístupy pro vývoj webových služeb – protokolem SOAP a architektonickým stylem REST. Oba přístupy v praxi silně souvisí s protokolem HTTP, a proto mu bude věnována následující kapitola. Pak dojde na formální popsání a srovnání samotných přístupů. Budou zmíněny i související pojmy a technologie tvořící jejich ekosystémy. Poté přijde na řadu seznámení s vybranými programovacími jazyky a technologiemi použitými v praktické části. Po praktické části budou představeny experimenty pro porovnání efektivity obou přístupů. V závěru práce dojde k celkovému porovnání obou zkoumaných přístupů.

2 Protokol HTTP

HTTP (Hypertext transfer protocol) je internetový protokol. Podle TCP/IP modelu pracuje na aplikační vrstvě [2, 3]. Používá se na výměnu dat (tzn. zdrojů) a funguje na principu požadavek-odpověď, tedy klient-server. První verze HTTP byla vyvinuta vědci z CERNu[1] v čele s Timem Berners-Lee v letech 1989–1991.

Základními rysy protokolu jsou jednoduchost, rozšiřitelnost a bezstavovost. Komunikace mezi klientem a serverem je srozumitelná i pro člověka, což jej činí jednoduchým na používání. Rozšiřitelnost je umožněna pomocí HTTP hlaviček a jejich správnou interpretací klientem a serverem. Bezstavovost znamená, že mezi jednotlivými požadavky neexistuje žádná vazba a tedy i stav, který by se mohl mezi více požadavky měnit.

V roce 1997 byla vydána první standardizovaná verze protokolu – HTTP/1.1. Tato verze byla poprvé definována formou RFC[2] dokumentů a byla klíčová, jelikož vyjasnila zmatky spjaté s protokolem a přidala značná vylepšení. Standard byl dvakrát revidován a to v letech 1999 a 2014.

S obrovským rozšířením internetu a pokroku v oblasti technologií začal být protokol HTTP/1.1 zastaralým. V reakci na tuto skutečnost vnikl v roce 2015 protokol HTTP/2 [4], který přináší řadu vylepšení zejména v oblasti výkonu jako např.:

- Je protokolem binárním na rozdíl od předchozí textové verze, data jsou přenášena v tzv. rámcích.
- Paralelní dotazy v rámci jednoho TCP připojení.
- Komprese hlaviček – např. u malých dat mohou být hlavičky vetší než samotná data.
- Server push technika – umožňuje serveru zaslat dodatečná data, která nebyla požadována, ale server na ně očekává požadavek v budoucnu. Klient si data uloží do mezipaměti a v případě potřeby je použije a tím odpadne nutnost odesílat další požadavek na server.

Navzdory tomu, že protokol HTTP/2 je modernější a rychlejší, je stále v dnešní době poměrně často využívána verze protokolu HTTP/1.1.

2.1 HTTP komunikace

Komunikace protokolem HTTP probíhá zasíláním tzv. HTTP zpráv [5]. Klient naváže se serverem spojení, zašle požadavek a server klientovi odpoví. Zpráva zaslaná klientem se nazývá požadavek, serverem odpověď.

[1] Mezinárodní evropská organizace pro jaderný výzkum sídlící v Ženevě.

[2] Request for Comments – dokumenty popisující internetové protokoly považovány de facto za standardy.

2.2 Požadavek klienta

Požadavek tvoří následujících položky:

- HTTP metoda – specifikuje účel dotazu, nejpoužívanější jsou metody `GET` (získání dat ze serveru) a `POST` (zaslání dat na server), viz 2.2.1.
- Adresa zdroje. Je definována jako relativní URL (Uniform Resource Locator) vzhledem k adrese serveru.
- Verze protokolu HTTP.
- Hlavička nesoucí další užitečné informace.
- Volitelné, tzn. tělo požadavku. Používá se zejména s metodou POST pro zaslání dat na server.

```
1 GET /aktuality HTTP/1.1
2 Accept: */*
3 Accept-Encoding: gzip, deflate
4 Connection: keep-alive
5 Host: www.inf.upol.cz
6 User-Agent: HTTPie/0.9.8
```

Zdrojový kód 1: Ukázka HTTP/1.1 požadavku

Zdrojový kód 1 zobrazuje ukázku HTTP/1.1 požadavku pro získání aktualit ze serveru www.inf.upol.cz. První řádek definuje HTTP metodu (`GET`), relativní URL (celá adresa by vypadala následovně – www.inf.upol.cz/aktuality) a verzi protokolu. Prvnímu řádku se říká – stavový řádek. Následují hlavičky, kdy každá hlavička je na samostatném řádku. Případné tělo požadavku by se nacházelo za hlavičkami a bylo by odděleno prázdným řádkem.

2.2.1 Metody HTTP

HTTP metody jsou popsány následovně:

- *GET* – Získá reprezentaci zdroje (dokumentu) na cílové adrese. Může obsahovat i tělo s daty, nicméně tento způsob se v praxi nevyužívá.
- *HEAD* – Stejná jako `GET` s tím rozdílem, že odpověď ze serveru obsahuje pouze stavový řádek s hlavičkami.
- *POST* – Vytváří nový datový zdroj v dané reprezentaci. Tělo požadavku obsahuje data, která se mají uložit na serveru. Příkladem použití může být odesílání HTML formuláře.

- *PUT* – Nahrazuje stávající zdroje na dané adrese daty obsaženými v těle požadavku.
- *DELETE* – Odstraní stávající zdroj na cílové adrese.
- *CONNECT* – Založí tunelované spojení se zdrojem. Používá se pro zabezpečené připojení pomocí TLS (Transport Layer Security) skrze proxy server.
- *OPTIONS* – Zjistí jaké metody podporuje zdroj na cílové URL adrese.
- *TRACE* – Slouží ke zjištění cesty dotazu, např. přes jaké proxy servery server požadavek prošel až k cíli.
- *PATCH* – Upravuje zdroj na cílové adrese.

Kromě popisu je můžeme dělit do tří skupin podle jejich vlastností – bezpečné, idempotentní a tzn. „Cacheable“ metody. Dělení metod do skupin není zaručeno protokolem ani případnou implementací. Nutno však podotknout, že jejich ignorací se vystavujeme přinejmenším riziku neočekávaného chovaní vůči uživateli.

Bezpečné metody

Jsou takové, jejichž chování nemění data ani neovlivní stav serveru. Jinými slovy, měly by být použity pouze pro získání dat. Do této skupiny patří metody: `GET`, `HEAD`, `OPTIONS` a `TRACE`.

Idempotentní metody

Tuto skupinu tvoří metody, které jsou definovány jako idempotentní. To znamená, že opakováním stejného požadavku vznikne stav stejný jako ten, který vznikne jediným použitím daného požadavku. Jelikož bezpečné metody nemění stav, jsou tedy i přirozeně idempotentní stejně jako metody `PUT` a `DELETE`.

„Cacheable“ metody

Jsou to metody, jejichž výsledky se mohou ukládat do mezipaměti pro budoucí použití. Specifikace do této skupiny uvádí metody `GET`, `HEAD` a `POST`. Nicméně následně zmiňuje, že drtivá většina implementací podporuje pouze první dvě metody.

2.3 Odpověď serveru

Odpověď serveru obsahuje verzi protokolu HTTP a stavový číselný kód s jeho textovým popisem. Stejně jako požadavek, obsahuje také hlavičky. Na konci odpovědi je pak volitelně tzv. tělo odpovědi, obsahující data dotazovaného zdroje.

Zdrojový kód 2 zobrazuje odpověď na požadavek ze zdrojového kódu 1. Prvnímu řádku se stejně jako u požadavku říká „stavový“. Za hlavičkami je prázdný

```
1  HTTP/1.1 200 OK
2  Cache-Control: no-store, no-cache, must-revalidate
3  Connection: Keep-Alive
4  Content-Encoding: gzip
5  Content-Length: 5073
6  Content-Type: text/html; charset=UTF-8
7  Date: Fri, 29 Nov 2019 15:26:20 GMT
8  Expires: Thu, 19 Nov 1981 08:52:00 GMT
9  Keep-Alive: timeout=5, max=100
10 Pragma: no-cache
11 Server: Apache/2.4.38 (Debian)
12 Set-Cookie: PHPSESSID=p69tt7c0seggbicil0j62mlapg; path=/
13 Vary: Accept-Encoding
14
15 <!DOCTYPE html>
```

Zdrojový kód 2: Ukázka HTTP/1.1 odpovědi

řádek a po něm tělo odpovědi. To tvoří HTML (Hypertext Markup Language) dokument, který je pro jednoduchost a stručnost reprezentován pouze prvním řádkem. Jak si lze všimnout, formát těla odpovědi souvisí s hlavičkou `Content-Length`.

2.3.1 Stavové kódy

Stavovými kódy server informuje, zda-li se zpracování HTTP požadavku zaslaného klientem úspěšně provedlo, nebo došlo při zpracovávání k nějakému neočekávanému stavu. Jsou reprezentovány jako třímístná čísla rozdělená do 5 kategorií dle první číslice:

- *1xx* – informační, potvrzuje přijetí požadavku, případně oznamuje, že se požadavek zpracovává.
- *2xx* – požadavek byl úspěšně přijat, pochopen a vykonán.
- *3xx* – přesměrování, další akce jsou nutné pro dokončení požadavku.
- *4xx* – chyby způsobeny klientem, požadavek není syntakticky správně, případně nemůže být dokončen z jiného důvodu.
- *5xx* – chyby způsobeny serverem, serveru se nezdařilo vykonat pravděpodobně správný požadavek.

Tabulka 1 popisuje vybrané, nejpoužívanější stavové kódy.

Tabulka 1: Popis vybraných stavových HTTP kódů

Kód	**Název**	**Popis**
200	OK	Nejběžnější odpověď, signalizuje úspěšné provedení požadavku.
201	Created	Značí úspěšné vytvoření nového zdroje na serveru.
202	Accepted	Požadavek byl přijat, ale zatím nebyl celý zpracován.
204	No content	Požadavek se úspěšně provedl, ale server nevrací žádná data.
301	Moved Permanently	Zdroj byl trvale přesunut na URL definovanou v hlavičce *Location*.
302	Found	Zdroj byl dočasně přesunut na URL definovanou v hlavičce *Location*.
304	Not Modified	Signalizuje, že není třeba přenášet zdroj jelikož nedošlo od minulého požadavku k jeho změně.
400	Bad Request	Server není schopen zpracovat požadavek, kvůli chybě na straně klienta (např. požadavek má špatnou syntaxi).
401	Unauthorized	Pro úspěšné provedení požadavku se musí klient autentizovat. Používá se s hlavičkou *WWW-Authenticate*.
403	Forbidden	Kliet nemá potřebná práva na provedení požadavku.
404	Not Found	Server nebyl schopen nalézt požadovaný zdroj.
405	Method Not Allowed	Daná metoda není povolena pro dotazovaný zdroj. Odpověď by měla obsahovat hlavičku *Allow* s podporovanými metodami.
408	Request Timeout	Server neobdržel celý požadavek po dobu, kterou byl připraven čekat a ukončuje spojení.
415	Unsupported Media Type	Data v těle požadavku jsou zaslána v nepodporovaném formátu.
500	Internal Server Error	Na serveru vznikla neočekávána chyba při zpracování požadavku. Jedná se o obecný chybový stav.
503	Service Unavailable	Signalizuje neschopnost zpracovat požadavek z důvodu přetížení či údržby serveru.
504	Gateway Timeout	Server fungující jako brána nebo proxy nedostal včas od cílového serveru odpověď a proto nemůže dokončit požadavek.

2.4 Hlavičky

Jak již bylo zmíněno, zpravidla každý požadavek i odpověď obsahují hlavičky pro předání doplňujících informací [2, 6, 7, 8]. Tyto informace mohou být využity pro rozšíření funkčnosti, ovšem za předpokladu, že jim bude klient i server rozumět. Hlavičky[3] by se daly rozdělit do několika skupin podle svého uplatnění. Následující text je popíše a přidá definice vybraných hlaviček z každé skupiny.

Obecné hlavičky

Používají se u požadavku i odpovědi, ale nijak nesouvisí s daty v těle (požadavku či odpovědi).

- *Date* – datum a čas vytvoření zprávy.
- *Cache-control* – slouží pro nastavení chování mezipaměti.
- *Connection* – říká, zda-li má server spojení stále udržovat nebo ho ukončit.

Hlavičky požadavku

Přidávají dodatečné informace o klientovi nebo požadovaném zdroji. Povinnou hlavičkou je v protokole HTTP/1.1 hlavička *HOST*, z důvodu rozlišení virtuálních serverů, které mohou být spuštěny v rámci jednoho serveru fyzického.

- *Accept* – definuje MIME typy, kterým klient rozumí, např.: text/html.
- *Accept-Encoding* – kompresní algoritmy, kterým rozumí klient.
- *Accept-Language* – jazyk, kterému klient rozumí.
- *Authorization* – obsahuje informace pro autentizaci uživatele.
- *If-Modified-Since* – zabraňuje opakovanému zasílání, nezměněných dat. Klient zašle datum a čas, server zašle data pouze v případě, že došlo k jejich změně.
- *Referer* – URL ze kterého byl zaslán požadavek. Zpravidla je používán internetovými prohlížeči pro definici, ze které stránky požadavek pochází.
- *User-Agent* – identifikace klienta a jeho verze.

Hlavičky odpovědi

Přidávají dodatečné informace o serveru nebo požadované odpovědi.

- *Location* – obsahuje URL pro případné přesměrování. Používá se společně s odpovědí se stavovými kódy 201 a 3xx (viz 2.3.1).

[3] Hlavičky se zapisují ve tvaru – <název hlavičky>: <hodnota>.

- *Server* – nese informace o serveru, který vyřizuje požadavky.
- *WWW-Authenticate* – definuje autentizační metodu pro zajištění přístupu ke zdroji. Zasílá se s odpovědí se stavovým kódem 401 2.3.1).

Hlavičky popisující zdroje

Nesou informace o zdroji přenášeném v těle zprávy, např.: o kódování, velikosti apod.

- *Content-Encoding* – značí jaká kompresní metoda byla použita pro přenášená data.
- *Content-length* – udává velikost přenášených dat.
- *Content-type* – MIME typ přenášeným dat, např.: text/html, text/plain, application/json apod. [9].

2.5 Bezpečnost

Protokol HTTP poskytuje autentizační mechanismus pro zabezpečený přístup ke zdrojům. Jelikož je běžná HTTP komunikace zasílaná v nešifrované podobě, je nutné použití jeho zabezpečené varianty HTTPS.

2.5.1 HTTPS

HTTPS (Hypertext Transfer Protocol Secure) je modifikace protokolu HTTP, který pracuje nad protokolem transportní vrstvy TLS (Transport Layer Security). Jeho komunikace je šifrována symetrickou šifrou pomocí klíče, který vznikne při navazování spojení mezi klientem a serverem. Potencionální útočník, který odposlouchává komunikaci, pak není bez znalosti klíče schopen zprávy rozluštit a nemůže tak zneužít případně odposlechnuté citlivé informace. V dnešní době je tento protokol již dost rozšířený a hlavně doporučená varianta pro všechny typy webových serverů.

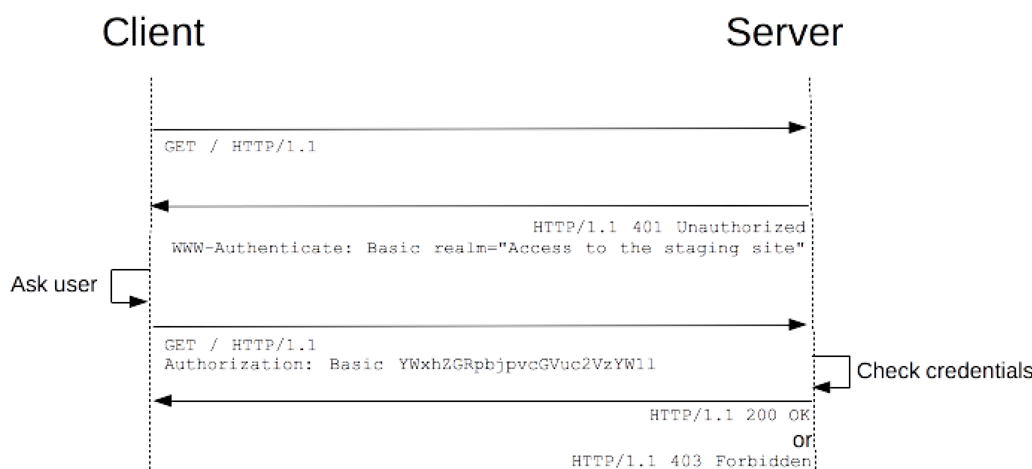
2.5.2 Autentizační mechanismus

Protokol HTTP poskytuje několik typů autentizace, jejichž data se přenášejí v hlavičce `Authorization` [10]. Hlavička obsahuje typ zabezpečení a klíč.

V případě, že se klient snaží přistoupit k zabezpečenému zdroji, server odešle odpověď se stavovým kódem 401 (Unathorized). Odpověď může obsahovat hlavičku `WWW-Authenticate`, která udává jaký typ zabezpečení server požaduje. V okamžiku, kdy klient obdrží odpověď, získá klíč (od uživatele např. zadáním jména a hesla, případně ho zná) a zopakuje původní požadavek, ale již s korektně vyplněnou hlavičkou `Authorization`. Server při přijetí požadavku ověří klíč a pokud je v pořádku, poskytne klientovi odpověď na jeho požadavek. V opačném

případě vrací odpověď se stavovým kódem 401 či 403 (Forbidden). Obrázek 1 postup ilustruje.

Jelikož je protokol HTTP bezstavový, je nutné s každým požadavkem na zabezpečený zdroj zasílat také autorizační hlavičku. Tento problém bývá v řadě aplikací řešen tzn. *session (relace, seance)*. Ta umožňuje v komunikaci mezi klientem a serverem držet po určitou dobu stav. To znamená, že se klient přihlásí pouze jednou, server vytvoří relaci a zašle klientovi informace k její identifikaci, ta se na klientské straně uloží a následně je v komunikaci připojována k požadavkům na server. Většinou se tato funkcionalita implementuje pomocí tzn. *HTTP cookies*.



Obrázek 1: HTTP autentizace, zdroj: [10]

2.5.3 Typy autentizace

Jak již bylo zmíněno, autentizační mechanismus protokolu HTTP podporuje několik typů autentizace. Říká se jim autentizační schémata a liší se zejména silou zabezpečení a případně také podporou na straně klienta. Pro zvýšení bezpečnosti je nutné používat společně s protokolem HTTPS. Stručně budou popsány vybrané: Basic a Bearer.

Basic

Jednoduchý typ autentizace. Spočívá v zaslání uživatelských údajů (typicky jméno a heslo) zakódovaných do *Base64*. Server si data dekóduje a má k dispozici původní údaje. Tento způsob se hodí spíše pro vzájemnou autentizaci aplikací, než uživatele oproti aplikaci. Příklad:

```
Authorization: Basic VXNlcjEyMzpwYXMkdzByZA==
```

Bearer – OAuth

Tento typ se používá pro generované tokeny (klíče). Ty mají většinou určitou časovou platnost, když vyprší, token již nelze znovu pro úspěšnou autentizaci použít. Standardem se zde stal de facto protokol *OAuth 2.0.* Ten může být použit jak pro autentizaci uživatele, tak pro autorizaci k provedení nějaké operace či přístupu k nějakému zdroji. Používá se hlavně pro delegovaní autorizace/autentizace na aplikaci třetí strany (tzn. ověřovací autority).

Způsob použití lze nejlépe představit na příkladu. Uvažujme jako autorizační autoritu Google. Uživatel je u něj zaregistrovaný. Při pokusu o přihlášení z klientské aplikace dojde k přesměrování na Google. Tam se uživatel přihlásí a klientovi je vrácen bezpečnostní token. Ten lze poté používat v autentizační hlavičce, přičemž server pro validaci tokenu využívá autorizační autoritu – v našem příkladu Google. Tento přístup se používá pro situaci, kdy klientská aplikace žádá o určitá data z Googlu (např. přístup k emailům). Hlavní výhoda toho řešení spočívá v tom, že klientská aplikace nezná přihlašovací údaje uživatele do Google účtu, ale má k dispozici jeho vybraná data.

2.6 Shrnutí

V kapitole byly představeny základní principy protokolu HTTP. Ten patří mezi bezesporu nejdůležitější a nejvyužívanější internetové protokoly. Jak se ukáže v dalších kapitolách hraje také zásadní roli ve spojitosti s webovými službami.

3 Webové služby

Tato kapitola pojednává o webových službách. Ty budou nejprve popsány obecně. Poté budou představeny základní principy a charakteristiky, konkrétně protokolu SOAP a architektonického stylu REST s využitím HTTP. Na závěr kapitoly budou oba přístupy porovnány.

3.1 Obecný popis

W3C[4] webovou službu definuje následovně [12]: „Webová služba je softwarový systém navržený pro interakci mezi dvěma stroji prostřednictvím počítačové sítě. Poskytuje rozhraní popsané strojově zpracovatelným formátem (WSDL). Ostatní systémy interagují s webovou službou definovaným rozhraním za použití SOAP zpráv, jejichž přenos je typicky zajišťován protokolem HTTP a XML serializací společně s dalšími webovými standardy."

Dále W3C rozděluje webové služby do dvou kategorií. První kategorie definuje služby dodržující architektonický styl REST a jsou označovány jako *REST-compliant*. Tyto služby pracují s webovými zdroji pomocí jednotné sady bezstavových operací. Druhá kategorie se nazývá *arbitrary Web services*. Ta označuje služby, které mohou poskytovat libovolnou množinu operací.

Neformálně a zjednodušeně by se tedy mohla webová služba popsat, jako mechanismus pro komunikaci a výměnu dat prostřednictvím počítačové sítě. Nezáleží však, jakými prostředky je služba implementována pokud dodržuje předepsané rozhraní pro komunikaci.

Jistě existují i další definice, které webové služby popisují rozdílněji či podrobněji, nicméně zmíněné jsou pro základní představu dostačující.

3.2 SOAP

SOAP (Simple Object Access Protocol) je protokol navržený pro výměnu strukturalizovaných dat v decentralizovaném a distribuovaném prostředí [13]. Při jeho návrhu byl kladen důraz na rozšířitelnost, neutralitu a nezávislost. Původní verzi protokolu v roce 1998 vytvořili za podpory firmy Mircrosoft – Dave Winer, Don Box, Bob Atkinson a Mohsen Al-Ghosein. V roce 2001 W3C vydalo verzi 1.1, ale nedostalo doporučení. První verzí která získala doporučení od W3C a je verze 1.2 vydaná roku 2003. Tato verze je již poslední verzí SOAP, i když byla v roce 2007 vydána tzv. „druhá edice" [14]. Specifikace protokolu byla udržována pracovní skupinou pod záštitou organizace W3C až do roku 2009, kdy byla činnost skupiny zastavena. Ačkoliv nebyla verze 1.1 podle W3C doporučena, stále se v praxi používá.

SOAP definuje podobu zpráv, které si v distribuovaném prostředí komunikující aplikace zasílají. Zprávy jsou zapisovány ve formátu XML (Extensible Markup Language). Pro přenos zpráv lze využít jakýkoliv protokol aplikační vrstvy

[4] Konsorcium, které spravuje a rozvíjí standardy pro World Wide Web [11].

