Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Možnosti dynamického definování webového API

Bakalářská práce

Autor: Jiří Falta
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Hradec Králové duben 2023

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

.....

V Hradci Králové dne 24. 4. 2023 Jiří Falta, v.r.

Poděkování:

Děkuji vedoucímu bakalářské práce panu doc. Mgr. Tomášovi Kozlovi, Ph.D. za jeho cenné poznatky a odborné vedení práce. Zároveň bych rád poděkoval své rodině a přítelkyni za poskytnutou podporu během studia.

Anotace

Tato bakalářská práce se zaměřuje na možnosti využití dynamicky definovaného webového API, jakožto nadstavbu nad architekturu REST. Při použití dynamického definování je umožněno vytváření nových koncových bodů API za běhu aplikace bez nutnosti nasazování nové verze. Součástí bakalářské práce je seznámení s využívaným internetovým protokolem HTTP. Následně jsou popsány nejčastější přístupy k webovému API, mezi které lze zařadit REST, SOAP a GraphQL. Popsány jsou i formáty dat ke komunikaci s webovými službami. Další část práce se věnuje již samotnému dynamickému definování v teoretické rovině, včetně popsání požadavků a schématu. Nedílnou součástí je i základní implementovaná ukázka a závěrečné teoretické porovnání výhod a nevýhod dynamicky definovaného webového API. Výsledkem práce je návrh nadstavby dynamického API pro REST architekturu.

Annotation

Title: Possibilities of Dynamic Web API Definition

This bachelor thesis focuses on the possibilities of using a dynamically defined web API as a superstructure over the REST architecture. Using dynamic definition allows the creation of new API endpoints at runtime without the need to deploy a new version. The bachelor thesis includes an introduction to the HTTP web protocol that is used. The most common approaches to web APIs are then described, which include REST, SOAP and GraphQL. Data formats for communicating with web services are also described. The next part of the paper deals with dynamic definition itself in theoretical terms, including a description of the requirements and schema. A basic implemented demonstration and a final theoretical comparison of the advantages and disadvantages of a dynamically defined web API are also an integral part. The result of this work is a proposal for a dynamic API superstructure for the REST architecture.

Obsah

1 Úvod ... 1

2 Hypertext Transfer Protocol ... 2

2.1 Uniform Resource Locator ... 2

2.2 Verze ... 3

2.2.1 HTTP/0.9 ... 3

2.2.2 HTTP/1.0 ... 3

2.2.3 HTTP/1.1 ... 4

2.2.4 HTTP/2 ... 6

2.2.5 HTTP/3 ... 7

2.3 Hypertext Transfer Protocol Secure ... 8

2.4 Struktura ... 9

2.4.1 Hlavičky ... 11

2.4.2 Metody ... 12

2.4.3 Stavové kódy ... 13

3 Webové API ... 15

3.1 Formáty dat ... 15

3.1.1 JSON ... 15

3.1.2 XML ... 16

3.2 SOAP ... 17

3.2.1 Zprávy ... 17

3.2.2 Jmenné prostory ... 20

3.2.3 WSDL ... 20

3.3 REST ... 22

3.3.1 Pravidla ... 22

3.3.2 Webové služby ... 24

3.3.3 Zabezpečení 25
3.3.4 Ukázky 25
3.4 GraphQL 27
4 Dynamicky definované API 31
4.1 Využití 31
4.2 Požadavky 31
4.2.1 Technologické požadavky 32
4.2.2 Požadavky na API 33
4.2.3 Požadavky na uložiště 34
4.3 Napodobované funkcionality 35
4.3.1 Možnost seskupování 35
4.3.2 Problematika verzování 35
4.3.3 Označení zastaralým 36
4.4 Schéma 37
4.4.1 Endpointy 38
4.4.2 Endpoint akce 40
4.4.3 Parametry 42
4.4.4 Datové typy 44
4.4.5 Skupiny endpointů 46
4.5 Plugin systém 46
4.5.1 Průběh vývoje 47
4.5.2 Společné rozhraní 47
4.5.3 Výhody a nevýhody 47
4.6 SOAP 48
5 Implementace 49
5.1 Architektura 49

5.2 Požadavky ke spuštění 51
5.3 Databáze 51
5.4 Ovládání 52
5.5 Middleware 53
5.6 Důležité služby 57
5.7 Zabezpečení 58
5.8 Praktické využití 58
6 Porovnání klasické vs. dynamické API 60
6.1 Výhody 60
6.2 Nevýhody 61
6.3 Výkon 61
7 Závěr 63
8 Seznam použité literatury 64
9 Přílohy 68

Seznam obrázků

Obrázek 1 - Požadavek v HTTP/1.0, přejato z [5], upraveno autorem 3
Obrázek 2 - Odpověď v HTTP/1.0, přejato z [5], upraveno autorem 4
Obrázek 3 - Požadavek a odpověď v HTTP/1.1 [5] 5
Obrázek 4 – Server Push [11] 7
Obrázek 5 – Porovnání HTTP/2 vs HTTP/3 [15] 8
Obrázek 6 – Porovnání HTTP metod, přejato z [7], upraveno autorem 13
Obrázek 7 – Ukázka JSON formátu [27] 15
Obrázek 8 – Ukázka XML formátu, vlastní zpracování 16
Obrázek 9 – Struktura SOAP zprávy [32] 18
Obrázek 10 – Schéma návrhu dynamického API, vlastní zpracování 38
Obrázek 11 – Architektura ukázkové implementace, vlastní zpracování 50
Obrázek 12 – Struktura ukázkové databáze, vlastní zpracování 52
Obrázek 13 – Průběh požadavků s middlewary, vlastní zpracování 54

Seznam zdrojových kódů

Zdrojový kód 1 – Ukázkový HTTP požadavek, vlastní zpracování 9
Zdrojový kód 2 – Ukázková HTTP odpověď, vlastní zpracování 10
Zdrojový kód 3 – Ukázkový RESTful GET požadavek, vlastní zpracování 25
Zdrojový kód 4 – Ukázkový RESTful POST požadavek, vlastní zpracování 26
Zdrojový kód 5 – Ukázkový RESTful DELETE požadavek, vlastní zpracování 26
Zdrojový kód 6 – Ukázkové GraphQL schéma, vlastní zpracování 28
Zdrojový kód 7 – Ukázkový GraphQL dotaz, vlastní zpracování 29
Zdrojový kód 8 – Ukázková GraphQL mutace, vlastní zpracování 30
Zdrojový kód 9 – Ukázkový middleware k API klíčům, vlastní zpracování 55
Zdrojový kód 10 – Ukázka registrace endpoint, vlastní zpracování 56
Zdrojový kód 11 – Ukázková metoda API služby, vlastní zpracování 58

Zkratky

API..... Application Programming Interface

CRUD..... Create, Read, Update, Delete

HTML..... Hypertext Markup Language

HTTP..... Hypertext Transfer Protocol

HTTPS..... Hypertext Transfer Protocol Secure

IS..... Informační systém

JSON..... JavaScript Object Notation

QUIC..... Quick UDP Internet Connections (neoficiálně)

REST..... Representational state transfer

RFC..... Request for Comments

SOAP..... Simple Object Access Protocol

SSL..... Secure Sockets Layer

TCP..... Transmission Control Protocol

TLS..... Transport Layer Security

UDP..... User Datagram Protocol

URL..... Uniform Resource Locator

XML..... Extensible Markup Language

1 Úvod

Webové služby tvoří jeden ze základních pilířů tvorby webových aplikací. Tyto služby lze také nalézt pod pojmem webové API neboli aplikační programové rozhraní. Pod tímto označením se skrývá rozhraní s metodami pro komunikaci mezi informačními systémy. Systém, který poskytuje API, obvykle obsahuje fixní metody (koncové body). V případě nutnosti změn v koncových bodech či při vytvoření nových je nutná kompilace systému a jeho následné nové nasazení na server. Cílem této bakalářské práce je navržení možností k vytvoření dynamického API, kde bude možné definovat nové koncové body za běhu systému bez nutnosti nasazení celé nové verze informačního systému na server.

Teoretická část této bakalářské práce se věnuje internetovému protokolu HTTP, který je využíván pro komunikaci s webovými službami. V rámci HTTP jsou popsány staré verze protokolu i nejnovější verze s označením HTTP/3, která procházela standardizací během tvorby této práce (rok 2022/2023). V další části jsou popsány nejvyužívanější formáty dat pro komunikaci s webovými API. Obdobně jsou vysvětleny nejčastější přístupy k API, mezi které se řadí SOAP, REST a GraphQL.

V rámci praktické části jsou navrženy možnosti pro umožnění dynamického definování API. Tyto možnosti jsou nastaveny skrze technologické požadavky, požadavky na API a uložiště. Dále je zde popsána funkcionalita a schéma navrhované architektury. Součástí praktické části je i popis jednoduché implementace systému s dynamickým API.

V závěru práce je porovnáno klasického webové API oproti dynamické variantě. Součástí porovnání jsou vyjádřeny výhody i nevýhody dynamického definování. Zároveň jsou popsány parametry ovlivňující výkon API, včetně možností na zlepšení výkonu.