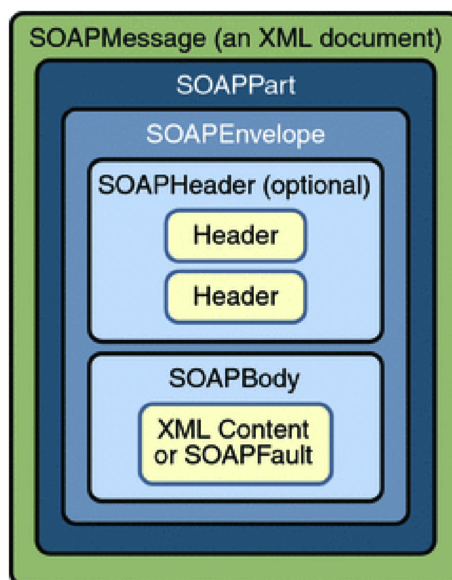
TCP/IP modelu. Prakticky se však používá protokol HTTP a to zejména díky jeho rozšířenosti a důvěryhodnosti, což umožňuje jednoduše procházet skrze firewally. Jako HTTP metody mohou být využity `GET` i `POST`, prakticky se ovšem používá výhradně metoda `POST`.

Přenos zpráv od počátečního uzlu (ten který požadavek zaslal), do uzlu koncového (ten který má být konečným příjemcem) může procházet přes 0-n uzlů, které figurují jako prostředníci. Každý uzel je jednoznačně identifikován pomocí URI (Uniform resource identifier). Speciálním identifikátorem je `http://schemas.xmlsoap.org/soap/actor/next`, který označuje další uzel v cestě, což je v případě počátečního uzlu první prostředník.

Zjednodušeně se dá SOAP popsat jako nástroj, který umožňuje vzdáleně (síťově) volat funkce pomocí zpráv. Zpráva může nést požadavek i odpověď. První druh zprávy obsahuje popis volané funkce, její parametry a druh návratové hodnoty (pokud funkce vrací nějaký výsledek).

3.2.1 Formát zpráv

XML formát pro zprávy byl vybrán pro svou rozšiřitelnost a jednoduchost. XML zprávy jsou textové. Jsou čitelné strojově i pro člověka a jejich výhodou je, že se dají validovat. SOAP zprávy se skládají z elementů definovaných v jmenném prostoru `http://schemas.xmlsoap.org/soap/envelope/`. Těmito elementy jsou envelope (obálka), header (hlavička), Body (tělo) a Fault (chyba) [15]. Obrázek 2 zobrazuje formát zprávy.



Obrázek 2: Formát SOAP zprávy, zdroj: [16]

Envelope (obálka)

Hlavní nebo-li kořenový element celé zprávy, každá SOAP zpráva ho musí obsahovat. Obálka může také obsahovat definici jmenných prostorů, které umožňují

odlišit elementy tvořící strukturu SOAP zprávy od elementů definující samotná přenášená data. Jak již název napovídá tento element, „obaluje“ ostatní elementy čímž také definuje začátek a konec zprávy.

Header (hlavička)

Nepovinný element zpráv. Pokud však existuje, musí být uveden jako první. Umožňuje přidávat dodatečné informace, podobně jako hlavičky u HTTP. Skládá se z jednotlivých záznamů, přičemž každý záznam nese data pro jiný účel – jako např.: pro autentizaci uživatele, ověření jeho práv nebo identifikaci probíhající transakce. Záznamy jsou zpracovávány uzly, přičemž každému záznamu lze pomocí atributu `actor` nastavit identifikátory uzlů, které mohou daný záznam zpracovávat. Uzel, který záznam zpracuje se musí postarat o to, aby nebyl již dále přeposílán a z hlavičky byl odstraněn. Pokud není atribut `actor` definován, je záznam určen ke zpracování na koncovém uzlu. Záznamům lze také definovat pravdivostní atribut `mustUnderstand`, který udává, zda-li uzel musí rozumět záznamu, který má zpracovávat. Pokud je tedy atribut nastaven na hodnotu „1“ a uzly záznamu nerozumí, je vyvolán chybový stav s informacemi o chybě a uzlu, který ji způsobil. Pokud atribut chybí, je situace stejná, jako kdyby měl nastanou hodnotu „0“.

Body (tělo)

Povinný element obsahuje informace, díky kterým může být na koncovém uzlu provedena příslušná operace. Popisuje jméno operace a její případné parametry. V případě, že jde o odpověď, tělo obsahuje informace o výsledku operace, která byla bez chyb zpracována.

Fault (chyba)

Tento element se vyskytuje pouze v odpovědích a signalizuje chybu, která nastala při zpracování požadavku. Standardně může obsahovat tyto pod-elementy:

- *Faultcode* – povinný, označení chyby pro programové zpracování. Musí být prezentován jednou z následujících možností:
  - *VersionMismatch* – označuje neplatný XML jmenný prostor pro `Envelope` element.
  - *MustUnderstand* – uzel, který měl zpracovávat `Header` záznam, s nastavený atributem *mustUnderstand* na „1“ mu neporozuměl.
  - *Client* – chyba na straně klienta. Zpráva byla špatně formátována nebo neobsahovala úplně informace pro úspěšně dokončení požadavku.
  - *Server* – chyba na straně serveru, který nebyl schopen zprávu zpracovat.
- *Faultstring* – povinný, popis chyby čitelný pro člověka.

- *Faultactor* – označuje uzel (pomocí jeho URI), který chybu způsobil. V případě, že chybu způsobil uzel, který není uzlem koncovým (tj. prostředníkem) je tento element povinným. V opačném případě povinný není.
- *Detail* – popis chyby, který by měl být vyplněn pokud se chyba týká elementu `Body`. Tzn. chyba vznikla v logice aplikace, např. aplikace nebyla schopna nalézt dotazovou entitu. Detail může obsahovat pod-elementy podobně jako element `Header`.

3.2.2 Styl zpráv

SOAP je možné používat se dvěma rozdílnými styly zpráv, které definují podobu dat v těle zprávy [17]:

- *RPC* – v přenášené zprávě je uvedeno jméno volané funkce a její parametry volání. Dnes prakticky nepoužíváno.
- *Document* – tělo zprávy obsahuje XML kód popsaný XML schématem.

3.2.3 Styl kódování

Určuje jakým způsobem budou kódována přenášená data do XML a naopak. Tomuto procesu se jinak říká serializace a deserializace. Existují dva způsoby [17]:

- *SOAP encoding* – použito SOAP kódování definované v rámci standardu. Formátování je podle datového modelu, který definuje skalární i komplexní datové typy.
- *Literal XML* – formát přenášených dat je dán schématem, například XSD (XML Schema Definition).

3.2.4 SOAP a HTTP

HTTP je nejběžněji používaným přenosovým protokolem pro SOAP. To zejména proto, že je široce využívaný a spolehlivý. I když má HTTP bohatou funkcionalitu, je protokolem SOAP využívána pouze její malá část. Z pohledu SOAPu lze o HTTP mluvit pouze jako o přenosové vrstvě. Zprávy bývají přenášeny pomocí metody `POST` a zpravidla obsahují pouze pár hlaviček.

Komunikace probíhá standardně dle režie HTTP. Požadavek i odpověď obsahuje hlavičky `Conntent-Length` a `Conntent-Type` s možnými hodnotami: `text/xml`, `application/xml` či `application/soap+xml` – záleží na verzi protokolu SOAP a jeho implementaci. Požadavek navíc obsahuje povinnou hlavičku `Host` a také `SOAPAction`. Tato hlavička je povinná a identifikuje SOAP požadavek – může být použita firewally pro filtraci požadavků. Obsahuje URI, která identifikuje službu, s kterou se má komunikovat. Pokud je prázdná, použije se adresa, na kterou je směřován požadavek. Ukázkový požadavek a odpověď zobrazuje zdrojový kód 3 respektive 4.

```
1  POST /StockQuote HTTP/1.1
2  Host: www.stockquoteserver.com
3  Content-Type: text/xml; charset="utf-8"
4  Content-Length: nnnn
5  SOAPAction: "Some-URI"
6
7  <SOAP-ENV:Envelope
8    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding"
9    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"/>
10
11   <SOAP-ENV:Body>
12     <m:GetLastTradePrice xmlns:m="Some-URI">
13       <symbol>DIS</symbol>
14     </m:GetLastTradePrice>
15   </SOAP-ENV:Body>
16 </SOAP-ENV:Envelope>
```

Zdrojový kód 3: SOAP požadavek pomocí HTTP, zdroj: [13]

3.2.5 Optimalizace zpráv – MTOM

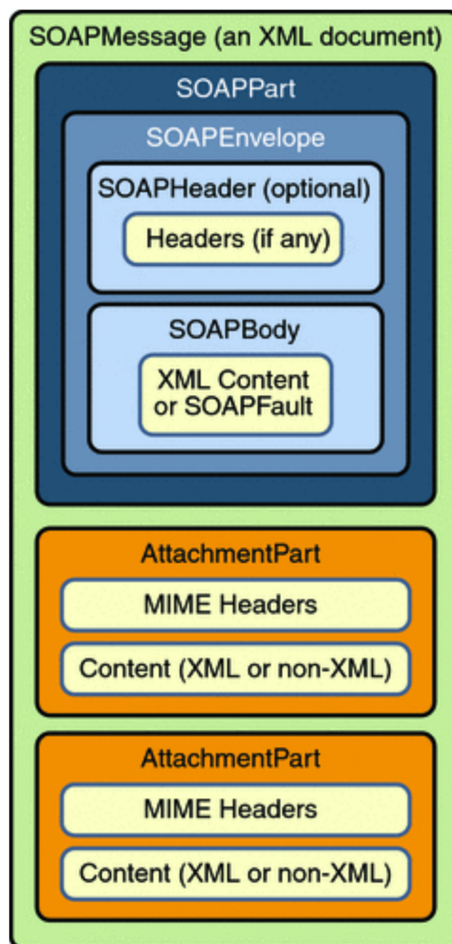
Jak již bylo uvedeno, SOAP využívá pro komunikaci XML formát. Ten je textový a neumožňuje tedy přenos souborů definovaných binárními daty. Tento problém lze řešit kódováním dat do *Base64* a následně běžným přenosem a dekódováním. Cenou za tohle řešení je nutnost dalšího výpočtu a taky fakt, že data po zakódování průměrně narostou o 33% [18]. To při nutnosti zasílání velkých souborů způsobuje poměrně velké zatížení jak na výpočet, tak i na samotný přenos.

SOAP tento problém řeší dalšími specifikacemi MTOM (Message Transmission Optimization Mechanism) a XOP (XML-binary Optimized Packaging) [19]. Data jsou v rámci HTTP přenášena jako tzv. `multipart`. Ty mohou obsahovat libovolný počet nezávislých částí, které mají rozdílný obsah a vlastní hlavičky (pro popis obsahu dané části např. `Content-Type` apod). První část bude obsahovat tradiční SOAP obálku s typem `application/xop+xml`. Ta bude obsahovat identifikátor odkazující na část s binárními daty. Obrázek 3 zobrazuje podobu zprávy složené z více částí, kde `AttachmentPart` značí části s binárními daty.

3.2.6 WSDL

WSDL (Web Services Description Language) je jazyk na bázi XML, navržený pro popis webových služeb. Myšlenka je taková, že webová služba běžící na serveru současně publikuje svůj popis ve WSDL formátu a případný klient má dokonalý přehled o API služby, jinými slovy zná funkcionalitu služby. Díky tomu klient dokáže za poměrně jednoduchého úsilí službu používat.

První WSDL verzí zveřejněnou konsorciem W3C byla verze 1.1, kterou vytvořili zaměstnanci firem Ariba, IBM a Microsoftu v roce 2001 [20]. Vycházeli



Obrázek 3: Formát multipart SOAP zprávy, zdroj: [16]

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"/>

  <SOAP-ENV:Body>
    <m:GetLastTradePriceResponse xmlns:m="Some-URI">
      <Price>34.5</Price>
    </m:GetLastTradePriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Zdrojový kód 4: SOAP odpověď pomocí HTTP, zdroj: [13]

při tom z proprietárních řešení zmíněných firem. Tato verze nedostala od W3C doporučení, avšak i navzdory tomu je, stejně jako SOAP 1.1, dodnes používána. V roce 2002 vydalo W3C pracovní návrh verze 1.2, který byl konsorciem doporučen k používání až v roce 2007 jako verze 2.0 [21].

Jedním z rozdílů mezi verzemi 1.1 a 2.0 je, že první verze je založena čistě na XML elementech a druhá verze definuje tzn. *Komponentový Model*, který umožňuje lepší abstrakci a popis služeb. Ten je nicméně založen také na XML a rozdíl není proto na první pohled výrazný. WSDL 2.0 lze využít i pro popis webových služeb založených na principu REST, o kterém pojednává kapitola 3.3. A to z toho důvodu, že na rozdíl od předchozích verzí umožňuje použití i jiných metod než pouze `GET` a `POST` protokolu HTTP.

I přes to, že verze 2.0 dostala od W3C doporučení k používání, tak si tato verze nevydobyla lepší postavení než-li používanější verze 1.1. V následující části bude proto pozornost zaměřena WSDL 1.1.

3.2.6.1 Popis

Stručně by se WSDL dalo popsat jako dokument popisující rozhraní služby, které definuje jména operací, jména a typy vstupních parametrů a návratové hodnoty. Dále poskytuje informace o tom, jakým způsobem službu volat (na jakém URL a portu je služba dostupná, jaký použít protokol apod.). WSDL má tedy obdobný účel jako hlavičkové soubory v programovacím jazyce „C“ nebo rozhraní (interface) v jazyce Java. Správně specifikované WSDL obsahuje veškeré informace nutné pro používání služby. Některé programovací jazyky disponují nástroji určenými na generování kódu pro volání dané služby z WSDL specifikace (lze i naopak generovat WSDL z programového kódu).

WSDL 1.1 definuje sedm elementů, kterými popisuje webové služby [20, 22].

Types (typy)

Obsahuje definice datových typů a elementů, které se používají jako parametry ve zprávách či jako vstupy a výstupy v popisu operací. Pro jejich definici se používá XLM Schema definition jazyk známý jako XSD.

Message (zpráva)

Zpráva se může skládat z jedné či více logických částí (part). Zpráva nese vstupní informace nebo výstupní. V praxi si ji lze představit jako vstupní parametry nebo výstupní hodnoty funkcí či metod u běžných programovacích jazyků.

Operation (operace)

Operaci definuje vstupní a výstupní zpráva pro danou funkcionalitu. Klient zasílá vstupní zprávy, server výstupní. Lze ji chápat jako funkci či metodu v běžných programovacích jazycích. Operace lze dělit na tyto typy:

- *One-way (jednosměrná)* – klient pošle zprávu, ale server neodpoví.
- *Request-response (požadavek-odpověď)* – klient pošle zprávu a server mu odpoví zprávou. Jedná se o nejběžnější typ zprávy.
- *Solicit-response (žádost-odpověď)* – v podstatě je to předchozí typ ale s obrácenými rolemi klienta a serveru. Server pošle zprávu a klient odpoví zprávou.
- *Notification (notifikace)* – server „notifikuje“ klienta zasláním výstupní zprávy.

Obrázek 4 znázorňuje typy operací, kde šipka značí zaslání zprávy a horní zpráva se zasílá jako první.

Port type (rozhraní)

Definuje množinu operací, které dohromady tvoří rozhraní služby.

Binding (vazba)

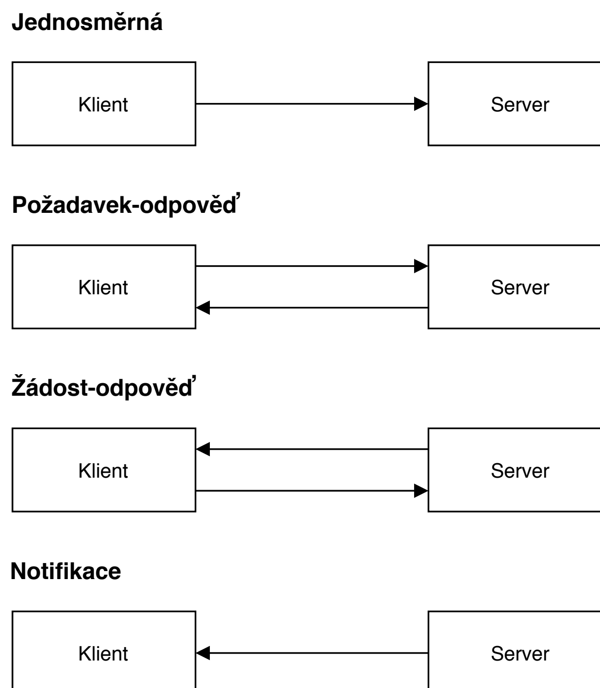
Definuje typ protokolu pro přenos SOAP zpráv (např.: HTTP, SMTP), styl vazeb (viz 3.2.6.2).

Port (výstupní bod)

Udává adresu, ze které je služba dostupná a kde ji lze používat. Typicky se jedná o URL adresu. Dalo by se říct, že vystavuje službu pro použití svému okolí.

Service (služba)

Je reprezentována množinou portů.



Obrázek 4: WSDL – typy operací

3.2.6.2 Styly vazeb

Styly vazeb (binding styles) ovlivňují výslednou podobu těla v SOAP zprávách [23, 24]. Dělí na dva typy: to na *RPC* (vzdálené volaní procedur nebo-li Remote procedure call) nebo *Document*. Ke každému stylu se navíc asociuje styl serializace – *Encoded* nebo *Literal*. Mohou vzniknout následující styly vazeb:

1. RPC/encoded.
2. RPC/literal.
3. Document/encoded (v praxi nepoužívaný).
4. Document/literal.
5. Document/literal wrapped (rozšíření předchozího).

Uvažujme, tuto metodu v programovacím jazyce Java:
`void myMethod(int x, float y);` jako funkcionalitu, kterou chceme skrze SOAP používat. Následující text stručně popíše rysy a výhody či nevýhody každého styl. Pro každý styl bude přiložen zdrojový kód demonstrující podobu WSDL dokumentu a SOAP zprávy.

RPC/encoded

Ukázka: zdrojový kód 5.

- WSDL je jednoduché jak jen to je možné.
- Jméno operace zapsané ve zprávě – umožňuje jednoduší identifikaci volané metody na serveru.
- Typy parametrů zapsány přímo ve zprávě – umožňuje polymorfismus metod, nicméně zvyšuje velikost přenášených dat.
- Zprávy nelze snadno validovat oproti XML schématu.

RPC/literal

Oproti předchozímu stylu se liší pouze tím, že zpráva nedefinuje typy parametrů. To znemožňuje polymorfismus, ale snižuje objem přenášených dat. Ukázka: zdrojový kód 6.

Document/literal

Ukázka: zdrojový kód 7.

- Jméno operace není zapsané ve zprávě, což může vést k problémům identifikace volané metody na serveru.
- Zpráva neobsahuje typovou definici parametrů.
- Lze validovat oproti XML schématu.

Document/literal wrapped

Stejný jako *Document/literal* styl s tím rozdílem, že zpráva obsahuje jméno operace. Nevýhodnou může být složitější definice WSDL. Tento styl je nejprosazovanější. Ukázka: zdrojový kód 8.

```
1  <!-- WSDL -->
2  <message name="myMethodRequest">
3    <part name="x" type="xsd:int"/>
4    <part name="y" type="xsd:float"/>
5  </message>
6  <message name="empty"/>
7  <portType name="PT">
8    <operation name="myMethod">
9      <input message="myMethodRequest"/>
10     <output message="empty"/>
11   </operation>
12 </portType>
13
14 <!-- SOAP zpráva -->
15 <soap:envelope>
16   <soap:body>
17     <myMethod>
18       <x xsi:type="xsd:int">5</x>
19       <y xsi:type="xsd:float">5.0</y>
20     </myMethod>
21   </soap:body>
22 </soap:envelope>
```

Zdrojový kód 5: WSDL – RPC/encoded styl, zdroj: [24]

```
1  <!-- WSDL -->
2  <message name="myMethodRequest">
3    <part name="x" type="xsd:int"/>
4    <part name="y" type="xsd:float"/>
5  </message>
6  <message name="empty"/>
7  <portType name="PT">
8    <operation name="myMethod">
9      <input message="myMethodRequest"/>
10     <output message="empty"/>
11   </operation>
12 </portType>
13
14 <!-- SOAP zpráva -->
15 <soap:envelope>
16   <soap:body>
17     <myMethod>
18       <x>5</x>
19       <y>5.0</y>
20     </myMethod>
21   </soap:body>
22 </soap:envelope>
```

Zdrojový kód 6: WSDL – RPC/literal styl, zdroj: [24]

```
<!-- WSDL -->
<types>
 <schema>
  <element name="xElement" type="xsd:int"/>
  <element name="yElement" type="xsd:float"/>
 </schema>
</types>
<message name="myMethodRequest">
 <part name="x" element="xElement"/>
 <part name="y" element="yElement"/>
</message>
<message name="empty"/>
<portType name="PT">
 <operation name="myMethod">
  <input message="myMethodRequest"/>
  <output message="empty"/>
 </operation>
</portType>

<!-- SOAP zpráva -->
<soap:envelope>
 <soap:body>
  <xElement>5</xElement>
  <yElement>5.0</yElement>
 </soap:body>
</soap:envelope>
```

Zdrojový kód 7: WSDL – Document/literal styl, zdroj: [24]

```
<!-- WSDL -->
<types>
  <schema>
    <element name="myMethod">
      <complexType>
        <sequence>
          <element name="x" type="xsd:int"/>
          <element name="y" type="xsd:float"/>
        </sequence>
      </complexType>
    </element>
    <element name="myMethodResponse">
      <complexType/>
    </element>
  </schema>
</types>
<message name="myMethodRequest">
  <part name="parameters" element="myMethod"/>
</message>
<message name="empty">
  <part name="parameters" element="myMethodResponse"/>
</message>
<portType name="PT">
  <operation name="myMethod">
    <input message="myMethodRequest"/>
    <output message="empty"/>
  </operation>
</portType>

<!-- SOAP zpráva -->
<soap:envelope>
  <soap:body>
    <myMethod>
      <x>5</x>
      <y>5.0</y>
    </myMethod>
  </soap:body>
</soap:envelope>
```

Zdrojový kód 8: WSDL – Document/literal wrapped styl, zdroj: [24]

3.2.7 Bezpečnost

Zabezpečení komunikace protokolu SOAP lze řešit na úrovní transportního protokolu. Tím bývá zpravidla HTTP a lze tedy použít jeho autentizační metody společně se šifrováním záměnnou HTTP za HTTPS (viz kapitola o bezpečnosti HTTP 2.5). Jinou možností je využití specifikace *WS-Security*.

WS-Security

Specifikace byla vytvořená společnostmi IBM, Miscrosoft a Verisng a stará se o ni skupina OASIS (Organization for the Advancement of Structured Information Standards) [25]. Využití specifikace nachází především, když je SOAP provozován na jiném než HTTP protokolu a nemůže tak využít jeho bezpečnostních výhod.

Díky specifikaci je možné zajistit integritu, šifrování či autentizaci zpráv. Také umožňuje přidat do SOAP zpráv bezpečnostní prvky (tzv. token, např. jméno a heslo, certifikát, nebo informace pro Kerberos server). Typ tokenů není přesně určen. WS-Security sama nezajišťuje ani neposkytuje bezpečnostní řešení, ale je základem pro jiné protokoly a technologie. Využívá například XML podpis nebo XML šifrování a určuje jak vložit hash zprávy nebo šifrovaná data do SOAP zprávy.

Rozšířením specifikace je WS-Trust zabývající se bezpečným vydáváním, obnovováním a validací bezpečnostních tokenů nebo WS-Federation pro delegaci autentizace na jiné systémy (autentizační servery) [26].

3.3 REST

REST (Representational state transfer) je architektonický styl navržený pro distribuovaná prostředí pracující s hypermédii jako například World Wide Web (WWW). Pojem hypermédium je rozšíření pojmu hypertext, který definuje nelineární textová data, která na sebe navzájem odkazují pomocí odkazů. Tedy hypermédium definuje v podstatě jakákoliv data (text, audio, video), která nelineárně odkazují na další Hypermédia, podobně jako u Hypertextu. REST definuje způsob práce se zdroji (hypermédii), což je rozdílný přístup oproti protokolu SOAP, který lze chápat jako vzdálené volání funkcím.

REST byl navrhnut v roce 2000 spoluautorem protokolu HTTP/1.1 Royem Thomasem Fieldingem v rámci jeho disertační práce „Architectural Styles and the Design of Network-based Software Architectures".

3.3.1 Omezení definující REST

REST je popsán jako hybridní architektonický styl, jenž je odvozený z několika síťových architektonických stylů přidáním 6 omezení (obsah kapitoly čerpán z [27, 28]):

1. Klient-server.

2. Stateless (bezstavovost).

3. Cacheable (možnost využití mezipaměti).
4. Uniform interface (jednotné rozhraní).
5. Layered system (vrstevný systém).
6. Code on demand (kód na vyžádání).

Tato omezení jsou postupně přidávána k tzv. "Null stylu"– architektonickému stylu s prázdnou množinou omezení, kde neexistují žádné omezující hranice mezi jednotlivými částmi celkového systému.

3.3.1.1 Klient-server

Omezení říká, že je nutné oddělit architekturu na dvě části dle principu klient-server. Server uchovává data, která poskytuje dle daného rozhraní klientovi. Ten se stará o jejich prezentaci v rámci uživatelského rozhraní.

To umožní lepší přenositelnost uživatelského rozhraní mezi platformami a škálovatelnost serverové části díky jejímu zjednodušení (nemusí řešit prezentaci dat uživateli).

3.3.1.2 Stateless (bezstavovost)

Omezuje komunikaci mezi klientem a serverem. Komunikace musí být bezstavová. To znamená, že každý klientský požadavek musí obsahovat všechna data nutná pro pochopení a provedení požadavku na serveru. Jinými slovy – nesmí být využito žádného stavu na serveru. Případný stav musí být udržován v rámci klienta.

Absence držení stavu mezi požadavky na serveru zlepšuje systému jeho škálovatelnost, spolehlivost a transparentnost:

- *Škálovatelnost* – snadnější implementace a lepší uvolňování zdrojů.
- *Spolehlivost* – jednodušší zotavení z případné chyby.
- *Transparentnost* – dopad na systém je ovlivněn pouze samotným požadavkem.

Bohužel omezení nepřináší pouze zmíněné výhody, ale také nezanedbatelnou nevýhodu. Tou je snížená propustnost systému vzhledem k nutnosti zasílání redundantních dat, která by mohla být již uložena na serveru.

3.3.1.3 Cacheable (možnost využití mezipaměti)

Tohle omezení má zefektivnit komunikaci systému. Toho je docíleno označením dat, která jsou poskytována serverem jako cacheable nebo non-cacheable. Data, která jsou serverem vrácena jako cacheable si klient může uložit do mezipaměti

a v případě stejného požadavku je použít bez nutnosti další komunikace se serverem. To se hodí zejména pro statická data nebo data, která se v průběhu času příliš nemění.

Zjevnou výhodou omezení je (výrazné) zvýšení výkonnosti systému. Naproti tomu omezení bohužel snižuje spolehlivost systému. To proto, že potencionálně může nastat situace, kdy se data v mezipaměti liší od dat, které by aktuálně poskytoval server.

3.3.1.4 Uniform interface (jednotné rozhraní)

Fielding označuje tohle omezení za klíčové. Odlišuje REST jako architektonický styl od ostatních síťových stylů tím, že klade důraz na jednotnost rozhraní. Jelikož mezi sebou komunikují různé části systému (klient, server, prostředníci), je zásadní udržet vzájemnou komunikaci jednotnou a konzistentní. *Uniform interface* obsahuje následující dílčí omezení:

- *Identification of resources (identifikace zdroje)* – zdroj musí mít jednoznačný identifikátor. Prostřednictvím identifikátoru se přistupuje ke zdroji v rámci klientských požadavků. K tomu může být použito například URI (Uniform Resource Identifier).

- *Manipulation of resources through representations (manipulace se zdrojem prostřednictvím jeho reprezentace)* – práce se zdrojem neprobíhá přímo, nýbrž prostřednictvím jeho reprezentace. Ta může být různá pro stejný identifikátor zdroje. Díky tomu může klient pracovat s reprezentací, které rozumí a modifikovat či smazat zdroj po jeho získání.

- *Selfdescriptive messages – (samopopisující zprávy)* – každá zpráva by měla obsahovat dostatek informací, aby mohla být úspěšně a jednoduše zpracována. Jde hlavně o formát reprezentace zdroje, kterým může být MIME typ.

- *Hypermedia as the engine of application state (hypermédia jako stav aplikace)* – zdroje jsou brány jako hypermédia. Server při poskytování zdroje klientovi zároveň poskytne identifikátory souvisejících zdrojů. Jde o důležitou součást aktuálního stavu zdroje.

Omezení má pozitivní dopad na transparentnost systému. Usnadňuje implementaci spojenou se vzájemnou komunikací různých částí systému. Nevýhodou může být jeho snížená efektivita komunikace – standardizovaná forma dat, nikoliv přesně vyhovující klientské části.

3.3.1.5 Layered system (vrstevný systém)

Vrstevný systém znamená rozdělení systému do několika vrstev. Samotné vrstvy mohou komunikovat pouze s přímo sousedícími vrstvami (nižší, vyšší). Omezení dává možnost přidávat k již zmíněnému principu klient-server mezivrstvy

(prostředníky). Výhodou toho řešení je rozdělení systému na více menších jednoduchých částí s jasně daným účelem. Např. pro zvýšení propustnosti systému – load balancing[5], využití mezipaměti či dodržování bezpečnostních politik.

Tento přístup také přináší nevýhodu v podobě vyšší zátěže serveru, způsobenou dodatečnou režií mezi vrstvami. Nevýhodu lze částečně vyřešit mezivrstvou s již zmíněnou mezipamětí, což může mít naopak na výkonnost velmi pozitivní účinek.

3.3.1.6 Code on demand (kód na vyžádání)

Poslední omezení a také jediné volitelné říká, že server může poskytovat spustitelný kód, který si klient vyžádá a na své straně vykoná. Kódem může být například Java applet, případně nějaký druh skriptu v libovolném skriptovacím jazyce.

Výhodou použití tohoto omezení je přesunutí zátěže ze serverové části na klientskou a tím zvyšuje výkonnost systému. Nevýhodou je, že podstatně snižuje transparentnost systému, což je také důvod, proč je omezení volitelné. Mělo by se použít pouze po pečlivém uvážení zda-li má opravdu smysl.

3.3.2 REST společně s HTTP

REST jako architektonický styl je definován obecně. Nenařizuje, jakým způsobem a pomocí jakých technologií má být implementován. Nicméně je vidět určitá podobnost přístupu REST a HTTP protokolu. Jde například o bezstavovost nebo možnost využití mezipaměti (v HTTP formou hlaviček), což lze zdůvodnit tím, že autor REST architektury souběžně spolupracoval na vývoji protokolu HTTP/1.1.

S přispěním zmíněné podobnosti a taky faktu, že HTTP patří k nejvíce rozšířeným protokolům v rámci internetu, je HTTP ideální technologií pro implementaci REST architektury. Ta může být implementována např. i pomocí protokolu SMTP, nicméně v praxi je de facto standart implementovat REST „na“ HTTP. Implementace, které splňují požadavky REST architektury jsou nazývány RESTful. Ty se dají dělit do 4 úrovní (viz obrázek 5) podle toho, do jaké míry splňují REST [29, 30]. Tyto úrovně definoval Leonard Richardson. Jsou obecně respektovány a často se pomocí nich vysvětluje princip RESTu.

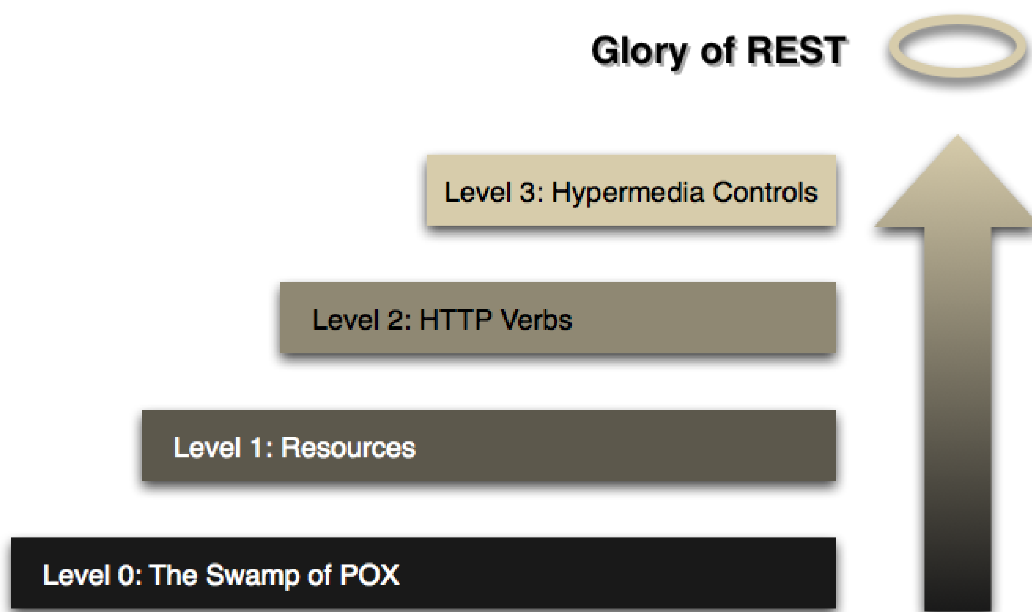
Nultá úroveň

Jako přenosový protokol definuje HTTP. Implementace na této úrovni používají jednu URL adresu i HTTP metodu (nejčastěji POST). Do této úrovně by se mohl zařadit i protokol SOAP.

První úroveň

Oproti nulté úrovni, kde je pouze jediná hlavní URL, tato úroveň definuje pro

[5] Rozdělení zátěže na více stejných serverů, dle daných pravidel (horizontální škálování).



Obrázek 5: Úrovně RESTful implementací, zdroj: [29]

každý zdroj vlastní URL, čímž jednotlivé zdroje jednoznačně identifikuje dle principu REST (viz kapitola 3.3.1.4), HTTP metoda zůstává jediná.

To si lze představit na příkladě s jednoduchou blogovací aplikací, kde zdroje jsou články, ke kterým lze navíc přidat komentář. Každý článek i komentář má vlastní číselné ID (běžně používaný způsob pro rozlišení zdrojů stejného typu). Pomocí REST API by k těmto zdrojům šlo přistupovat tímto způsobem:

- */clanky* – přístup ke všem článkům.
- */clanky/1* – přístup k článku s ID 1.
- */clanky/1/komentare* – přístup ke všem komentářům u článku s ID 1.
- */clanky/1/komentare/3* – přístup ke komentáři s ID 3 u článku s ID 1.

Jelikož HTTP metoda se v této úrovni používá pouze jediná, způsob práce se zdrojem by mohla obsahovat třeba samotná URL. Tedy např. pro přidání nového článku: */clanky/pridat* apod.

Druhá úroveň

Obohacuje předchozí úroveň o používání více HTTP metod místo jedné. Každá z metod má svůj účel, pro který by měla být využita. Metody bývají většinou čtyři a jejich účel je dohromady znám pod zkratkou CRUD – vytvoření (create), čtení (read), úprava (update) a smazání (delete) zdroje. Často se využívá také pátá metoda, která se používá také pro úpravu zdroje. Metody jsou následující:

- *POST* – vytvoří nový zdroj.

- *GET* – získá, zdroj.
- *PUT* – částečná úprava nebo úplné nahrazení zdroje novým.
- *DELETE* – smaže zdroj.
- *PATCH* – částečná úprava zdroje. Pokud je použita, tak se metoda `PUT` používá pouze pro úplné nahrazení zdroje.

Server odpovídá na volání stavovými kódy (viz kapitola o HTTP stavových kódech 2.3.1), Vstupní a výstupní data jsou přenášena v tělech zpráv, jak je běžné pro protokol HTTP.

Třetí úroveň

Třetí a poslední úroveň má vlastnosti předchozí úrovně a navíc přidává princip z REST architektury – *Hypermedia as the engine of application state*, známý jako akronym HATEOAS. To znamená, že uživateli by měla teoreticky stačit pouze základní (kořenová) URL adresa zdroje, ze kterého by byl schopen získat odkazy na zdroje další, které by fungovaly obdobně. Každá URL adresa by tak poskytovala nejen operace se zdrojem, ale i informace jaké další operace (a kde) lze dále provádět.

Jako příklad lze opět použít aplikaci pro blogovaní, kde je zdroj „články“ a pokud chceme přidat nový článek použijeme k tomu metodu `POST`. Jakmile se článek na serveru vytvoří, je zaslána odpověď obsahující hlavičku `Location` s URL adresou pro přístup k nově vytvořenému zdroji (článku), tedy například `/clanky/1`.

Navzdory tomu, že definice RESTu velí pracovat s REST API dynamicky pomocí odkazů, tak se princip HATEOAS v praxi moc nevyužívá a dává se přednost vývoji podle jasně specifikované dokumentaci.

3.3.3 Bezpečnost

REST, stejně jako protokol SOAP, není vázán na transportní vrstvu a nemusí tedy nutně používat HTTP. Žádné jiné protokoly nejsou používány, a proto lze jako základní bezpečnostní prvky pro REST používat stejné, jako v kapitole o HTTP bezpečnosti (viz 2.5). Byla by škoda alespoň stručně nezmínit JWT, které se ve světe RESTových služeb těší velké oblibě.

3.3.3.1 JWT

JWT (JSON Web Tokens) je standard, který umožňuje důvěryhodný přenos informací definovaných jako JSON objekt [31]. Důvěryhodnost je zaručena podpisem. Ten může být realizován kontrolním součtem s tajemstvím pomocí HMAC (Keyed-hash Message Authentication Code), případně podepsáním pomocí veřejného/privátního klíče, např. algoritmem RSA. Ten ovšem zaručuje pouze integritu a důvěryhodnost přenášených dat. Jinými slovy prokazuje, že při přenosu

nedošlo k poškození nebo změně dat. Data jsou stále přenášena v otevřeném formátu a proto je nutné použití protokolu HTTPS případně data šifrovat za pomoci např. algoritmu RSA.

JWT se díky výše zmíněným vlastnostem hodí využít pro autentizaci/autorizaci, případně pouze pro výměnu dat.

Autentizace

Autentizace s využitím JWT je snadná. Klient zašle požadavek pro přihlášení s uživatelovými přihlašovacími údaji. Pokud jsou správné, server vystaví JWT token a zašle ho zpět klientovi. Token je pak připojován k požadavkům v autorizační hlavičce protokolu HTTP: `Authorization: Bearer <token>`.

Výhodou řešení autentizace pomocí JWT je, že umožňuje absolutní bezstavovost, protože všechny důležité informace mohou být přímo součástí tokenu, jako například: uživatelské jméno či uživatelská role. Na serveru pak nemusí být držen stav, protože si server požadované informace může dekódovat z tokenu.

Na závěr je dobré zmínit, že JWT token v dosti případech vydává a ověřuje samostatný autorizační server. Ten je oddělený od serveru který vystavuje API ke konzumaci. Princi odděleného autorizačního serveru je výhodný, jelikož zvyšuje bezpečnost. JWT tokeny se většinou používají i pro OAuth princip autentizace.

3.4 Srovnání SOAP a REST

Nyní, po popisu charakteristik a principů SOAP a REST je možné provést jejich srovnání v teoretické rovině [32, 33].

Hlavní rozdíl je v přístupu obou technologií. SOAP jde cestou služeb poskytujících operace. Ta může být pro řadu vývojářů intuitivnější, něž přístup založený na zdrojích (hypermédiích), který využívá REST. V případě SOAPu pracují jak jsou zvyklí z moderních programovacích jazyků a službu navrhnou jako například třídu v objektově orientovaném programování. Návrh API, postavený na REST, vyžaduje daleko větší pečlivost a zvažování, zda-li jsou HTTP metody či stavové kódy pro danou funkcionalitu navrženy správně. Tento problém však do určité míry řeší princip CRUD, který se běžně využívá pro návrh a vývoj aplikací využívající REST.

Definice rozhraní webových služeb, založených na SOAP, je flexibilní – operaci lze libovolně pojmenovat dle jejího účelu. Oproti tomu je REST značně omezený, jelikož pro rozlišení účelu dané operace na zdroji používá HTTP metody. To se může zdát jako nevýhoda RESTu, nicméně tím REST efektivně využívá rysy protokolu HTTP, což vede k jeho vyšší efektivitě. SOAP používá pouze metodu `POST` a veškerá data nutná pro komunikaci vkládá do XML obálek. To s sebou nese celou řadu nevýhod. XML formát je výhodný díky své popisnosti zejména pro konfigurační soubory, pro přenos dat po sítí už tolik ne. Režie způsobená přenosem objemnějších dat snižuje celkovou odezvu systému. Další nevýhoda je nutnost obálku zpracovat, tedy zjednodušeně řečeno, zjistit jakou operaci a s jakými daty představuje. To by v případě jednoduchých aplikací nemusel být až

takový problém. Ten by však mohl nastat v případě potřeby implementace systému s vysokým zatížením, kde se každá výkonnostní nevýhoda dříve či později projeví. RESTové volání navíc umožňuje v některých situacích využít mezipaměti, což může rapidně zvednout výkonnost. Tu lze také zvýšit tzn. škálovatelností, kdy je serverová část spuštěna ve více instancích, které se o obsloužení požadavků definovaným způsobem dělí.

V oblasti výkonnosti tedy REST jednoznačně vítězí i díky tomu, že pro komunikaci nemusí zpracovávat žádná data. Navíc není závislý pouze na jednom formátu, ale data může přenášet i ve formátech dalších, jako například HTML, PNG, JSON či YAML. To může zjednodušit aplikace, které tyto formáty vyžadují přenášet. V rámci komunikace je využití proxy serverů komplikovanější pro SOAP, protože každý proxy server musí formátu jeho zprávy rozumět. V případě RESTu tento problém nehrozí, ten pouze využívá funkcionality HTTP a tomu proxy server musí rozumět.

Bezpečností zase vyčnívá SOAP. Ten poskytuje opravdu rozsáhlé možnosti zabezpečení komunikace od šifrování přes kontrolu integrity dat, až například po transakční zpracování dat.

Odlišností či výhod/nevýhod mezi SOAPem a RESTem je určitě více. Zmíněné jsou však považovány za zásadní a nejvíce důležité. Celé srovnání shrnuje následující tabulka 2.

3.5 Shrnutí

V této kapitole byly popsány základní rysy a srovnání webových služeb postavených na protokolu SOAP a architektonickém stylu REST v teoretické rovinně. Poskytuje tak teoretický základ pro další kapitoly práce.

Tabulka 2: Stručné srovnání SOAP – REST, zdroj: [32, 33]

	SOAP	**REST**
Typ	Standardizovaný protokol.	Architektonický styl.
Komunikace	Prostřednictvím služeb s definovanými operacemi.	Prostřednictvím tzv. zdrojů.
Stavovost	Podporuje stavovou i bezstavovou komunikaci.	Komunikace je z principu bezstavová.
Mezipaměť	Nemůže používat mezipaměť (cache).	Může používat mezipaměť.
Bezpečnost	Definuje vlastní standardy, např. WS-security.	Neřeší, ponechává na transportní vrstvě (HTTPS).
Formát zpráv	Pouze XML.	Libovolný – HTML, XML, JSON, YAML apod.
Výkon	Vyšší nároky na objem přenášených dat i výpočtu (zpracování XML).	Nižší nároky na objem přenášených dat. Nemusí zpracovávat přenášená data pro zjištění účelu zprávy.
Výhody	Standardizovanost, rozsáhlé možnosti zabezpečení, rozšířitelnost.	Škálovatelnost, výkonnost, jednoduchost a flexibilita.
Nevýhody	Horší výkon, složitost (velké množství standardů – chaotičnost), menší flexibilita.	Bezpečnost, do určité míry absence standardů.

4 Technologie a nástroje pro webové služby

V této kapitole budou představeny ekosystémy tvořené technologiemi a nástroji určenými pro zjednodušení či zefektivnění vývoje webových služeb. Oba ekosystém budou popsány a na závěr kapitoly dojde k jejich stručnému srovnání.

4.1 SOAP

Hlavní doménou ekosystému protokolu SOAP je velké množství specifikací (standardů) zabývající se řešením problémů nastávajících při vývoji pomocí protokolu. Většina specifikací byla vytvořena firmami (nebo za jejich účasti) IBM či Microsoft a o standardizaci se stará OASIS. Některé specifikace již byly probrány či zmíněny v kapitole 3.2 pojednávající o samotném protokolu SOAP. Jako například WSDL, MTOM či specifikace týkající se bezpečnosti.

Kromě popisu vybraných specifikací dojde i na rozšířený testovací nástroj SoapUI.

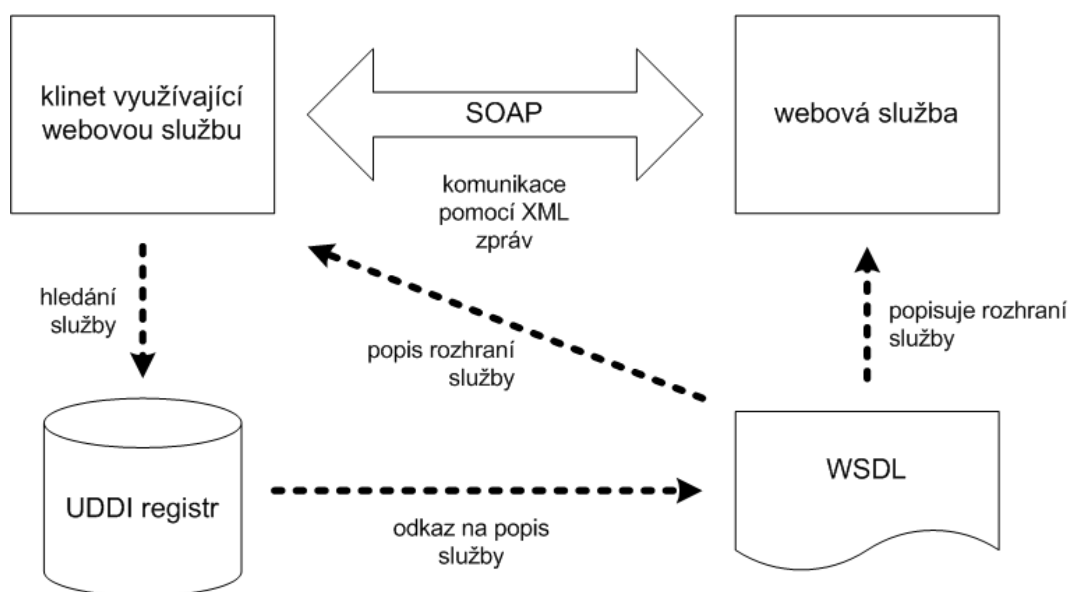
4.1.1 UDDI

UDDI (Universal Description, Discovery and Integration) je specifikace pro vyhledávání a registraci webových služeb, založená na XML [34]. Slouží jako registr webových služeb, které jsou popsány pomocí WSDL. Záznamy jsou reprezentovány v XML. UDDI bývá implementována jako webová služba a pro komunikaci využívá protokol SOAP.

UDDI register obsahuje následující entity:

- *Podnikatelské entity (business entity)* – obsahuje informace o daném podnikatelském subjektu jako je název, popis činnosti, kontaktní údaje nebo geografickou polohu.
- *Služby (business service)* – popisuje službu, obsahuje seznam šablon vazeb. Každá podnikatelská entita v registru má uloženy služby, které poskytuje.
- *Šablony vazeb (binding template)* – definuje způsob komunikace se službou. Obsahuje odkaz na WSDL soubor a službu, kterou implementuje.
- *Typy služeb (service type)* – abstraktní definice služby. Obdobná jako rozhraní v programovacím jazyce. Podnikatelské entity mohou poskytovat služby se stejným rozhraním (typem služeb).

Užití UDDI probíhá tak, že vývojář či klientská aplikace prohledá registr nabízených služeb, najde si ty, které potřebuje využít a získá pro ně WSDL soubor. Poté může služby začít používat. Vzájemný vztah SOAP, WSDL a UDDI zobrazuje obrázek 6.



Obrázek 6: Vzájemný vztah SOAP-WSDL-UDDI zdroj: [34]

4.1.2 WS specifikace

Specifikace rozšiřující funkcionalitu SOAPu jsou označovány jako WS-* (mimo pár případů jako např.: WSDL). V kapitole budou popsány vybrané partie, které specifikace pokrývají, mimo bezpečnost popsanou v rámci kapitoly 3.2.7 (obsah čerpán mj. z [26]).

Použitelnost rozšiřujících specifikací je úměrná podpoře programovacími jazyky či na nich postavenými technologiemi (aplikačními servery, knihovnami). Až na pár výjimek (zejména co se týče bezpečnosti) jsou specifikace podporovány hlavně v rámci platforem .NET (C#) a Java. To je dáno jejich robustností a tedy i typickému použití při vývoji rozsáhlejších aplikací.

Adresování

WS-Addressing poskytuje rozšířené možnosti směrování komunikace [35]. Tu SOAP sám o sobě ponechává na transportní vrstvě kterou používá. Zpráva tak bývá odeslána jednomu příjemci a odesílateli je vrácen výsledek. Tento způsob komunikace je pro většinu služeb dostačující. Pokud ne, pomocí WS-Addressing lze směrování každé konkrétní zprávy flexibilně nastavit – více příjemců, kam zaslat výsledek či chybu apod. To může být žádoucí například při asynchronní komunikaci, kdy se na výsledek operace nečeká a ten je po vykonání zaslán na definovanou adresu.

Spolehlivost komunikace

K zaručení spolehlivé komunikace slouží specifikace WS-ReliableMessaging [36]. Zasílání zpráv probíhá skrze prostředníky, kteří potvrzují příjem. Poskytuje čtyři typy možných záruk:

- *ExactlyOnce* – zpráva bude doručena přesně jednou, bez duplicit, pokud doručena být nemůže, vznikne chyba.
- *AtMostOnce* – zpráva bude doručena maximálně jednou, bez duplicit, avšak doručena být nemusí.
- *AtLeastOnce* – minimálně jednou bude zpráva doručena, pokud doručena být nemůže, vznikne chyba.
- *InOrder* – příjemce přijme zprávy v pořadí v jakém byly zaslány odesílatelem. Tato záruka může byt spojena s některou z předešlých záruk.

Popis vlastností služeb

Sada specifikací jejichž název obsahuje slovo *Policy* (např. WS-Policy) má za cíl popisovat vlastnosti (omezení, požadavky) služeb. Pro definici vlastností je využit jazyk založený na XML. Vlastnost může mít určitý rozsah působnosti. Tím může být celá služba, vybrané operace nebo samotné zprávy apod. Definice může být součástí WSDL či UUID. Nejčastěji se využívá k určení bezpečnostních požadavků (nutnost šifrování, autentizačních tokenů atd.), využívání jiných specifikací (např. WS-ReliableMessaging) či libovolných aplikačních vlastností.

Události/Notifikace

Pro distribuci událostí (notifikací) existuje více specifikací s podobnou funkcionalitou (WS-Eventing, WS-Notification). Obsahují principy známé z návrhového vzoru Publish–subscribe. Umožňuje webovým službám „publikovat“ zprávy pod daným tématem, které mohou jiné služby „odebírat“. Odběr zpráv může být službou po čase zrušen. Bývá definována maximální doba odebírání (předcházení zahlcení).

Transakční zpracování

Také v rámci webových služeb lze využít transakčního zpracování dat. To umožňuje provést sérii operací dohromady, jako jednu nedělitelnou, která splňuje vlastnosti ACID (atomicita, konzistence, izolovanost, trvalost). Tento princip je známý zejména z oblasti databázových systémů. Pokud během zpracování nastane chyba, je obnoven stav známý před započetím zpracování. To však může být někdy velmi obtížné, a proto lze pro tyto případy definovat kompenzační mechanismy. Transakční zpracovaní popisuje WS-Transaction [37].

4.1.3 SoapUI

SoapUI je komplexní nástroj, umožňující jednoduše testovat API. [38]. Je k dispozici pro testování API založených na SOAP, ale i REST. Je naprogramován v jazyce Java, což umožňuje přenositelnost mezi různými platformami. Existuje open source, ale i placená verze s rozšířenou funkcionalitou. SoapUI je program vhodný pro vývojáře díky podpoře funkčního testování a zejména také mockování

API. Dále taky podporuje zátěžové a bezpečnostní testování. Testy mohou být programovány ve skriptovacím jazyce Groovy a mohou být automaticky spouštěny přes příkazovou řádku, což může být vhodné například při využití v rámci *Continuous Integration*[6]. Příjemnou funkcí je také například možnost validace WSDL souborů, což se jistě hodí při vývoji, kde se nejprve navrhuje API a až poté se implementuje.

Výhodou tohoto nástroje je jeho velká tradice a počet uživatelů, což přispívá k vyladěnosti aplikace a pozitivně se to projevuje také na kvalitní a obsáhlé dokumentaci.

4.2 REST

Ekosystém okolo RESTu tvoří hlavně modelovací jazyky pro formální definici API. Vznikla jich celá řada včetně nástrojů, které s nimi umí různými způsoby pracovat (editace, generování kódu, mockování API). Zřídka mohou existovat specifikace založené na stylu REST jako například Odata. Samozřejmostí jsou pak nástroje pro testování.

4.2.1 Jazyky pro modelování API

OpenAPI

OpenAPI specifikace (dříve známa jako Swager specifikace) je navržena tak, aby definovanému API porozuměl člověk i počítač [39]. Jazyk, který OpenAPI specifikuje, může být zapsán ve formátu YAML či JSON. Umí oddělit prvky API do komponent, na které lze pak odkazovat (definice data objektů, odpovědí, parametrů požadavků apod.). V současné době je nejpoužívanější pro popis REST služeb, což příznivě ovlivňuje počet nástrojů, které s ním umí pracovat.

Zdrojový kód 9 ukazuje definici REST API ve formátu YAML. API má definovaný jeden zdroj identifikovaný relativní URL `/users`, který podporuje jedinou metodu a to `GET`, pro získání všech uživatelů.

API Blueprint

API Blueprint vychází z jazyka Markdown [7] [40]. Má obdobné vyjadřovací schopnosti jako OpenAPI. Také dokáže dělit prvky API do znovu použitelných komponent, nicméně jeho syntaxe je jednodušší. Na stránkách projektu lze najít poměrně velké množství podpůrných nástrojů pro práci s tímto jazykem – od editorů až po nástroje pro testování.

RAML

Dalším modelovacím jazykem je RAML (RESTful API Modeling Language) [41]. Je zapisován ve formátu YAML a má dost podobnou syntaxi jako OpenApi. Na

[6]Metodika pro zefektivnění a zkvalitnění procesu vývoje softwaru.
[7]Značkovací jazyk pro jednoduché formátování textu, vyniká zejména snadnou syntaxí.

```
openapi: 3.0.0
info:
 title: User endpoint.
 version: ’1.0’

paths:
  /users:
    get:
      responses:
        200:
          description: User was succesfully gets from server.
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    name:
                      type: string
                      example: "Ivos Apolenar"
                    age:
                      type: integer
                      example: 27
```

Zdrojový kód 9: OpenAPI – definice REST API ve formátu YAML

stránkách projektu lze najít poměrně velké množství podpůrných nástrojů pro práci tímto jazykem. Od editorů až po nástroje pro testování.

```
# User endpoint.
## Users collection [/users]
### List all users [GET]

+ Response 200 (application/json)

[
{
  "name": "Ivos Apolenar",
  "age": 27
}
]
```

Zdrojový kód 10: API Blueprint – ukázka definice REST API

```
#RAML 1.0
title: User Endpoint

/users:
  get:
    responses:
      200:
        body:
          application/json:
            example: |
              [
                {
                "name": "Ivos Apolenar",
                "age": 27
                }
              ]
```

Zdrojový kód 11: RAML – ukázka definice REST API

4.2.2 Nástroje pro práci s API

Swagger

Swagger je kolekce nástrojů pro vývojáře v týmu, ale i jednotlivce pracující s OpenAPI specifikací [42]. Poskytuje nástroje pokrývající celý životní cyklus vývoje REST API – od návrhu dokumentace, testování až po samotný proces nasazení. Obsahuje open source i profesionální licencované nástroje. Následující výčet je stručně popisuje:

- *Open source*
    - *Swagger Codegen* – umožňuje pro podporované programovací jazyky generovat serverové i klientské aplikace podle OpenAPI specifikace.
    - *Swagger Editor* – WYSIWYG[8] editor určený pro vývoj API designu a jeho dokumentace v rámci OpenAPI specifikace. Průběžně validuje upravovaný dokument a na případné chyby tak ihned upozorňuje. Je poskytován jako webová aplikace, kterou si může uživatel stáhnout a používat ji lokálně ve webovém prohlížeči.
    - *Swagger UI* – interaktivní prohlížeč definovaného API.
- *Licencované*
    - *Swagger Inspector* – REST API klient pro účely funkčního testování, umožňuje nahrát definici API ze souboru, případně ze zadané URL a tvořit testovací scénáře.
    - *SwaggerHub* – integrace všech předešlých nástrojů do jednoho. Určen především pro týmové využití. Pro definovaná API, lze vytvořit mock server.

Apiary

Nástroj pro zjednodušení vývoje API. Původně ho vyvinul český start-up, který posléze odkoupil softwarový gigant Oracle [43]. Funkcionalitou je podobný jako zmíněný SwaggerHub. Pro definici API nepoužívá OpenAPI, nýbrž BluePrint. Podporuje vytváření mock serverů, na kterých lze podrobně nahlížet do příchozích požadavků a odchozích odpovědí.

4.2.3 OData

OData (Open Data Protocol) je standard definující generický přístup pro vytváření a používání REST API [44]. Byl vytvořen firmou Microsoft a je standardizován skupinou OASIS. Odata odstiňuje uživatele od režijních věcí, spojených

[8] Anglický akronym pro „What you see is what you get“, tedy „Co vidíš, to dostaneš“. Označuje způsob editace, při které uživatel rovnou vidí podobu upravovaného dokument.

při používání běžného REST přístupu a umožňuje soustředění na danou logiku aplikace.

Využívá běžné operace pro práci se zdroji (entitami) dle principu REST pro přidání, úpravu, mazání a pokročilejší dotazování v rámci URL (filtrace, stránkování, řazení apod.). Navíc přináší možnost užití obecných či na entity vázaných procedur rozšiřující funkcionalitu. Struktura entit je popsána metadaty ve formě XML či JSON schématu, což umožňuje generické zpracování např. pomocí dotazovacích proxy apod.

OData svým přístupem a funkcionalitou projevuje podobné rysy jako API pro práci s databází. Existují implementace, které se dají propojit přímo či nepřímo[9] s databází a zpřístupnit ji tak skrze REST. Podpora existuje pro více programovacích jazyků, nejrozšířenější je však pro C#, Javu a JavaScript (zejména klientská část).

4.2.4 Postman

Postman je komplexní aplikace pro návrh, testování a monitoring REST API [45]. Původně existoval pouze jako doplněk prohlížečů, dnes už jako multiplatformní nativní aplikace. Testované požadavky je možné ukládat do kolekcí a dokumentovat je. Z kolekcí lze generovat dokumentaci a Postman nabízí také podporu pro automatické testy, které jsou psány v programovacím jazyce JavaScript. Na servech Postmana může být spuštěn mock pro simulaci API z definovaných kolekcí.

Další příjemnou funkcí je zmíněný monitoring API, kde lze například nastavit automatické spouštění testu v dané periodě a při negativním výsledku dojde k upozornění uživatele. Postman také umožňuje týmovou spolupráci a je dostupný v placené verzi i zdarma.

4.3 Mock server

Při popisu technologií a nástrojů pro práci s webovými službami byl zmíněn tzv. mock server. Nyní je na čase říct co vlastně takový server dělá a k čemu je dobrý.

Mock server se používá pro simulaci chování jiného serveru. Jeho funkcionalita se zpravidla omezuje pouze na poskytování statických či dynamických, náhodně generovaných dat na základě požadavku. Používá se zejména při vývoji aplikací typu klient-server. Vývoj serverové části je většinou pomalejší, a proto se jeho funkčnost dočasně nahrazuje právě pomocí mocku. To zajistí pohodlnější vývoj klientské části aplikace. Další možnost použití je pro účely testování, kde mock server určitě smysl má. To protože ho můžeme jednoduše nastavit, aby vracel požadovaná testovací data bez nutnosti uložení např. do databáze apod.

Pokud je potřeba, mock server může využít zmíněných nástrojů, které jsou schopny z definice API mock server spustit. Jejich výhodou je, že nepotřebují vlastní testovací stroj pro nasazení mocku. V opačném případě je možné použít

[9]Například skrze objektově relační mapování.

připravené řešení či naprogramovat mock vlastní. Poslední volba se hodí zejména, když je po mock serveru požadována specifická funkčnost.

4.4 Apache JMeter

Nástroj primárně určený pro měření výkonnosti webových aplikací, využívající protokol HTTP [46]. Nicméně lze použít i pro testování SQL databáze či služeb pracujících na jiných protokolech. Jedná se open source nástroj naprogramovaný v jazyce Java

Pomocí tzn. test plánů lze definovat specifické požadavky pro simulaci reálné zátěže. Lze uvést formu a počet různých požadavků, počet vláken simulujících jednotlivé uživatele či různé druhy zpoždění. Testovací data mohou být generována či čtena z připravených souborů. Poskytuje nástroje pro extrakci dat z populárních formátů HTML, JSON, XML či obyčejného textu. Funkcionalita JMeteru může být rozšířena pomocí uživatelsky definovatelných pluginů, které se vytvářejí zejména skriptovacím jazykem Groovy.

Jelikož JMeter poskytuje nástroje pro protokol HTTP, může být využit pro testování webových služeb typu SOAP i REST. Samozřejmě za předpokladu, že používají protokol HTTP jako přenosovou vrstvu.

4.5 Shrnutí

Ekosystém SOAPu je postaven především na rozšiřujících specifikacích (standardech). Je jich velké množství a řeší celou řadu problémů vyskytujících se při vývoji rozsáhlých systémů. Na druhou stranu některé jsou dost složité, což může snižovat jejich reálnou použitelnost. Navíc se vyskytují obdobné specifikace, které řeší ten samý problém a ekosystém tak může působit nepřehledně. REST ekosystém se drží jednoduchosti a formální specifikace se v něm vyskytují vzácně, výjimkou je například zmiňovaný standard Odata – definující způsob přístupu ke zdrojům.

Pro formální popis API je v kontrastu monopol WSDL oproti možnosti výběru různých modelovacích jazyků. Co se týče testovaní, tak oba ekosystémy mají k dispozici kvalitní nástroje. Obecně platí, že SOAP je složitější něž REST a obdobně to platí i pro jejich ekosystémy.

Mimo jednotlivé ekosystémy došlo i na popis tzn. mock serveru, který je využíván k simulaci navrženého API pro vývojové účely. Dále také byla popsána aplikace Apache JMeter, která byla využita pro účely experimentů, kterými se zabývá kapitola 7.

5 Technologie pro webové služby použité v praktické části

Tato kapitola se zaměřuje na vybrané programovací jazyky a technologie, které byly použity pro vývoj demonstrační aplikace popsané v kapitole 6. Budou stručně popsány a pro každou technologii bude uveden ukázkový příklad komunikace pomocí SOAP či REST. Bude se jednat o službu která vrací pozdrav jako prostý řetězec. Parametrem bude jméno, kterému je pozdrav určen. V případě RESTu se bude jednat o zdroj na adrese `/hello/{name}`, kde `{name}` značí tzv. variabilní parametr, v tomto případě jméno pro pozdrav. Pro REST technologie budou navíc uváděny příklady jednoduché autentizace a překladu chyb na HTTP odpovědi.

V kontextu s REST technologiemi je vhodné stručně definovat následující pojmy:

- *Routování HTTP požadavků* – zjednodušeně routování, znamená delegování příchozího HTTP požadavku na obslužné funkce či metody, včetně předání příslušných parametrů a vrácení HTTP odpovědi.
- *Middleware* – drobná jednoúčelová komponenta systému, implementující definované rozhraní s možností modifikovat HTTP zprávy. Jednotlivé middlewary na sebe mohou navazovat. Lze použít například pro autentizaci požadavků.
- *Error handler* – komponenta mapující chyby aplikace na výstup (HTTP odpověď) v požadovaném tvaru.

5.1 Java

Java je objektově orientovaný programovací jazyk. Dlouhodobě patří mezi nejpoužívanější jazyky. Ve Standardní verzi (Java SE) obsahuje základní funkcionalitu a je vhodná pro vývoj desktopových aplikací. Nad ní je postavená edice pro vývoj rozsáhlých aplikací a informačních systémů (Jakarta EE, dříve Java EE). Jako poslední je zjednodušená edice, zaměřená na vývoj aplikací pro mobilní nebo jednoúčelové zařízení (Java ME). Souhrnně jsou označovány jako „Platforma Java".

Jakarta EE obsahuje API specifikace JAX-WS a JAX-RS pro vývoj webových služeb využívající SOAP respektive REST. Tyto specifikace jsou implementovány v rámci tzn. aplikačních serverů (případně i samostatnými knihovnami), např.: GlassFish, Payara, WildFly/JBoss, IBM WebSphere/IBM WebSphere Liberty. Aplikace naprogramované pomocí specifikací by měly spolehlivě fungovat na jakémkoliv aplikačním serveru, který je implementuje. V praxi to však ne vždy zcela platí, jelikož většinou je třeba pracovat s částečně odlišnou funkcionalitou pro daný aplikační server (týká se hlavně různých konfigurací apod.).

5.1.1 JAX-WS

JAX-WS (Java API for XML Web Services) je API specifikace pro vývoj webových služeb komunikující, skrze XML, obzvláště pro služby pracující na protokolu SOAP [47]. Odstiňuje vývojáře od složitosti komunikace samotného protokolu a ten se tak může soustředit na logiku aplikace. Podporuje různé styly komunikace, preferuje a implicitně používá *Document/literal*. Specifikace umožňuje vytvářet serverovou i klientskou část služeb. JAX-WS se poprvé objevilo v Javě EE 5, konkrétně ve verzi 2.0. Existují nástroje, které umožňují generovat kód z WSDL souboru i naopak (wsimport, wsgen, případně nástroje poskytující vývojová prostředí). Tvorbu zjednodušuje použití Java anotací[10] pro popis služeb. Záleží na jednotlivých implementacích, kolik standardů implementují (jako např. WS-Security), nicméně základní funkcionality protokolu SOAP 1.1/1.2 by měly poskytovat všechny.

5.1.1.1 Server

Poskytovatel webové služby je definován jako klasická Java třída. Ta nesmí být konečná ani abstraktní, musí mít bezparametrický konstruktor. Metody, představující operace služby nesmí být statické a konečné. Typy parametrů a návratových hodnot jsou omezeny technologií JAXB (Java Architecture for XML Binding), která provádí jejich XML serializaci/deserializaci. Následuje přehled nejdůležitějších anotací pro implementaci:

- *@WebService* – označení třídy/rozhraní reprezentující službu.
- *@SoapBinding* – specifikuje styl zpráv a kódování.
- *@WebMethod* – značí operaci služby.
- *@WebParam* – definuje parametry operace, mohou být vstupní, výstupní i vstupně-výstupní, případně posloužit pro mapovaní SOAP hlaviček.
- *@WebResult* – nastavení jména elementu s návratovou hodnotou, může být použit i pro návratové SOAP hlavičky.

Takto navržené služby se zkompilují a zabalí do formátu WAR (Web application ARchive)[11], který se poté nasadí na aplikační server. Ten poté službu společně s WSDL souborem vystaví pro klienty. Je třeba dodat, že WAR soubor musí obsahovat konfiguraci webové aplikace i webové služby, kterou poskytuje (liší se dle aplikačního serveru).

[10] Způsob pro přiřazení metadat některému z elementů jazyka (třída, metoda, proměnná apod.). Mohou být pouze informativní či ovlivňovat chování programu.

[11] Soubor pro distribuci webových aplikací v jazyce Java.

```
1  @WebService
2  public class HelloServcice {
3
4   @WebMethod
5   public String sayHello(String name) {
6     return "Hello " + name + ".";
7   }
8  }
```

Zdrojový kód 12: JAX-WS – implementace koncového bodu služby, zdroj: [47]

5.1.1.2 Klient

Klientská aplikace je tvořena zpravidla pomocí již zmíněného nástroje *wsimport*. Ten zpracuje WSDL soubor vystavený serverem a vygeneruje lokální proxy, která poskytuje port pro volání vzdálené webové služby. Port má stejné rozhraní jako třída implementující koncový bod na serveru.

```
1  public static void main(String[] args) {
2   HelloService service = new HelloService();
3   System.out.println(service.getPort().sayHello("world"));
4  }
```

Zdrojový kód 13: JAX-WS – implementace klienta

5.1.2 JAX-RS

JAX-RS (Java API for RESTful Web Services) je API specifikace pro vývoj serverové i klientské část webových služeb podle architektonického stylu REST, za pomoci protokolu HTTP [48]. V rámci Javy EE se poprvé objevila ve verzi 1.1 (od Java EE 6). Mimo jiné poskytuje API pro (de)serializaci dat z těla zpráv. Záleží na konkrétním aplikačním serveru, ale vetšinou implementace poskytují nástroje pro práci s formátem JSON, či XML.

5.1.2.1 Server

Webová služba je realizována třídou, stejně jako u JAX-WS s obdobnými omezeními. Pro realizaci jsou využity Java anotace, které značně vývoj zjednodušují a urychlí.

Hlavní anotací je anotace `@Path`, která akceptuje URI, parametr identifikující zdroj. Dá se použít na celou třídu i její metody. Metoda reprezentující jednotlivou operaci se anotuje anotacemi, jejichž jména jsou odvozeny z názvu HTTP metod, tedy např.: `@GET`, `@POST` apod. Typ přenášených dat lze specifikovat anotacemi `@Produces` a `@Consumes`, opět lze použít pro celou třídu

či jednotlivé metody. Parametry dotazu lze zpřístupnit anotacemi (používají se u argumentů metod): `@QueryParam`, `@HeaderParam`, `@PathParam` aj. Důležitou anotací je anotace `@Provider`. Ta se používá pro značení tříd, které implementují rozhraní z JAX-RS a chceme, aby ho JAX-RS bylo schopné objevit a použít pro daný účel. Zkompilovaná RESTová aplikace je zabalena do WAR souboru (stejně jako u JAX-WS s patřičnými konfiguracemi) a nasazena na aplikační server.

```
1  @Path("/hello")
2  public class HelloWorld {
3
4    @GET
5    @Path(/{name})
6    @Produces("text/plain")
7    public Response sayHello(@PathParam("name") String name) {
8      return Response.ok("Hello " + name + ".").build();
9    }
10 }
```

Zdrojový kód 14: JAX-RS – implementace zdroje

Chybové stavy

JAX-RS poskytuje nástroje pro implementaci error handlerů. Rozhraní `javax.ws.rs.ext.ExceptionMapper` je implementováno pro požadovaný typ výjimky, který má obsluhovat. Pokud nastane výjimka je odchycena a požadovaným způsobem přeložena na odpověď, která je vrácena klientovi. Výhodou toho řešení je, že lze vytvářet vlastní výjimky (runtime), které nemusejí být odchytávány a nakonec se o ně postará error handler.

```
1  @Provider
2  public class ErrorHandler implements ExceptionMapper<Exception> {
3
4    @Override
5    public Response toResponse(Exception exception) {
6      return Response
7        .status(500)
8        .entity("An internal error has occurred.")
9        .type("text/plain")
10       .build();
11   }
12 }
```

Zdrojový kód 15: JAX-RS – generický error handler na straně serveru

Filtrace požadavků

Filtrace požadavků je důležitá součást JAX-RS. Umožňuje příchozí požadavky na server měnit, zahazovat nebo provádět jiné potřebné operace (chová se v podstatě jako middleware). Vytvoření vlastního filtru spočívá v implementaci rozhraní `javaxws.rs.container.ContainerRequestFilter`. K dipozici je kontext požadavku, kde je možné manipulovat např. s hlavičkami či tělem požadavku.

Filtraci požadavku pro účel autentizace ukazuje zdrojový kód 16. Z kontextu požadavku se získá autorizační hlavička s očekávaným autentizačním tokenem. Ten se ověří a v případě, že není validní je klientovi přímo vrácena odpověď se stavovým kódem 401. Při implementaci filtru je důležité třídě, která filtr implementuje, přiřadit anotaci `@PreMatching`. Ta způsobí, že bude filtr zpracován před každým požadavkem na jakýkoliv zdroj.

```
1  @Provider
2  @PreMatching
3  public class AuthFilter implements ContainerRequestFilter {
4
5    @Override
6    public void filter(ContainerRequestContext context) {
7      String token = context.getHeaderString("Authorization");
8
9      if (!isTokenValid(token)) {
10       context.abortWith(
11         Response.status(401).build());
12     }
13   }
14 }
```

Zdrojový kód 16: JAX-RS – filtrace požadavku pro autentizaci na straně serveru

5.1.2.2 Klient

JAX-RS obsahuje také specifikaci definující klienta pro volání RESTových služeb. Samozřejmě umožňuje volat libovolné metody na zdrojích, přidávat entity do těla či hlaviček požadavku. U klienta lze zaregistrovat rozšíření v podobě JAX-RS komponenty pro modifikaci chování. Tím lze například docílit, aby všechny požadavky obsahovaly implicitně autentizační hlavičku a nemusela být tak předávána při každém požadavku explicitně.

5.2 PHP

PHP (Hypertext Preprocessor) je skriptovací, jazyk určený především pro vývoj dynamických internetových stránek či webových aplikací. I když už není tak populární jako dřív, stále se pro vývoj používá. Navrátit PHP zpět na výsluní se

```
public static void main(String[] args) {
  String name = "world";
  Client client = ClientBuilder.newBuilder().build();
  String greeting = client.target("/hello")
    .path(name)
    .request("text/plain")
    .get()
    .readEntity(String.class);

  System.out.println(greeting);
}
```

Zdrojový kód 17: JAX-RS – implementace klienta

snaží nové verze jazyka. Ty například vylepšují výkon interpretru či zavádí do jazyka možnost statické definice datových typů. PHP nedefinuje standardy pro vývoj webových služeb, nicméně implementace SOAP API umožňuje rozšíření jazyka o modul SOAP. Implementace RESTových služeb je většinou realizována pomocí některého z nezávislých frameworků.

5.2.1 SOAP extenze

Pro použití musí být nainstalována a povolena v konfiguračních souborech PHP. Poskytuje třídy pro vytvoření serverové i klientské části SOAP služby. Podporuje pouze základní standard SOAP verze 1.1 či 1.2, nepodporuje optimalizaci zpráv MTOM.

5.2.1.1 Server

Pro definici SOAP serveru je k dispozici třída `SoapServer` [49]. Při její inicializaci lze nastavit cestu k WSDL souboru a konfiguraci serveru. Konfigurace je tvořena asociativním polem a umožňuje nastavit např. verzi SOAPu či styl komunikace. Nepodporuje generování WSDL, tudíž se hodí spíše pro implementaci služeb s již navrženým API. Následující seznam popisuje nejdůležitější metody třídy:

- *addFunction* – přijímá jméno funkce (či pole s jmény), které má služba poskytovat. Jméno funkce a parametry musí odpovídat definovanému API. Funkce musí být přístupné ze skriptu, kde je server vytvořen.
- *setClass* – přijímá jméno třídy implementující API (musí mít správně pojmenovány metody a jejich parametry).
- *setObject* – podobné jako předchozí, avšak jako parametr požaduje instanci třídy implementující API.
- *addSoapHeader* – přidá k odpovědi SOAP hlavičku.

- *handle* – volá aby server začal obsluhovat požadavky. Standardně funguje na HTTP metodě POST, lze ovlivnit parametrem.

Vytvořený server poté obsluhuje požadavky na adrese, kterou určuje skript ve kterém je server definován. Pokud by bylo třeba tohle chování ovlivnit, je možné použít „Output Buffering“, který PHP podporuje. Ten funguje tak, že to co by se normálně zapsalo hned na výstup (prohlížeč, terminál) se ukládá do bufferu. Z bufferu lze data poté vybrat a naložit s nimi dle potřeby.

```
1 class HelloService {
2   public function sayHello(string name): string {
3     return "Hello $name.";
4   }
5 }
6
7 $soapServer = new SoapServer('helloService.wsdl');
8 $soapServer->setObject(new HelloService());
9 $soapServer->handle();
```

Zdrojový kód 18: PHP SOAP – implementace koncového bodu služby

5.2.1.2 Klient

Pro volání SOAP služeb je určená třída `SoapClient`. Inicializace je podobná jako u serveru. Parametry jsou adresa WSDL souboru a asociativní pole s případnou konfigurací. Z předané adresy WSDL klient zjistí jak službu volat. Pokud adresa není uvedena, lze k lokalizaci služby využít konfiguraci.

Pro volání operace slouží metoda `__soapCall`. Jako parametry se předají jméno operace a asociativní pole s parametry, které operace vyžaduje (zabalené v dalším poli). Alternativou je volání dané služby na instanci klienta přímo pomocí šipkové notace (stejně jako volání běžné metody). Klient dále poskytuje metody na nastavení SOAP hlaviček, zkoumání služby (získání definice operací) apod.

```
1 $soapClient = new SoapClient('http://localhost/helloService?wsdl');
2 $greeting = $soapClient->sayHello(['name' => 'world']);
3
4 echo $greeting;
```

Zdrojový kód 19: PHP SOAP – implementace klienta

5.2.2 Slim framework

Jednoduchý framework navrženy především pro tvorbu služeb založených na stylu REST [50]. Jednoduše lze nainstalovat pomocí Composeru[12]. Podporuje PSR-7, což je neoficiální standard specifikující rozhraní pro HTTP zprávy (požadavek, odpověď) a tím umožňuje spolupráci s odlišnými frameworky, které jej také dodržují.

Framework obsahuje po instalaci řadu souborů. Soubor `app/routes.php`, definuje routování požadavků na funkce či metody. Ty musejí odpovídat rozhraní – jako parametry přijímat požadavek, odpověď, pole (zde budou potencionální proměnné z požadavku – path, query) a vracet opět odpověď. Dále soubor `app/middleware.php` sloužící pro registraci middleware. Pro celkový běh je důležitý skript `public/index.php`, který celý framework inicializuje (zde se například definuje mimo jiné i globální error handler).

```
1  $app->get('/hello/{name}', function ($request, $response, $args) {
2    $name = $args['name'];
3    $response->getBody()->write("Hello $name.");
4
5    return $response->withHeader('Content-Type', 'text/plain');
6  });
```

Zdrojový kód 20: Slim framework – implementace zdroje

```
1  class AuthenticationMiddleware implements Middleware {
2
3    public function process($request, $handler): {
4      $authorizationHeaders = $request->getHeader('Authorization');
5
6      if ($this->isTokenValid($authorizationHeaders)) {
7        return $handler->handle($request);
8      }
9
10     $response = new \Slim\Psr7\Response();
11
12     return $response
13       ->withStatus(401)
14       ->withHeader('Content-Type', 'text/plain');
15   }
16 }
```

Zdrojový kód 21: Slim framework – middleware kontrolující autentizaci požadavku

[12]Nástroj pro správu programových balíčků jazyka PHP.

```
class HttpErrorHandler extends SlimErrorHandler {

  protected function respond(): {
    $response = $this->responseFactory->createResponse(500);
    $response->getBody()->write('An internal error has occurred.');

    return $response->withHeader('Content-Type', 'text/plain');
  }
}
```

Zdrojový kód 22: Slim framework – generický error handler

5.2.3 Guzzle

Guzzle je PHP knihovna, fungující jako HTTP klient. Usnadňuje posílání HTTP zpráv [51]. Stejně jako Slim framework dodržuje PSR-7. Pro svou jednoduchost a funkcionalitu je oblíbeným klientem pro volání RESTových služeb. Pro volání služby je nutné inicializovat třídu
`GuzzleHttp\Client`. Tu lze nakonfigurovat pro snížení redundance kódu a zjednodušení používání (základní URI, výchozí hlavičky apod.) Poté lze volat REST služby. Problém je pouze se serializací dat do formátu JSON – je nutné použít vlastní řešení, případně k tomu určenou knihovnu.

```
  $response = $guzzleClient->get('http://localhost/hello/word');
  $greeting = $response->getBody()->getContents();

  echo $greeting
```

Zdrojový kód 23: Guzzle – implementace klienta

5.3 Python

Python je skriptovací programovací jazyk. Používá se pro vývoj desktopových, webových, ale i jinak specializovaných aplikací (prototypování, vědecké výpočty apod). Pro svou jednoduchost a použitelnost se stává v posledních letech stále populárnější. K vývoji webových služeb lze použít frameworků či knihoven.

Pro vývoj SOAP serveru je využita knihovna *Spyne*, pro vývoj klienta knihovna *Zeep*. REST služba na straně serveru je vytvořena pomocí frameworku *Flask* a klienta obstarává knihovna *Requests*.

5.3.1 Spyne

Knihovna pro poskytování nejen SOAPových webových služeb [52]. Alternativní a udržovaná knihovna s podobnou funkcionalitou pro Python momentálně nee-

xistuje. Vývoj podle předem definovaného API pomocí WSDL je obtížný. Mnohem jednoduší je akceptovat API, které je generováno z naprogramované služby. Nepodporuje MTOM. Koncové body jsou definovány jako třídy s metodami představující SOAP operace (musí dědit z `ServiceBase`). Metody jsou opatřeny dekorátorem `@rpc`, který přijímá množství parametrů pro nastavení vystavované operace (vstupní parametry, návratová hodnota, jména operací apod.) Třídy jako koncové body jsou poté předány třídám zajištující provoz samotného serveru.

```
1 class PublicationService(ServiceBase):
2   @rpc(
3     String,
4     _returns=String,
5     _operation_name='sayHello'
6 )
7   def say_hello(self, name):
8     return 'Hello ' + name + '.'
```

Zdrojový kód 24: Spyne – implementace koncového bodu služby

5.3.2 Zeep

Moderní SOAP klient pro verzi 1.1 i 1.2 [53]. Podporuje MTOM i některé části z WS-Security specifikace. Obsahuje třídu `Client`, která se inicializuje s adresou WSLD požadované služby jako parametr. Volitelně přijímá další argumenty zejména pro konfiguraci. Pro volání služby poskytuje service proxy objekt, na kterém lze volat jednotlivé operace jejich jménem.

```
1 client = Client(localhost/helloService?wsdl)
2 print(client.service.sayHello('world'))
```

Zdrojový kód 25: Zeep – implementace klienta

5.3.3 Flask

Minimalistický webový framework postavený nad specifikací WSGI[13], což umožňuje integraci s technologiemi, které specifikaci také dodržují [54, 55]. Je jednoduchý na použití a dokonale se hodí na vývoj jednoduché REST aplikace. V komunitě se těší popularitě a je k dispozici celá řada rozšíření. Nainstalovat lze jednoduše pomocí správce balíčků pro moduly jazyka Python. Inicializovaný Flask poskytuje mimo jiné tyto dekorátory použitelné na funkce:

[13] Web Server Gateway Interface – specifikace pro vývoj webových aplikací v jazyce Python.

- *@route* – routování, jako parametry přijímá kontextovou URI zdroje a výčet podporovaných HTTP metod. Path proměné se navazují na parametry funkce.
- *@before_request* – slouží k vytvoření middlewaru, který se aplikuje před zpracováním požadavku pomocí routování.
- *@errorhandler* – přijímá třídu reprezentují výjimku, kterou má obsluhovat, ta je pak dostupná skrze definovaný parametr funkce.

```
1 @app.route('/hello/<name>', methods=['GET'])
2 def say_hello(name: str) -> Response:
3   greeting = 'Hello ' + name + '.'
4
5   return Response(
6     status=200,
7     mimetype='text/plain',
8     response=greeting)
```

Zdrojový kód 26: Flask – implementace zdroje

```
1 @app.before_request
2   def authentication_middleware():
3   authorization_token = request.headers.get('Authorization')
4
5   if not is_token_valid(authorization_token) -> Response:
6     return Response(status=401)
```

Zdrojový kód 27: Flask – middleware kontrolující autentizaci požadavku

```
1 @app.errorhandler(Exception)
2 def handle_validation_error(error: Exception) -> Response:
3
4 return Response(
5   status=500,
6   mimetype='text/plain',
7   response='An internal error has occurred.')
```

Zdrojový kód 28: Flask – generický error handler

5.3.4 Requests

Populární knihovna fungující jako HTTP klient. Usnadňuje zasílání HTTP zpráv. Má velmi jednoduché rozhraní, což zjednodušuje volání REST služeb. Kromě standardních funkcí, poskytuje mimo jiné podporu pro autentizaci.

```
1 greeting = requests.get('http://localhost/hello/world').text
2 print(greeting)
```

Zdrojový kód 29: Requests – implementace klienta

5.4 JavaScript/Node.js

Node.js je multiplatformní běhové prostředí, umožňující spouštět skriptovací jazyk JavaScript mimo webové prohlížeče [56]. Je postaveno na enginu, který je používán webovým prohlížečem Google Chrome. Hlavním účelem Node.js je vývoj serverové části webové aplikace. Běh programu je řízen pomocí událostí s možností provádět asynchronní I/O operace. Díky tomu mají aplikace postavené na Node.js vysokou propustnost a škálovatelnost.

Jádro Node.js tvoří tzv. smyčka událostí (event loop). Ta běží v jediném vlákně, přičemž asynchronní požadavky pouze předá dál (Work poolu, který zajistí nezávislé zpracování) a obslouží až callback[14]. Tento princip tedy umožní nečekat na zpracování potencionálně dlouhotrvajících operací jako je zápis/čtení souboru či databáze. Výchozí správce balíčků pro Node.js je NPM (Node Package Manager), pomocí kterého lze doinstalovávat požadované moduly.

Pro komunikaci skrze SOAP byl využit stejnojmenný modul. Pro REST zase framework *Express* a klientská knihovna *Axios*.

5.4.1 Modul Soap

Dobrý nástroj pro realizaci serverové i klientské části SOAP API [57]. Poskytuje bohatou funkcionalitu včetně MTOM či WS-security.

5.4.1.1 Server

Serverová část se vytvoří pomocí funkce *listen*. Té se předá server sloužící jako přenosová vrstva (může být tvořena *HTTP* modulem či pomocí frameworku *Express*) a cestou kde bude služba k dispozici. Dále se předá objekt reprezentující službu s funkcemi představující SOAP operace a samotný WSDL soubor definující API služby. V poslední řadě je předán callback provedený po spuštění serveru.

[14]Funkce volaná po provedení dané operace, při programování se předává jako argument jiné funkci.

```
var service = {
  helloService: {
    helloServicePort: {
      sayHello: function (args, callback) {
        callback({greeting: 'Hello ' + args + "."})
      }
    }
  }
};

soap.listen(server, '/helloService', service, wsdl, function () {});
```

Zdrojový kód 30: Node.js SOAP – implementace koncového bodu

5.4.1.2 Klient

Volání služby se provádí prostřednictvím funkce *createClient* s předaným URL WSDL souboru a callbackem kde je k dispozici proxy objekt, jehož prostřednictvím lze volat SOAP operace.

```
var wsdl = 'http://localhost/helloService?wsdl'
var args = {name: 'world'};
soap.createClient(wsdl, function(err, client) {
  client.sayHello(args, function(err, result) {
    console.log(result.greeting);
  });
});
```

Zdrojový kód 31: Node.js SOAP – implementace klienta

5.4.2 Express

Minimalistický framework pro vývoj webových služeb, komunikující prostřednictvím protokolu HTTP. Je tedy vhodný a dost používaný pro vývoj REST služeb. Inicializovaný *Express* poskytuje metody `get`, `post`, `delete`... pro routování. Parametry jsou kontextová URI zdroje a funkce obsluhující požadavek (může jich být více předaných polem). Důležitá je metoda funkce `use`, která slouží pro registraci middlewaru. Error handlery jsou implementovány jako middleware a musí být registrovány až po routovacích funkcích.

Jak již bylo řečeno, routovací i middleware funkce přijímají callback funkce. Ty mají k dispozici požadavek, odpověď a callback funkci pro volání další handlerů. Pokud má middleware plnit účel error handleru má k dispozici navíc samotný error.

```
app.get('/hello/:name', function (req, res) {
  res.header('Content-Type', 'text/plain')
    .send('Hello ' + req.params.name + '.');
});
```

Zdrojový kód 32: Express – implementace zdroje

```
app.use((request, response, next) => {
  if (!isTokenValid(request.headers.authorization)) {
    response.status(401)
      .header('Content-Type', 'text/plain').send();
  } else {
    next();
  }
});
```

Zdrojový kód 33: Express – middleware kontrolující autentizaci požadavku

5.4.3 Axios

HTTP klient pro JavaScript i Node.js [58]. Je založený na příslibech a poskytuje možnost použití asynchronních funkcí, což je přístup, jak zjednodušit programování založené na callback funkcích. Kromě očekávané funkcionality HTTP klienta obsahuje také například autentizační principy či možnost zjednodušeně volat AJAX.

5.5 Shrnutí

Uvedené popisy a příklady technologií vykazují podobné rysy. V případě SOAPu jsou při použití již definovaného WSDL souboru k dispozici proxy objekty (explicitně či implicitně generované), které v podstatě řeší celou komunikaci. To prokazuje určitou standardizaci. V případě RESTu lze vypozorovat minimálně dodržování osvědčených principů, kdy je server implementován pomocí principu routování, middlewaru a error handlerů.

Dle tohoto pozorování se dá říct, že přechod na jinou technologii nemusí být z pohledu webových služeb nic složitého a jde v podstatě pouze o osvojení programování v daném jazyce.

```
app.use((error, request, response, next) => {
  response.status(500)
    .header('Content-Type', 'text/plain')
    .send('An internal error has occurred.');
});
```

Zdrojový kód 34: Express – generický error handler

```
var greeting = (await axios.request({
  method: 'GET',
  url: 'http://localhost:9040/hello/word'
})).data;

console.log(greeting)
```

Zdrojový kód 35: Axios – implementace klienta

6 Praktická část

Kapitola o praktické části práce je zaměřena na popis ukázkové aplikace. Nejprve bude stručně popsána samotná aplikace a její rozhraní. Poté budou představeny různé implementace za pomoci programovacích jazyků a technologií pro webové služby, jenž byly popsány v předchozí kapitole. Na závěr kapitoly dojde ke stručnému zhodnocení.

6.1 Aplikace

Pro ukázkové a testovací účely využití webových služeb byla navržena jednoduchá aplikace typu klient-server, komunikující pomocí webových služeb. Aplikace obsahuje funkcionalitu pro správu publikací. Každá publikace je definována daty:

- Celočíselným identifikátorem (generováno pro jednoznačnou identifikaci).
- Titulkem.
- Rokem vydání.
- Autory, kteří publikaci napsali.
- Volitelným dokumentem ve formátu PDF.

Titulek, autoři a název dokumentu je omezen na 255 znaků. Dokument musí mít koncovku .pdf a maximální velikost 20MB. Aplikace poskytuje uživateli následující funkcionalitu:

- Přidat publikaci (společně s validací vstupních dat).
- Upravit publikaci (společně s validací vstupních dat).
- Smazat publikaci.
- Zobrazit detail přidané publikace (zobrazení, stáhnutí dokumentu).
- Zobrazit všechny přidané publikace.

6.1.1 Rozhraní webových služeb

K návrhu rozhraní aplikace bylo využito tzv. „API first" přístupu. Ten spočívá v tom, že se před samotným programováním navrhne rozhraní nebo-li API aplikace a to se posléze implementuje. To umožňuje lepší kontrolu nad formou API. Tento přístup je výhodný proto, že bylo nutné implementovat aplikaci se stejným API v různých programovacích jazycích. Jednotné API umožňuje jednotnou komunikaci libovolného klienta s libovolným serverem, což také demonstruje interoperabilitu aplikací napsaných v rozdílných programovacích jazycích pomocí webových služeb.

K návrhu SOAP API posloužil standard WSDL ve verzi 1.1 (použitá SOAP verze je také 1.1). REST API je definováno specifikací OpenAPI verze 3.0.

6.1.1.1 SOAP API

Je rozděleno do dvou WSDL souborů. První definuje operace pro práci s publikacemi a druhý pro operace s dokumenty. Oba WSDL soubory používají styl vazeb typu – *Document/literal wrapped* a mají definované vlastní XML schéma s elementy, které jsou použity pro vstupní a výstupní zprávy. Každá operace je typu „Požadavek-odpověď“. To znamená, že pokaždé vrací odpověď, byť třeba prázdnou.

Služba pro práci s publikacemi je popsaná souborem `publicationService.wsdl` a má následující operace:

- *validatePublication* – validuje publikaci. Pokud validace selže, jsou vráceny chybové hlášky.
- *addPublication* – přidá publikaci.
- *updatePublication* – upraví stávající publikaci podle identifikátoru, pokud existuje.
- *deletePublication* – smaže publikaci podle identifikátoru, pokud existuje.
- *getPublication* – vrátí publikaci podle identifikátoru, pokud existuje.
- *getPublications* – vrátí všechny přidané publikace.

Služba pracující s dokumenty publikací je popsaná souborem `documentService.wsdl`. Kvůli jednoduchosti jsou binární data zakódovaná do Base64 a přenášena v těle zprávy. Není tedy využito optimalizace zpráv MTOM. Služba má následující operace:

- *validateDocument* – validuje předaný dokument. Pokud validace selže, jsou vráceny chybové hlášky.
- *addUpdateDocument* – přidá/nahradí dokument (úprava podle identifikátoru publikace).
- *deleteDocument* – smaže dokument podle identifikátoru publikace, pokud existuje.
- *getDocument* – vrátí dokument podle identifikátoru publikace, pokud existuje.

6.1.1.2 REST API

Pro zápis je využit formát YAML, zejména pro svou lepší čitelnost. API je definováno pomocí 3 souborů. Jeden soubor definuje API pro publikace a druhý pro dokumenty. Třetí soubor je pro sdílené datové struktury (např.: definice parametrů HTTP požadavků či generických HTTP odpovědí), které jsou využité v obou předešlých souborech, pro snížení redundance a snadnější údržbu. Samotná data

publikace jsou reprezentována formátem JSON. Výjimkou je samotný dokument publikace, který je reprezentován binárními daty.

API pro práci s publikacemi je definována souborem `publication.yaml` a obsahuje následující koncové body s metodami:

- */publications/validate*:
  - *POST* – validuje publikaci. Pokud validace, selže jsou vráceny chybové hlášky.
- */publications*:
  - *POST* – přidá publikaci.
  - *GET* – vrátí všechny přidané publikace.
- */publications/{publicationId}* – kde {publicationId} značí identifikátor publikace:
  - *PUT* – upraví stávající publikaci podle identifikátoru, pokud existuje.
  - *DELETE* – smaže publikaci podle identifikátoru, pokud existuje.
  - *GET* – vrátí publikaci podle identifikátoru, pokud existuje.

API pro práci s dokumenty publikací je definováno souborem `document.yaml`. Pro účely nahraní dokumentu byla vytvořena vlastní hlavička `X-Document-Name` – nesoucí jméno dokumentu. Při získávání dokumentu je jeho jméno přítomno v hlavičce `Content-Disposition`.

- */publications/document/validate*:
  - *POST* – validuje předaný dokument publikace. Pokud validace selže, jsou vráceny chybové hlášky.
- */publications/{publicationId}/document* – kde {publicationId} značí identifikátor publikace, ke které dokument náleží:
  - *POST* – přidá předaný dokument publikaci.
  - *DELETE* – smaže dokument publikace.
  - *GET* – vrátí dokument publikace.

RESTové API je také zabezpečeno HTTP Basic autentizací. Klientská aplikace tedy musí zasílat autorizační hlavičku v požadavku. V případě chybějícího či nekorektního obsahu hlavičky se požadovaná operace neprovede a je vrácen stavový kód 401.

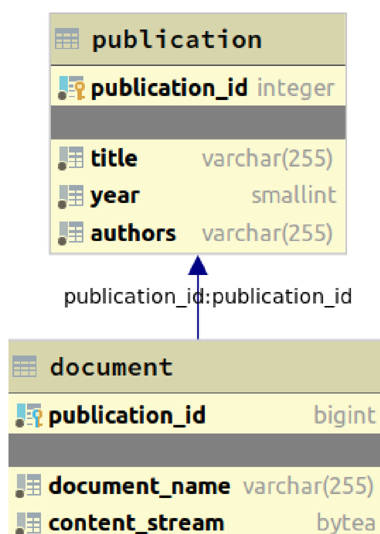
6.1.2 Server

Serverová část tvoří jádro s implementovanou logikou aplikace. Jádro řeší zejména manipulaci s daty na úrovni databázového systému či validaci dat. Jádro je pak využito pro vystavení webových služeb využívající SOAP a REST, které implementují navržené rozhraní pro komunikaci.

6.1.2.1 Databáze

Každá z implementací aplikace využívá pro persistenci dat objektově-relační databázi PostgreSQL. Byla vybrána pro výbornou pověst, za kterou jistě vděčí své spolehlivosti, široké funkcionalitě a dobré výkonnosti.

Databáze má pouhé dvě tabulky – pro publikace a dokumenty. První tabulka je primární s generovaným celočíselným identifikátorem, který ji zároveň propojuje s tabulkou druhou. Schéma databáze je zobrazeno obrázkem 7.



Obrázek 7: Schéma databáze pro testovací aplikace

Je vhodné dodat, že pro RESTové implementace se v určitých případech přímo nabízí využít databázové systémy nerelačního typu, tzn. NoSQL. Motivací pro nasazení databáze tohoto typu je jednoduchost a výborná škálovatelnost. Některé ukládají data jako JSON dokumenty, což se dokonale hodí pro typ implementací využívající JSON jako přenosový formát dat, jelikož odpadá složitost s konverzí. Příkladem databáze tohoto typu je oblíbená MongoDB.

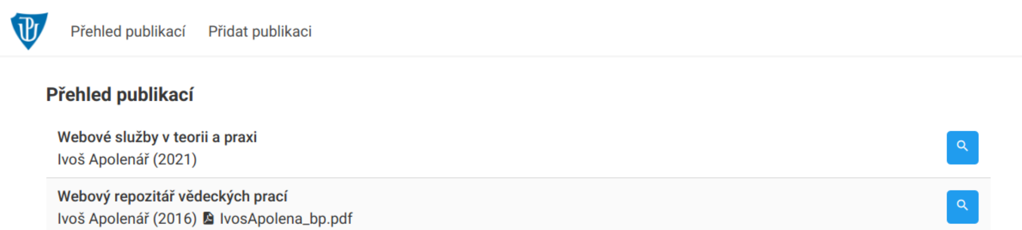
6.1.3 Klient

Klientská část je navrhnuta jako webová aplikace. Je tvořena čtyřmi stránkami:

- *Přehled publikací* – zobrazuje všechny přidané publikace řazené od nejnovější.

- *Přidat publikaci* – poskytuje formulář pro přidání nové publikace (včetně možnosti přidat dokument) s validací dat.
- *Upravit publikaci* – poskytuje formulář pro úpravu stávající publikace (včetně možností přidat/nahradit/smazat dokument) s validací dat.
- *Přehled publikace* – zobrazuje detail vybrané publikace s kontextovými akcemi (upravit, smazat, zobrazit/stáhnout dokument).

Komunikace se serverovou částí probíhá přes SOAP nebo REST. Vždy se používá jeden typ, přičemž rozhoduje konfigurace, která také udává na jaké URL adrese má používaná webová služba fungovat. Obrázek 8 ukazuje podobu úvodní stránky klientské webové aplikace.



Obrázek 8: Ukázka klientské aplikace

6.2 Implementace

Implementace byly provedeny prostřednictvím technologií představených v předchozí kapitole 5.

Každá z implementací má obdobnou osvědčenou architekturu, rozdělenou do několika vrstev:

- Server:
  - *Databázová vrstva* – zajišťuje komunikaci s databází pro ukládání a načítaní entit, jenž reprezentují data (publikace, dokumenty).
  - *Servisní vrstva* – zapouzdření logiky aplikace (využívá předchozí vrstvy, provádí validace apod.), poskytuje rozhraní pro Api vrstvu.
  - *Api vrstva* – Vystavuje SOAP a REST API pro komunikaci, případně zajišťuje konverzi dat pro spolupráci se servisní vrstvou.
- Klient:
  - *Servisní vrstva* – komunikuje se serverovou částí skrze definované API, záleží na konfiguraci, zda-li se jedná o SOAP či REST.

- *Prezentační vrstva* – poskytuje interakci s uživatelem prostřednictvím dynamických webových stránek. Jinými slovy, představuje uživatelské rozhraní.
- *Řídící vrstva* – řídí komunikaci mezi předchozími dvěma vrstvami. Provádí logiku spjatou se zpracováním formulářových dat a případně zajišťuje drobné operace jako např. mapování dat.

Jediná výjimka je u klientské části implementované v Node.js. Ta je tvořena formou testů. Ty ověřují, zda-li je API serverových částí korektní vzhledem k návrhu a funkčnosti.

6.2.1 Java

Aplikace je postavena nad platformou Java EE a byla vyvíjena jako Maven[15] projekt. Hlavní modul definuje závislosti na potřebných knihovnách a jejich verze. Ty pak využívají čtyři pod-moduly:

- *Logic* – Obsahuje logiku aplikace, tedy databázovou a servisní vrstvu. Ta je implementována formou EJB[16] komponent. Databázová vrstva využívá JPA[17]. Modul je distribuován jako JAR (Java ARchive) a jako závislost je součástí modulů *Soap* a *Rest*.
- *Soap* – Vystavuje SOAP API, využívá specifikaci JAX-WS. Pomocné třídy reprezentující přenášená data a rozhraní služby bylo vygenerováno. Rozhraní bylo poté implementováno, distribuuje se jako WAR.
- *Rest* – Mimo jiné obsahuje kontrolery, vystavující REST API dle specifikace JAX-RS. Modul se distribuuje jako WAR.
- *Client* – Implementuje celou klientskou aplikaci. Prezentační vrstva je tvořena pomocí JSF[18]. Modul se distribuuje jako WAR.

Celá aplikace byla původně vyvíjena pro nasazení na aplikační server Payara. Poté byly provedeny drobné úpravy kódu pro nasazení na servery WildFly a IBM WebSphere Liberty.

6.2.2 PHP

Jako základ pro serverovou i klientskou část byl využit Slim Framework. Obě části jsou na sobě nezávislé a každá si definuje vlastní závislosti na ostatních knihovnách pomocí Composeru.

[15] Nástroj pro správu a konfigurovatelné sestavování převážně Java aplikací.

[16] Enterprise Java Beans – serverové komponenty pro tvorbu rozsáhlých modulárních aplikací.

[17] Java persistence API – specifikace pro objektově relační mapování, usnadňující ukládání dat z objektů jazyka do databáze a naopak.

[18] Java Server Faces – komponentově orientovaný framework pro tvorbu uživatelského rozhraní pro platformu JAVA EE.

- *Server* – databázová vrstva komunikuje s databází skrze abstraktní ovladač „PHP Data Objects“. SOAP API je vystaveno pomocí POST routování, PHP SOAP extenze a PHP output bufferingu. REST API používá pro konverzi JSON formátu na PHP objekty a naopak, knihovnu `symfony/serializer`.

- *Klient* – servisní vrstva volá API buď pomocí SOAP extenze nebo knihovny Guzzle. Prezentační vrstva stojí na šablonovacím systému Twig. Routovací metody zde plní funkci řídící vrstvy.

Pro spuštění serveru stačí PHP interpret s extenzemi pro SOAP, PDO a PostgreSQL[19]. Klient musí mít také SOAP extenzi. Kvůli poskytování statického obsahu pro účely prezentační vrstvy, musí být klient nasazen na webový server podporující PHP. Aplikace byla vyvíjena na webovém serveru Apache HTTP.

6.2.3 Python

Podobně jako v předešlém případě jsou serverová a klientská část aplikace v Pythonu postaveny nezávisle na frameworku Flask. Serverová část navíc využívá modul Spyne. Pro instalaci frameworku a modulů byl použit správce balíčku pro moduly Pip. Při vývoji bylo pro každou část vytvořeno tzv. virtuální prostředí. Díky tomu se moduly neinstalovaly globálně, což zamezí případným konfliktům mezi instalovanými moduly.

- *Server* – databázová vrstva používá databázový adaptér Psycopg, který plně implementuje DB API 2.0[20]. SOAP API je vystaveno modulem Spyne, REST API pomocí samotného Flasku.

- *Klient* – servisní vrstva volá SOAP API knihovnou Zeep, REST pomocí knihovny Requests. Prezentační vrstva poskytuje data skrze šablonovací systém Jinja. Řídící vrstva využívá routování z Flasku.

Pro běh serveru i klienta stačí spustit pomocí Python interpretru inicializační skripty. Drobný problém je se SOAP API. To se kvůli problémům s modulem Spyne nepodařilo implementovat v úplné kompatibilitě s navrženým API, a proto funguje pouze v rámci implementace Pythonu. Není tak kompatibilní s ostatními implementacemi.

6.2.4 JavaScript/Node.js

Zde je situace od předešlých implementací odlišná. Serverová část je navržena standardním způsobem. Klientská část je tvořena formou testů. Ty provolávají webové služby a ověřují, zda-li jsou implementovány korektně. Pro instalaci modulů byl použit NPM.

---

[19] Užitečné zejména pro zjednodušení vývoje. Pro produkční nasazení nevhodné z hlediska výkonnosti.

[20] Specifikaci pro práci s databází v jazyce Python.

- *Server* – databázová vrstva používá databázový modul *node-postgres*. REST API je vystaveno pomocí frameworku *Express*. Ten je využit i pro přenos zpráv modulu Soap, který implementuje SOAP API.

- *Klient* – stojí na populárním JavaScriptovém testovacím frameworku *Jest*, který udržuje společnost Facebook Inc. Testuje všechny operace dostupné jak skrze SOAP (modul Soap) tak i REST (module Axios).

Server se dá jednoduše spustit pomocí běhového prostředí Node.js. Testy pomocí NPM.

6.3 Zhodnocení implementací

V kapitole 6.1 byla představena ukázková a testovací aplikace, která byla implementována ve čtyřech programovacích jazycích (viz kapitola 6.2). Přímo se tak nabízí zhodnotit komfort či různé aspekty vývoje v rámci těchto jazyků nejen z pohledu webových služeb.

6.3.1 Java

Díky API first návrhu bylo vytvoření SOAP části rychlé a jednoduché za pomoci generování kódu nástrojem wsimport. Stačilo pouze implementovat vygenerované rozhraní a bylo hotovo. Výhodu tedy je, že není třeba řešit podobu rozhraní služby (název operací, parametry či návratové hodnoty) jelikož je vygenerována přímo z WSDL souboru. Trochu složitější to bylo u klientského kódu, kdy bylo třeba dynamicky měnit adresu služeb dle konfiguračního souboru. To vyžadovalo nutnost hlubšího ponoření do problematiky k nalezení správného řešení.

Vystavení RESTových zdrojů za použití anotací na třídách bylo snadné a přehledné. Třídy s obslužným kódem se opatří anotacemi a o delegování HTTP požadavku se již není třeba starat. Použitím anotací se také jednoduše přiřadí např. hodnoty hlaviček do proměnných atp. REST klient vyžadoval o něco více práce. Stejně jako v případě SOAPu není třeba řešit (de)serializaci dat na objekty. Výhodou je jednoduchost, flexibilita a vyšší transparentnost (dynamická změna adresy služeb nebyla problém). Nevýhodou je jednoznačně nutnost vyvíjet podle definice API, čímž může vznikat prostor pro chyby.

Jako mocná technologie se ukázal Maven. Vynikal při správě dodatečných balíků nebo díky tzv. pluginům, pomocí kterých lze např. nastavit generování právě SOAP kódu. K pohodlnosti vývoje přispěla i tzv. dependecy injection[21] nebo JPA při práci s databází. S aplikačními servery se pracuje obtížněji a je nutné se s nimi dostatečně seznámit, ale jinak jde bezesporu o silnou stránku Javy.

Díky zmíněným aspektům byl vývoj svižný, avšak kvůli komplexnosti a rozmanité funkčnosti je Java vhodná spíše pro střední až velké projekty, kde se dostatečně projeví její výhody.

[21] Vkládání závislostí mezi komponentami programu. V praxi umožňuje vyšší pružnost vývoje nebo testování kódu.

6.3.2 PHP

Implementace stojí na *Slim* Frameworku a byla používána dependency injection. Naprogramovat REST služby bylo namáhavější než u Javy. Chybí více abstrakce a je nutné pracovat přímo s objektem požadavku (pro získání hlaviček, parametrů z URL apod). Směrování požadavku na obslužný kód je nutné uvést ve speciálním souboru, což snižuje komfort používání a hlavně údržby. Navzdory tomu, že hlavní devízou *Slim* frameworku je jednoduchost, absence pokročilejších serializačních nástrojů pro JSON je pro framework s tímto zaměřením nepřípustná. Není sice problém doinstalovat příslušný balíček, ale chvíli potrvá integrace do projektu a problémy mohou nastat při povyšování verzí.

Naprogramovat samotnou SOAP službu nebyl problém. Stačilo vytvořit třídu s metodami, které korespondovaly se jmény operací z WSDL. Obtížnější bylo vystavení SOAPové služby na potřebnou adresu. Pomohlo až směrování Slim frameworku a „Output Buffering".

Klientská řešení pro konzumaci obou typů služby byla jednoduchá. Pro SOAP klient by bylo komfortnější použít kvalitní nástroj pro konverzi asociativních polí na objekty, jelikož pole slouží pro předání parametrů volání služby.

Instalace balíčků pomocí Composeru byla jednoduchá. Composer navíc generuje autoload funkci, čímž odpadá vývojáři starost s propojováním závislých PHP skriptů. Nasazení aplikace patří určitě k těm složitějším. Společně s konfigurací samotného PHP je nutné konfigurovat i zvolený webových server (typicky Apache či Nginx). To se dá sice řešit využitím hojně nabízených hostingů, které jsou již předkonfigurované, to se hodí patrně pouze pro malé projekty.

6.3.3 Python

Python se s využitím frameworku *Flask* jeví jako výborná volba pro menší projekty postavené na stylu REST. Významnou roli v tom hrají dekorátory, které poskytují obdobné pohodlí jako anotace v Javě. Směrování požadavků na obslužný kód je tak na jednom místě. Na druhou stranu chybí např. podpora pokročilejší serializace do JSONu. Základní podpora je sice přítomna jako součást Pythonu, avšak pro složitější datové struktury není dostatečná.

Implementaci SOAP služeb nelze v Pythonu doporučit. *Spyne* je sice použitelný framework, ale nehodí se pro API first vývoj, protože WSDL generuje sám z implementace. Kvůli tomu může být doladění kompatibility poměrně náročný úkol. Kvalitní alternativa neexistuje a tak Python není pro vývoj SOAP služeb nejlepší volbou. Oba typy služeb se nasazují na servery dodržující specifikaci WSGI a jejich správa je malinko jednoduší něž v případě Javy či PHP.

S klientskými částmi obecně nebývá problém a stejně tak je tomu i tady. RESTová volání vyžadovala více programování než u SOAPu, který je pro jednoduchá API příjemně stručný.

Obecně je vývoj v Pythonu velmi rychlý. Balíčků je dostatek a snadno se instalují.

6.3.4 JavaScript/Node.js

Node.js bylo vyvinuto primárně pro psaní serverových webových aplikací. Použití frameworku *Express* poskytuje vše nutné pro vývoj REST služeb. Výhodou je, že nejvyužívanější přenosový formát JSON je serializovanou formou datových typů v JavaScriptu. Díky tomu je s ním práce opravu jednoduchá. Vytvoření samotné RESTové služby je sice jako v PHP jednoduché, ale chybí stejně tak vyšší komfort.

Příjemným překvapením je relativně kvalitní a obsáhlá podpora protokolu SOAP s názornou dokumentací. Vystavení jednodušší služby je pak v případě API first přístupu bezproblémové. Opět obdobně jako v PHP stačí vytvořit objekt, kde jména funkcí korespondují se jmény operací ve WSLD a objekt poté přidat ke kódu spouštějící server. Co se týká klientských částí, je hodnocení obdobné jako v případě jazyka Python.

Žádný problém není ani s instalací balíku pomocí NPM, které navíc dokáže např. spouštět předdefinované úkoly, což lze využít třeba pro spouštění automatizovaných testů či usnadnění spuštění aplikace. Práce s relační databází není problém. V případě použití RESTu, však může být databáze postavená na JSON dokumentech (např. již zmiňovaná MongoDB) lepším řešením, pokud je to ovšem z hlediska použití aplikace vhodné. Nasazování aplikací bylo z daných technologií nejjednodušší.

Problémem může být styl programování pomocí callbacků, který se svou filozofií významně liší od jiných běžně užívaných stylů. Po osvojení stylu může začít být otravná a vyčerpávající nutnost ošetření chyby v rámci každého nového callback volání. U složitějších aplikací je tedy v podstatě nutnost přejít na tzv. asynchronní funkce, které tyto problémy mj. řeší.

Po zvládnutí zmíněných nástrah může Node.js posloužit jako mocný, jednoduchý a přímočarý nástroj vhodný pro vývoj zejména RESTových služeb.

6.4 Shrnutí

Jazyky vybrané pro implementaci patří mezi nejpopulárnější. Proto není překvapením, že v nich lze (více či méně) komfortně vyvíjet aplikace s využitím RESTu a SOAPu.

Obecně se pro vývoj menších a jednodušších serverových řešení jeví REST jako jasná volba. Zde vyniká zejména Python s jednoduchým, ale dobře se používajícím frameworkem *Flask*. REST lze pochopitelně využít i pro vývoj rozsáhlejších aplikací. Co není k dispozici, lze díky jednoduchosti RESTu snadněji doprogramovat, případně zvolit robustnější framework či jazyk.

Právě pro rozsáhlejší serverové projekty se nabízí zvážit využití SOAPu. Ten díky své komplexnosti v souvislosti s rozšiřujícími specifikacemi poskytuje řešení pro více běžně vyskytujících se problémů. V případě jeho použití je ovšem nutno klást velký důraz na výběr kvalitních technologií. To drasticky zmenšuje množinu použitelných jazyků (technologií), kde by v případě této práce obstála pouze Java.

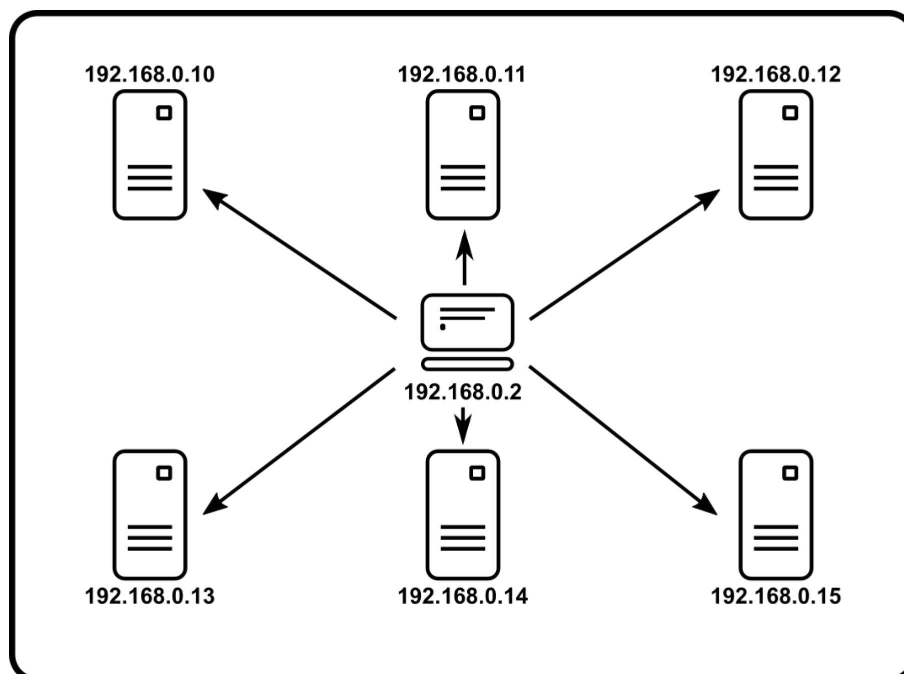
7 Experimenty

Experimenty představené v této kapitole budou demonstrovat efektivitu implementací webových služeb, popsaných v předešlých kapitolách. Bude popsáno testovací prostředí a testovací scénář. Výsledky experimentů budou prezentovány jak v rámci samotných implementací, tak i obecně v rámci typu webové služby.

7.1 Testovací prostředí

Testovací prostředí bylo tvořeno šesti samostatnými počítači komunikujícími v rámci izolované lokální sítě[22]. Jeden fungoval jako server, na kterém byla spuštěna instance jedné implementace v dané technologii. Zbylé počítače plnily úlohu klientů a simulovaly zátěž pomocí aplikace Apache JMeter. Klienti spolupracovali na principu master–slave. Master koordinoval celou simulaci, tzn. byl zodpovědný za běh celého testu a sběr výsledků. Slave klienti generovali samotnou zátěž na server.

Obrázek 9 ilustruje zmíněný master–slave princip testování, počítač s adresou `192.168.0.2` figuruje jako master.



Obrázek 9: Master–Slave princip využitý pro experimenty [59]

7.1.1 Konfigurace

Server disponoval procesorem Intel i5 760K se čtyřmi jádry přetaktovanými na 4GHz. Fungoval v terminálovém režimu s minimem aktivních služeb. Každé tech-

[22]Díky tomu bylo výrazně sníženo riziko ovlivnění testů možnou variabilní rychlostí sítě.

nologii (serverovým i klientským) bylo obecně nutné nastavit dostatečné množství paměti pro bezproblémovou funkčnost. Serverovým pak také zejména dostatečný rozsah databázových připojení pro tzn. connection pooling. Specifické konfigurace pro serverové technologie jsou popsány níže.

Java

Běhovým prostředím bylo Oracle JDK 1.8.0_261. Aplikace nasazena a testována na aplikačních serverech:

- *Payara 5.2020.5.*
- *WildFly 19.0.1.Final.*
- *WebSphere Liberty 20.0.0.3.*

Každému aplikační serveru bylo nutné přidat databázový ovladač a definovat datasource – nastavení přístupu do databáze.

Python

Python využíval interpretr naprogramovaný v jazyce C (CPython) ve verzi jazyka 3.8.5. Aplikace spouštěna WSGI serverem Gunicorn v rámci tzn. workerů. Každý worker spouští aplikaci jako samostatný izolovaný proces, tím dochází ke zvýšení výkonnosti aplikace.

NodeJs

Běhové prostředí NodeJs bylo ve verzi 10.19.0.

PHP

Aplikace pro svůj běh využívala server Apache 2.4.29 společně s FastCGI Process Manager (FPM) 7.4.11. FPM je alternativní implementace PHP navržená pro vyšší výkon. Jako MPM (Multi processing module)[23] byl použit MPM worker. PHP jako jediné neumožňuje tzn. connection pooling. Místo něj bylo ovšem využito stálého databázového připojení. Aplikace využívá tzv. preloadingu – zjednodušeně řečeno nástroj pro rychlejší vykonávání PHP kódu. V poslední řadě byly zvýšeny timeouty pro zpracování požadavků, jelikož předtím PHP nebylo schopné pro daný testovací scénář splnit testy bez chybových stavů.

PostgreSQL

Databázový stroj byl ve verzi 12.4. Nutností bylo zvýšit maximální počet souběžných databázových připojení.

Apache JMeter

JMeter využívá pro distribuované testování Java RMI (Remote Method Invocation). To umožňuje volat vzdáleně metody na jiném stroji. Ve výchozím nastavení je vyžadováno SSL (Secure Sockets Layer) zabezpečení s certifikátem.

[23] Technologie pro zpracovávání souběžných požadavků v rámci serveru Apache.

Pro testování v rámci izolované lokální sítě je však SSL lepší vypnout. Více na webové stránce JMeteru, která se zabývá distribuovaným testování: https://jmeter.apache.org/usermanual/remote-test.html.

7.2 Testovací scénář

Pro experimenty byl navrhnut testovací scénář. Data pro testování byla vygenerována a uložena do CSV[24] souboru. Stejná data byla využita v každém testu pro docílení shodných podmínek. Scénář obsahoval všechny operace (viz kapitola 6.1.1) v tomto pořadí:

- Validace publikace.
- Validace dokumentu.
- Přidání publikace.
- Přidání dokumentu (náhodně generované bajty velikosti: 1 KB - 5 KB).
- Získání publikace.
- Získání dokumentu.
- Úprava publikace.
- Smazání dokumentu.
- Získání všech publikací (omezeno na max. 3 naposledy přidané).
- Smazání publikace.

Konfigurace samotného testu v aplikaci Apache JMeter uvažovala simulaci 1000 uživatelů (reprezentováni vlákny). Ti se ke službě připojovali postupně v rámci 10 minut. Každý uživatel provedl výše zmíněné operace v 50-ti iteracích. Jak již bylo zmíněno v kapitole 7.1, do testu byly zahrnuty čtyři slave klienti. Na každém z nich souběžně probíhal izolovaně ten samý test, zatímco master klient agregoval výsledky. Server v rámci celého testu musel obsloužit celkem 2 milióny požadavků od 4 tisíc „uživatelů“. Všechny požadavky musely být validní, tedy žádný nesměl skončit chybou. Jak může vypadat výstup z testu ukazuje obrázek 10. Výstup je ve formátu CSV a jeden řádek reprezentuje naměřená data jednoho požadavek vyjma řádku prvního, který značí co jednotlivé „sloupce“ obsahují.

Je nutné zmínit, že pro zachování férovosti byl každý test spouštěn s nově vytvořenou databází a restartovaným serverem. Pro účely REST testů byla v implementacích vypnuta autentizace požadavků. Ta není pro SOAP implementována a tudíž by mohla zkreslovat výsledky testů.

[24] Textový soubor obsahující řádky, ve kterých jsou data oddělena čárkou či jiným symbolem.

```
timeStamp,elapsed,label,bytes,sentBytes
19:47:54.252,57,Validate publication,122,280
19:47:54.271,3,Validate document,122,252
19:47:54.275,26,Add publication,179,272
19:47:54.321,16,Add document,188,4169
19:47:54.338,6,Get publication,339,169
19:47:54.345,4,Get document,4193,178
19:47:54.350,5,Update publication,111,274
19:47:54.355,6,Delete document,111,200
19:47:54.362,3,Get all publications,420,167
19:47:54.366,6,Delete publication,111,191
```

Obrázek 10: Experimenty – ukázka možného výstupu po dokončení testu pomocí aplikace Apache JMeter

7.3 Výsledky

Testovací scénář byl vykonán pro každou technologii, kterou tvoří programovací jazyk a typ webové služby, v případě Javy také aplikační server. Pomocí JMeteru bylo tedy spuštěno celkem 12 testů.

V rámci každého testu byl pro každý zaslaný požadavek zaznamenán čas vrácení odpovědi v milisekundách a přenesené bajty (pro požadavek i jeho odpověď). Z těchto nasbíraných dat byla vypočítána minima, maxima, mediány, průměry a směrodatné odchylky[25]. To pro každou testovanou implementaci zvlášť i celkově v rámci typu webové služby (SOAP, REST). Pro každý test byl navíc zaznamenán čas po který probíhal a jak moc vytěžoval serverové CPU (Central Processing Unit[26]).

7.3.1 Jednotlivé implementace

Výsledky byly zpracovány z naměřených dat pro každou implementaci odděleně. Díky tomu, lze mezi nimi porovnat výsledky.

Tabulka 3 srovnává jednotlivé implementace z pohledu rychlosti vrácení odpovědi. Ukazuje, že suverénně nejrychlejší byl REST v NodeJs, který byl v prezentovaných hodnotách nejlepší, mimo medián také s výrazným náskokem. Navíc je při pohledu na jeho hodnoty zřejmé, že si udržoval kvalitní výkonost po celou dobu průběhu testu. Celkově rychlá byla Java, zejména ve spolupráci s aplikačním serverem WebSphere. Překvapením byla jeho výkonná implementace SOAPu. Nejenže byla nejrychlejší ve své kategorii, ale také výkonnostně převýšila většinu RESTových implementací, na rozdíl od ostatních aplikačních serverů, kde byl SOAP výrazně pomalejší (navzdory poměrně vyšší max. hodnotě). Python převedl obstojný výkon. Nicméně jeho i NodeJs SOAPová implementace byly znatelně pomalejší, než jejich RESTové protějšky. PHP ve výkonnostním srovnání selhalo a patří mu s velkým rozdílem všechny nejpomalejší výsledky (optimalizačním snahám navzdory). Vizuální srovnání lze nalézt na obrázku 11.

[25] V tabulkách prezentující čas vrácení odpovědi se vyskytuje číslo nula. To znamená, že požadavek byl zpracován tak rychle, že jej JMeter nedokázal korektně zaznamenat.

[26] Neboli procesor.

Všechny RESTové implementace měly efektivnější komunikaci co se týče přenesených bajtů. Ukazuje to tabulka 4, která srovnává jednotlivé implementace ze zmíněného hlediska. Drobné rozdíly mezi technologiemi stejné webové služby byly způsobeny rozdílnými HTTP hlavičkami, respektive rozdílnými podobami obálek v případě SOAPu (XML jmenné prostory apod.). Zde by poměrně jednoduchými optimalizacemi bylo možné ušetřit cenné bajty. Vizuální srovnání lze nalézt na obrázku 12.

Porovnání celkových časů běhů testů dopadlo obdobně jako v případě časů vrácení odpovědí. Nejrychlejší byla Java společně NodeJs. Zbylé testy byly pomalejší, včetně PHP, které bylo zdaleka nejpomalejší. Dokládá to tabulka 5 společně s obrázkem 13.

Z Tabulky 6 a obrázku 14 je zřejmé, že REST obecně vytěžoval méně CPU. Dobrých výsledků oproti konkurenci ostatních dosáhla Java na aplikačním serveru Websphere. Výborných pak NodeJs a to díky „asynchronnímu“ vykonávání programu a stvrdilo tak své předpoklady.

Tabulka 3: Experimenty – srovnání implementací: čas vrácení odpovědi (ms)

Technologie	**Min**	**Max**	**Medián**	**Průměr**	**Odchylka**
SOAP					
Java Payara	0	1090	113	121,56	45,85
Java WildFly	0	1158	121	116,92	34,67
Java WebSphere	0	1030	3	10,85	18,1
PHP	2	10756	2849	2582,33	1119,76
Python	1	4735	1114	1054,6	613,49
NodeJs	0	3070	482	565,72	477,53
REST					
Java Payara	0	1062	4	18,75	26,14
Java WildFly	0	504	4	22,23	30,78
Java WebSphere	0	649	3	10,51	17,39
PHP	2	8719	2702	2457,94	1077,4
Python	1	2110	405	428,19	291,7
NodeJs	0	211	3	2,76	2,04

Tabulka 4: Experimenty – srovnání implementací: přenesené bajty

Technologie	**Min**	**Max**	**Medián**	**Průměr**	**Odchylka**
			SOAP		
Java Payara	927	7745	1081	1872,84	1741,07
Java WildFly	765	7582	919	1710,58	1741,01
Java WebSphere	794	7611	948	1743,84	1739,91
PHP	875	7676	1054	1840,56	1731,75
Python	849	7705	1023	1836,57	1737
NodeJs	873	7652	1016	1804,26	1729,8
			REST		
Java Payara	393	5530	525	1115,24	1309,52
Java WildFly	255	5416	410	994,14	1313,32
Java WebSphere	326	5444	442	1036,01	1306,24
PHP	328	5520	516	1092,9	1312,43
Python	344	5476	473	1063,32	1301,09
NodeJs	302	5449	434	1041,52	1313,34

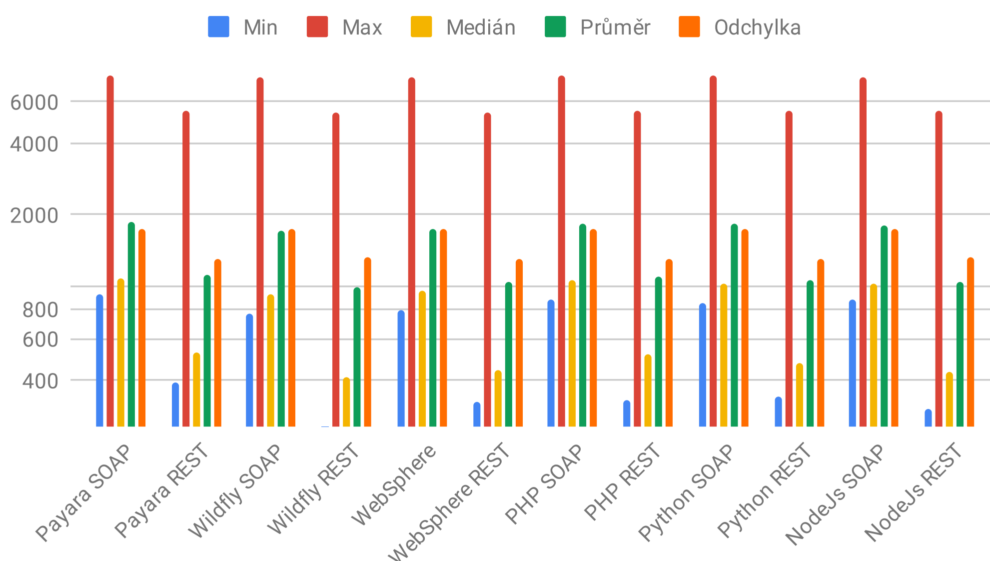
Tabulka 5: Experimenty – srovnání implementací: časy trvání testů (mm:ss)

Technologie	**SOAP**	**REST**
Java Payara	10:56	10:02
Java WildFly	10:37	10:02
Java WebSphere	10:03	10:02
PHP	31:52	30:47
Python	19:21	14:38
NodeJs	14:53	10:02

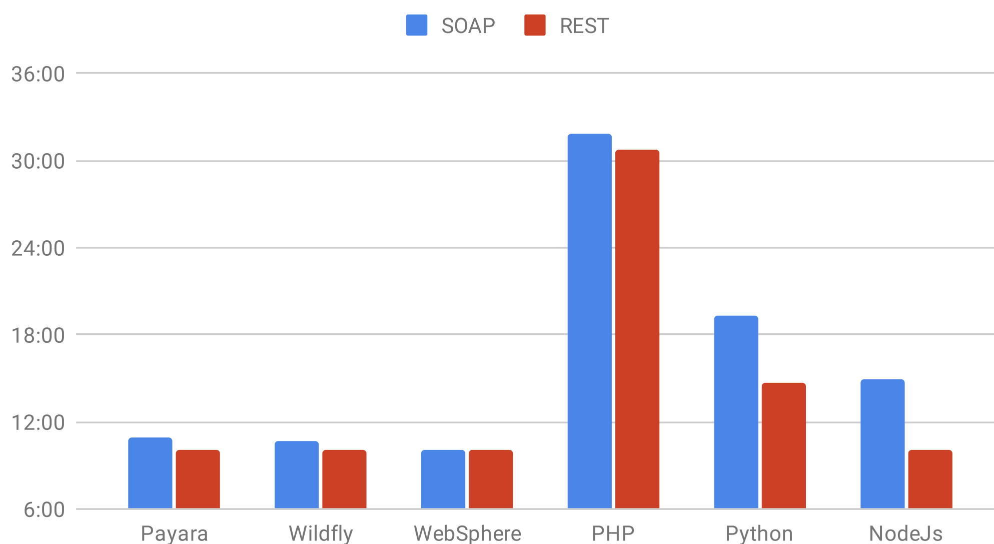
Tabulka 6: Experimenty – srovnání implementací: průměrné vytížení CPU (%)

Technologie	**SOAP**	**REST**
Java Payara	99,1	87,55
Java WildFly	98,9	82,08
Java WebSphere	79,04	76
PHP	99,89	99,92
Python	97,29	94,88
NodeJs	58,17	55,73

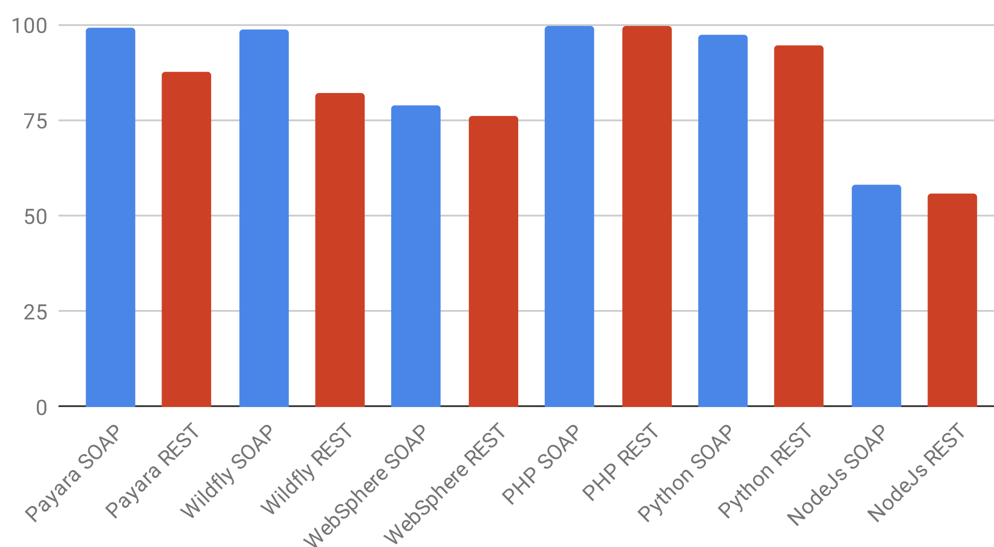
Obrázek 11: Experimenty – srovnání implementací: čas vrácení odpovědi (ms)



Obrázek 12: Experimenty – srovnání implementací: přenesené bajty



Obrázek 13: Experimenty – srovnání implementací: časy trvání testů (mm:ss)



Obrázek 14: Experimenty – srovnání implementací: průměrné vytížení CPU (%)

7.3.2 Webové služby

Všechny výsledky byly rozděleny do dvou kategorií dle typu webové služby. Výstupy z jednotlivých testů byly tedy v rámci kategorie sloučeny do jedné množiny a výpočty probíhaly nad ní. Díky tomu lze porovnat výsledky obecněji z pohledu typu webové služby.

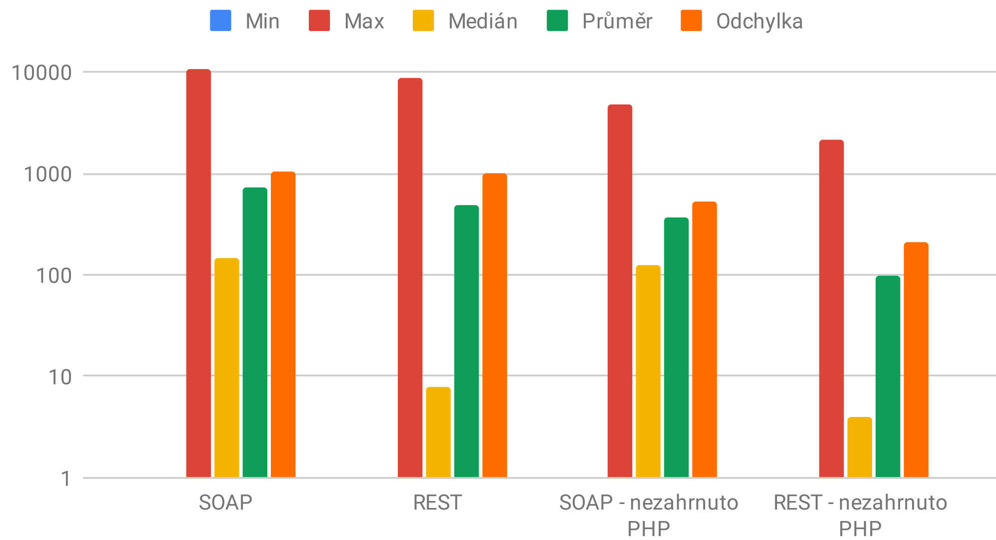
Tabulka 7 a obrázky 15, 16 zobrazují výsledky testů v rámci kategorií webových služeb. Ukazují, že srovnání vychází lépe pro REST, který je efektivnější než-li SOAP, jelikož jasně zvítězil ve všech prezentovaných hodnotách. PHP dosáhlo v oblasti rychlosti zdaleka nejhoršího výsledku a z hlediska celkového souboru dat webových služeb obsahuje velké množství extrémních hodnot (maximálních). To ovlivňuje zejména průměr a směrodatnou odchylku. Proto byly pro zajímavost vypočítány i výsledky bez zahrnutí PHP.

Zajímavé by mohlo být obecné srovnání rychlostí odpovědí dle typu webové služby v závislosti na pořadí odpovědi[27] vynesené do grafu. To by mohlo být vypočítáno zprůměrováním rychlostí implementací na stejné pozici v rámci své kategorie webové služby. Nicméně zobrazení 2 miliónů hodnot by působilo v grafu nepřehledně. Problém lze vyřešit vybráním reprezentantů (vzorků) vzorkováním. Kdy pro získání 5 tisíc vzorků rozdělíme soubor dat na menší soubory o 400 prvcích a tyto prvky zprůměrujeme a tím získáme daný vzorek. Na obrázku 17 jsou data vynesena do grafu. Osa x určuje pořadí vzorku a osa y jeho průměrnou rychlost vrácení odpovědi v milisekundách. Jak pro SOAP i REST hodnoty postupně rostou kvůli zvyšující se zátěži na serveru. Poté jsou u RESTu nějakou dobu zhruba konstantní, zatímco u SOAPu stále rostou. Až nakonec se snižující se zátěží serveru hodnoty u obou klesají. Lze tedy usoudit, že REST se lépe vypořádával se zátěží.

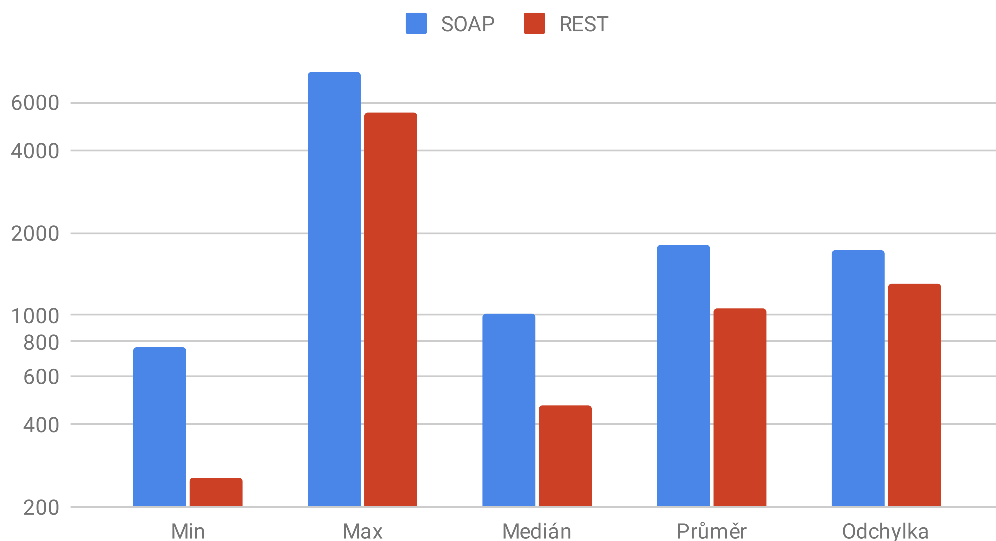
Tabulka 7: Experimenty – srovnání SOAP a REST

Technologie	**Min**	**Max**	**Median**	**Průměr**	**Odchylka**
		Čas vrácení odpovědi (ms)			
SOAP	0	10756	144	742	1055,76
REST	0	8719	8	490,06	1002,71
SOAP bez PHP	0	4735	124	373,93	523,39
REST bez PHP	0	2110	4	96,49	212,03
		Přenesené bajty (B)			
SOAP	765	7745	1020	1801,44	1737,7
REST	255	5530	468	1057,19	1309,92

[27] Tedy postupně od první až po poslední odpověď.



Obrázek 15: Experimenty – srovnání dle typu webové služby: čas vrácení odpovědi (ms)



Obrázek 16: Experimenty – srovnání dle typu webové služby: přenesené bajty

Obrázek 17: Experimenty – srovnání obecné rychlosti odpovědi v průběhu testů dle typu webové služby pomocí průměrování a vzorkování

7.4 Shrnutí

Výsledky experimentu prokazují lepší obecnou efektivitu stylu REST oproti protokolu SOAP. Ukázalo se, že sice SOAP může RESTu konkurovat, ale je třeba mít k dispozici technologie pro kvalitní (efektivní) implementaci SOAPu. To z výsledků splňuje pouze Java, zejména ve spolupráci s aplikačním serverem WebSphere.

V reálném provozu by se pravděpodobně efektivita RESTu více projevila. To například v méně spolehlivé počítačové síti. SOAP pro komunikaci potřebuje obecně větší množství dat a v pomalé – nespolehlivé síti by se tento rys zřejmě projevil na jeho výkonnosti více než u RESTu.

Co se týče jednotlivých programovacích jazyků, tak výkonnostně vyčnívala Java a RESTová implementace pomocí NodeJs, naopak PHP v konkurenci propadlo.

8 SOAP proti REST

V předchozích kapitolách byly představeny webové služby SOAP a REST z několika aspektů. Tato kapitola představuje krátký souhrn a celkové porovnání obou přístupů z několika hledisek.

8.1 Standard proti stylu

Standardizace jistě není špatný proces. A některé standardy okolo protokolu SOAP jsou určitě přínosné. Nicméně celkově je vytvořen obrovský, poměrně chaotický a někdy ne úplně jednoduše pochopitelný systém standardů. Díky čemuž se jeví jako složitý a těžkopádný k jednoduchému použití.

Naproti tomu REST definuje poměrně jednoduchá omezení, některá i dobře známa, protože vyplývají snad z nejznámějšího protokolu pro komunikaci vůbec, tedy HTTP. Daní za jednoduchost může být absence vodítek pro řešení některých problémů, například bezpečnosti, kterou REST neřeší vůbec. Pro nezkušeného vývojáře proto může být někdy obtížné nalézt optimální a správné řešení.

8.2 Návrh aplikace

U návrhu aplikace z pohledu API má pro SOAP zcela monopolní postavení WSDL. To je dobře a i srozumitelně navrženo a s jeho pomocí se dá navrhnout jednoduše i obrovské API. Nevýhodou naopak může být absence standardizace běžně se vyskytujících operací u většiny aplikací – CRUD (create, read, update, delete). Tyto operace se mohou jmenovat libovolně a je proto nutné vždy využít WSDL. Celkově je však návrh aplikace pro SOAP jednoduchý a to zejména proto, že zachovává typ přístupu osvědčeného z běžných programovacích jazyků.

Návrh většího API pro REST může být někdy dost namáhavé. Je třeba se zamyslet nad podobou identifikace zdroje, použitím správné metody a nezřídka, také nad samotnými hlavičkami. To je však spíše problém mohutnějších API. V případě menších API může být tento proces podstatně přímočařejší, pokud se autor drží běžného modelu použití metod (CRUD). Pro REST existují také formáty pro formální popis API jako je WSDL. Je jich však více a je otázkou pro jaký se rozhodnout. Ortodoxní zastánci přesného výkladu filozofie RESTu tvrdí, že tyto nástroje jdou proti ní. To z důvodu, že REST by (přísně vzato) měl ctít HATEOAS (viz třetí úroveň z 3.3.2) a pro prozkoumání implementovaného API by potom měl stačit pouze základní zdroj, který poskytuje odkazy na další atd.

8.3 Implementace

V případě implementace dost záleží na použitém programovacím jazyce, případně na dostupnosti kvalitních nástrojů. Například v případě použití Javy lze serverovou i klientskou část jednoduše vygenerovat z popisu WSDL, ten poté stačí pouze začlenit do projektu, lehce nakonfigurovat a máme z pohledu komunikace

mezi webovými službami v podstatě hotovo. Opačný postup, tedy nejdříve programování a poté generování WSDL, je také jednoduchý a dostatečně rychlý.

V případě absence kvalitních knihoven či nástrojů se ovšem implementace SOAPu může změnit z jednoduché záležitosti ve velký problém. Protokol je složitý a proto také například řešení problému jak zaslat velký binární soubor co nejoptimálněji může při absenci podpory ze strany technologie pokazit nejednomu vývojáři den.

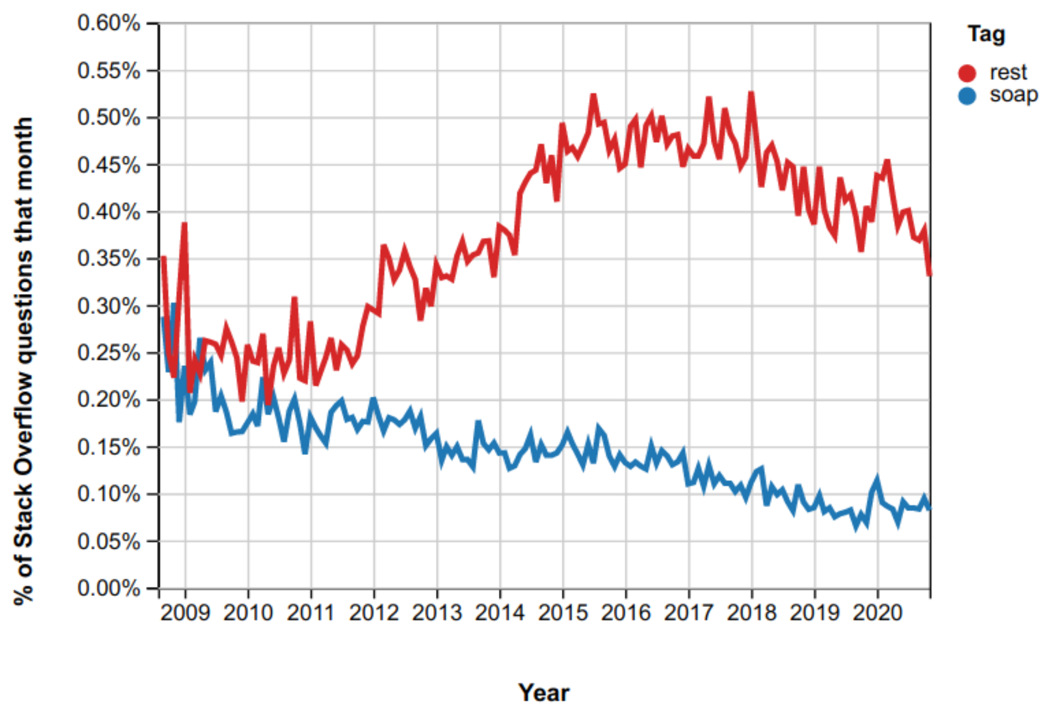
V teoretické rovinně je tedy implementace REST API složitější než v případě SOAPu. Je nutné řešit více problémů samostatně, ovšem opět výběr vhodného jazyka a technologie tyto rozdíly dostatečně smazává. Navíc vzhledem k jednoduchosti a otevřenosti lze v případě nedostupnosti či specifickým požadavkům zvládnout implementaci vlastními silami daleko snáze. Moderní jazyky (Go, Ruby, Rust) navíc dnes poskytují lepší komfort pro práci s REST oproti SOAPu, kde je většinou k dispozici pouze použitelný klient.

Samotné generování kódu, i když na to nástroje existují, není tak pohodlné a vygenerovaný kód potřebuje větší péči. Stejně tak jako v případě generování kvalitní API dokumentace je nutné do programů vkládat k tomuto účelu metadata.

8.4 Využitelnost

SOAP se vzhledem ke své standardizaci a těžkopádnosti dnes používá málo. Nalézá využití zejména pro robustní spíše monolitické aplikace. Případně tam, kde je důraz na extrémní zabezpečení nebo je nutné využít jiného transportního protokolu než je HTTP.

V dnešní době se daleko častěji sází na REST zejména proto, že je jednodušší, flexibilnější a moderní aplikace se často vyvíjí formou mikroslužeb. U nich je požadavek na jejich škálovatelnost, odolnost proti výpadkům, rychlé nasazení a také na efektivitu komunikace, na kterou jsou kladeny vyšší požadavky. To proto, že je efektivní komunikace potřeba daleko více vzhledem k vyššímu počtu malých izolovaných služeb. Výkonnostní benefity také jistě ocení uživatelé slabších zařízení jako jsou například mobilní telefony. REST prostě dnes daleko lépe vyhovuje požadavků kladeným na klasické aplikace a SOAP je využíván zřídka například v případně komunikace se starou aplikací. Nadvládu RESTu lze také doložit statistikou dotazování na oba přístupy mezi vývojáři z respektovaného fóra Stack Overflow. Ta na obrázku 18 ukazuje, že REST v rámci posledních 10 let převyšuje SOAP, v posledních 5 letech pak výrazně.



Obrázek 18: Srovnání popularity SOAP a REST dle Stack Overflow, zdroj: [60]

Závěr

Obsahem práce je popis a porovnání webových služeb, postavených na protokolu SOAP nebo architektonickém stylu REST, které využívají HTTP. V teoretické části byly oba přístupy formálně popsány, stejně jako jejich ekosystémy, skládající se z podpůrných technologií a nástrojů.

V rámci praktické části byla implementována jednoduchá aplikace pomocí vybraných technologií v programovacích jazycích Java, PHP, Python a JavaScriptu za podpory prostředí Node.js. Aplikace demonstruje rozdílné přístupy k obou typům služeb. Na implementovaných řešeních serverové části byly provedeny experimenty, prokazující lepší efektivitu RESTových služeb vhledem k rychlosti odezvy a množství přenesených dat v rámci komunikace.

V samotném závěru práce bylo provedeno celkové srovnání obou přístupů. Ukázalo se, že REST odpovídá lépe požadavkům kladeným na moderní aplikace, a proto je dnes využívanější než SOAP.

Conclusions

The content of this thesis is a description and comparison of web services based on SOAP protocol or architectural style REST, which use HTTP. In the theoretical part are both approaches formally described, as well as their ecosystems consisting of supporting technologies and tools.

In the practical part, a simple application was implemented via selected technologies in Java, PHP, Python and JavaScript programming languages with the support of the Node.js environment. The application demonstrates different approaches to both types of services. Experiments made on implemented solutions of the server part proved better efficiency of REST services regard to the speed of response and the amount of transferred data within the communication.

An overall comparison of both approaches can be found at the very end of the thesis. The REST proved better capabilities to meet the demands of modern applications and is, therefore, more used today than SOAP.

A Obsah přiloženého DVD

`bin/`

Soubory pro nasazení testovacích aplikací. Pro Javu soubory WAR, pro ostatní jazyky soubory ZIP. K dispozici jsou také docker obrazy ve formátu TAR.

`doc/`

Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

`src/`

Kompletní zdrojové kódy: aplikací, definice API a docker obrazů.

`readme.txt`

Instrukce pro nasazení testovacích aplikací ve formátu Markdown.

Navíc DVD obsahuje:

`conf/`

Konfigurační soubory.

`lib/`

Ovladač k databázi PostreSQL pro jazyk Java.

`test/`

Testovací plány pro Apache JMeter použité v rámci experimentů. Skripty v jazyce Python pro generování testovacích dat a zpracování výstupných dat z testů. Testovací kolekce pro nástroj Postman.

Literatura

[1] TIŠNOVSKÝ, Pavel. *Mikroslužby: moderní aplikace využívající známých konceptů* [online]. 2019 [cit. 2020-12-13]. Dostupný z: ⟨https://www.root.cz/clanky/mikrosluzby-moderni-aplikace-vyuzivajici-znamych-konceptu/⟩.

[2] FIELDING, Roy T.; RESCHKE, Julian. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [online]. 2014 [cit. 2020-12-13]. 101 s. Request for Comments. RFC 7231⟨https://rfc-editor.org/rfc/rfc7231.txt⟩.

[3] MOZILLA AND INDIVIDUAL CONTRIBUTORS. *Evolution of HTTP* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP⟩.

[4] BELSHE, Mike; PEON, Roberto; THOMSON, Martin. *Hypertext Transfer Protocol Version 2 (HTTP/2)* [online]. 2015 [cit. 2020-12-13]. 96 s. Request for Comments. RFC 7540⟨https://rfc-editor.org/rfc/rfc7540.txt⟩.

[5] MOZILLA AND INDIVIDUAL CONTRIBUTORS. *An overview of HTTP* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview⟩.

[6] FIELDING, Roy T.; RESCHKE, Julian. *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests* [online]. 2014 [cit. 2020-12-13]. 28 s. Request for Comments. RFC 7232⟨https://rfc-editor.org/rfc/rfc7232.txt⟩.

[7] FIELDING, Roy T.; RESCHKE, Julian. *Hypertext Transfer Protocol (HTTP/1.1): Authentication* [online]. 2014 [cit. 2020-12-13]. 19 s. Request for Comments. RFC 7235⟨https://rfc-editor.org/rfc/rfc7235.txt⟩.

[8] MOZILLA AND INDIVIDUAL CONTRIBUTORS. *HTTP headers* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers⟩.

[9] MOZILLA AND INDIVIDUAL CONTRIBUTORS. *Common MIME types* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types⟩.

[10] MOZILLA AND INDIVIDUAL CONTRIBUTORS. *HTTP authentication* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication⟩.

[11] *ABOUT W3C*. [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://www.w3.org/Consortium/⟩.

[12] HAAS, Hugo; FERRIS, Christopher; MCCABE, Francis aj. *Web Services Architecture* [online]. 2004 [cit. 2020-12-13]. Dostupný z: ⟨http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/⟩.

[13] BOX, Don; EHNEBUSKE, David; KAKIVAYA, Gopal aj. *Simple Object Access Protocol (SOAP) 1.1* [online]. 2000 [cit. 2020-12-13]. Dostupný z: ⟨https://www.w3.org/TR/2000/NOTE-SOAP-20000508/⟩.

[14] GUDGIN, Martin; HADLEY, Marc; MENDELSOHN, Noah aj. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)* [online]. 2007 [cit. 2020-12-13]. Dostupný z: ⟨https://www.w3.org/TR/soap12-part1/⟩.

[15] SKONNARD, Aaron. *XML – pohotová referenční příručka*. Praha: Grada, 2006. 342 s. ISBN 8024709724.

[16] ORACLE CORPORATION AND/OR ITS AFFILIATES. *What Is in a Message? (The Java EE 5 Tutorial)* [online]. 2010 [cit. 2020-12-13]. Dostupný z: ⟨https://docs.oracle.com/cd/E19575-01/819-3669/bnbhj/index.html⟩.

[17] *Web service standards – SOAP*. [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://www.ibm.com/support/knowledgecenter/en/SSCLKU_7.5.5/org.eclipse.jst.ws.doc.user/concepts/csoap.html⟩.

[18] MOZILLA AND INDIVIDUAL CONTRIBUTORS. *Base64 encoding and decoding* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://developer.mozilla.org/en-US/docs/Web/API/WindowBase64/Base64_encoding_and_decoding⟩.

[19] ANGELOV, Dimitar; FERRIS, Christopher; KARMARKAR, Anish aj. *SOAP 1.1 Binding for MTOM 1.0* [online]. 2006 [cit. 2020-12-13]. Dostupný z: ⟨https://www.w3.org/Submission/soap11mtom10/⟩.

[20] CHRISTENSEN, Erik; CURBERA, Francisco; MEREDITH, Greg; WEERAWARANA, Sanjiva. *Web Services Description Language (WSDL) 1.1* [online]. 2001 [cit. 2020-12-13]. Dostupný z: ⟨https://www.w3.org/TR/wsdl.html⟩.

[21] CHINNICI, Roberto; MOREAU, Jean-Jacques; RYMAN, Arthur; WEERAWARANA, Sanjiva. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language* [online]. 2007 [cit. 2020-12-13]. Dostupný z: ⟨https://www.w3.org/TR/wsdl20/⟩.

[22] PANZIERA, Luca. *Service Matchmaking:Exploiting the Web*. 2013. Dizertační práce. Università degli studi di Milano - Bicocca.

[23] IBM®IBM KNOWLEDGE CENTER. *Literal vs. Encoded, RPC– vs. Document-Style* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://www.ibm.com/support/knowledgecenter/en/SSB27H_6.2.0/fa2ws_ovw_soap_syntax_lit.html⟩.

[24] BUTEK, Russell. *Which style of WSDL should I use?* [online]. 2005 [cit. 2020-12-13]. Dostupný z: ⟨https://www.ibm.com/developerworks/library/ws-whichwsdl/index.html⟩.

[25] OASIS. *Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)* [online]. 2004 [cit. 2020-12-13]. Dostupný z: ⟨https://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf⟩.

[26] BUSTAMANTE, Michele Leroux. *Making Sense of all these Crazy Web Service Standards* [online]. 2007 [cit. 2020-12-13]. Dostupný z: ⟨https://www.infoq.com/articles/ws-standards-wcf-bustamante/⟩.

[27] FIELDING, Roy T. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. Dizertační práce. University of California, Irvine.

[28] MASSE, Mark. *REST API Design Rulebook*. 2011. 114 s. ISBN 1449310508.

[29] FOWLER, Martin. *Richardson Maturity Model: steps toward the glory of REST* [online]. 2010 [cit. 2020-12-13]. Dostupný z: ⟨https://martinfowler.com/articles/richardsonMaturityModel.html⟩.

[30] RICHARDSON, Leonard. *Act Three: The Maturity Heuristic* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://restfulapi.net/richardson-maturity-model/⟩.

[31] JWT.IO. *Introduction to JSON Web Tokens* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://jwt.io/introduction/⟩.

[32] MONUS, Anna. *SOAP vs REST vs JSON - a 2020 comparison* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://raygun.com/blog/soap-vs-rest-vs-json/⟩.

[33] UPWORK® GLOBAL INC. *SOAP vs. REST: A Look at Two Different API Styles* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://www.upwork.com/resources/soap-vs-rest-a-look-at-two-different-api-styles⟩.

[34] KOSEK, Jiří. *Využití webových služeb a protokolu SOAP při komunikaci: Kapitola 4. Komunikační infrastruktura* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://www.kosek.cz/diplomka/html/websluzby.html⟩.

[35] LINKER, Beth. *Introduction to WS-Addressing* [online]. 2005 [cit. 2020-12-13]. Dostupný z: ⟨https://www.oracle.com/technical-resources/articles/enterprise-architecture/ws-addressing-intro.html⟩.

[36] DAVIS, Doug; KARMARKAR, Anish; PILZ, Gilbert; WINKLER, Steve; YALÇINALP, Ümit. *Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.2* [online]. 2009 [cit. 2020-12-13]. Dostupný z: ⟨https://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.html⟩.

[37] *WS-Transaction*. [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://www.ibm.com/support/knowledgecenter/SSEQTP_8.5.5/com.ibm.websphere.base.doc/ae/twbs_wstx_learning.html⟩.

[38] SMARTBEAR SOFTWARE. *SoapUI* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://www.soapui.org/⟩.

[39] OPENAPI INITIATIVE. *OpenAPI Specification* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://github.com/OAI/OpenAPI-Specification⟩.

[40] *API Blueprint. A powerful high-level API description language for web APIs.* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://apiblueprint.org/⟩.

[41] *The simplest way to design APIs.* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://raml.org/⟩.

[42] SMARTBEAR SOFTWARE. *About Swagger* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://swagger.io/about/⟩.

[43] ORACLE AND/OR ITS AFFILIATES. *How Apiary works* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://apiary.io/how-apiary-works⟩.

[44] *OData - the best way to REST*. [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://www.odata.org/⟩.

[45] KUTÁČ, Pavel. *Postman, 1. část* [online]. 2018 [cit. 2020-12-13]. Dostupný z: ⟨https://www.kutac.cz/weby-a-vse-okolo/postman-1-cast⟩.

[46] APACHE SOFTWARE FOUNDATION. *Apache JMeter* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://jmeter.apache.org/⟩.

[47] ORACLE AND/OR ITS AFFILIATES. *Creating a Simple Web Service and Clients with JAX-WS* [online]. 2017 [cit. 2020-12-13]. Dostupný z: ⟨https://javaee.github.io/tutorial/jaxws002.html⟩.

[48] ORACLE AND/OR ITS AFFILIATES. *Building RESTful Web Services with JAX-RS* [online]. 2017 [cit. 2020-12-13]. Dostupný z: ⟨https://javaee.github.io/tutorial/jaxrs.html⟩.

[49] THE PHP GROUP. *SOAP* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://www.php.net/manual/en/book.soap.php⟩.

[50] JOSH LOCKHART, ANDREW SMITH, ROB ALLEN, PIERRE BÉRUBÉ, AND THE SLIM FRAMEWORK TEAM. *Slima a micro framework for PHP* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://www.slimframework.com/⟩.

[51] DOWLING, Michael. *Guzzle Documentation* [online]. [cit. 2020-12-13]. Dostupný z: ⟨http://docs.guzzlephp.org/en/stable/⟩.

[52] *What is Spyne?* [online]. [cit. 2020-12-13]. Dostupný z: ⟨http://spyne.io⟩.

[53] MICHAEL VAN TELLINGEN. *Zeep: Python SOAP client* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://python-zeep.readthedocs.io/en/master/⟩.

[54] *Welcome to Flask – Flask Documentation (1.1.x)*. [online]. [cit. 2020-12-13]. Dostupný z: ⟨http://flask.palletsprojects.com/en/1.1.x/⟩.

[55] *What is WSGI?* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://wsgi.readthedocs.io/en/latest/what.html⟩.

[56] *Node.js*. [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://nodejs.org/en/⟩.

[57] PULIM, Vinay. *Soap* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://github.com/vpulim/node-soap#readme⟩.

[58] *Axios*. [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://github.com/axios/axios⟩.

[59] APACHE SOFTWARE FOUNDATION. *Apache JMeter Distributed Testing Step-by-step* [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.html⟩.

[60] *Stack Overflow Trends*. [online]. [cit. 2020-12-13]. Dostupný z: ⟨https://insights.stackoverflow.com/trends?tags=soap,rest⟩.