2 Hypertext Transfer Protocol

Hypertext Transfer Protokol (HTTP) je bezstavový protokol aplikační vrstvy pro distribuované informační systémy. Nejnovější definice je standardizována v dokumentu RFC-9110. [1] Protokol slouží k přenosu dat mezi serverem a klientskou aplikací. Komunikace probíhá způsobem dotazu a odpovědi. Klientská aplikace (např. internetový prohlížeč) vytvoří a zašle HTTP dotaz na server. Server daný požadavek zpracuje a vytvoří HTTP odpověď, kterou klientovi vzápětí odešle. Klientská aplikace následně odpověď zpracuje (např. zobrazí webovou stránku). [2] Aby bylo možné jednoznačně určit, z kterého zdroje jsou data žádána, využívá HTTP tzv. jednotný lokátor zdroje (URL).

2.1 Uniform Resource Locator

Uniform Resource Locator (URL) je reprezentován řetězcem znaků, který přesně udává adresu serveru, na kterém jsou umístěna žádaná data. Obvyklá URL adresa pro HTTP se skládá ze 3 částí: schéma, doména a cesta. [3] Může však obsahovat i další části, níže je ukázáno základní schéma URL dle definice v dokumentu RFC-1738 [4]:

<schéma>://<host>:<port>/<cesta>?<dotaz>

Ukázková adresa na přihlašovací stránku kurzů UHK (ekvivalentní části jsou vyobrazeny totožnou barvou):

https://kurzy.uhk.cz:443/login/index.php?remember=0

- Schéma – Udává, jaké schéma či protokol má být využit (např. http, https, ftp, telnet, …), vždy se nachází před "://"
- Host – Host či hostitelská část identifikuje hostitelský server. Nejčastěji je reprezentována internetovou doménou (např. www.uhk.cz)
- Port – Specifikuje číslo síťového portu pro komunikaci. Ve většině případů není nutné port specifikovat, jelikož je automaticky určen protokolem – např. 80 pro HTTP, 443 pro HTTPS
- Cesta – Udává umístění na serveru (např. složka login, soubor index.php)
- Dotaz – Poskytuje informace pro formuláře. Vždy začíná za otazníkem, hodnoty jsou skládány slovníkově (klíč=hodnota) a více záznamů se odděluje ampersandem (např. barva=modra&pocasi=slunecno)
- Kotva – Odkazuje na místo v dokumentu, nejčastěji na hlavní nadpisy

2.2 Verze

První verzi HTTP vyvinul mezi lety 1989-1991 Sir Timothy John Berners-Lee a jeho tým jakožto základní protokol pro komunikaci na tehdy vznikajícím World Wide Webu. Od svého vzniku prošlo HTTP několika změnami a verzemi, od první verze 0.9 až po nově standardizovanou verzi 3. [5]

2.2.1 HTTP/0.9

Prvotní verze HTTP 0.9 byla velice jednoduchá, postavila však pevný základ, na kterém mohly stavět další verze. Už ve verzi 0.9 bylo rozhodnuto použití TCP, portu 80, principu klient-server a požadavku-odpovědi. Všechny požadavky obsahovaly pouze jeden řádek a začínaly stejným klíčovým slovem GET, následovaným cestou k žádanému dokumentu (např. GET /clanek.html). Po nalezení daného dokumentu, server pouze vrátil jeho celý obsah. Pro případné chyby však nebyly připraveny žádné stavové či chybové kódy. [5]

2.2.2 HTTP/1.0

Nedostatky verze 0.9 se pokusila vyřešit verze HTTP/1.0 vydaná v roce 1996. Požadavky už neobsahovaly pouze 2 části na jednom řádku. Přidala se informace o verzi a začal se rozvíjet koncept HTTP hlaviček (headers). Nově bylo možné odesílat libovolné hlavičky, každá na vlastním řádku a v přesně daném formátu – klíč: hodnota. Mezi důležité nově vzniklé hlavičky patřila i hlavička "Content-Type", jež umožnila definovat žádaný typ odpovědi. Díky této změně mohl klient žádat i jiné dokumenty než HTML, jako třeba obrázky, prostý text atd. Hlavičky byly přidány i do samotných odpovědí. Odpověď vždy začínala stavovým kódem (případně verzí HTML a stavovým kódem), následovaly libovolné hlavičky (např. formát odpovědi, její velikost, čas serveru, název serveru atd.) a až poté samotný obsah odpovědi. [5]

```
GET /clanek.html HTTP/1.0
Content-Type: text/html
User-Agent: Firefox/3.2 (Windows 10)
```

Obrázek 1 - Požadavek v HTTP/1.0, přejato z [5], upraveno autorem

```
200 OK
Date: Mon, 5 Jul 2022 02:46:51 UTC
Conent-Type: text/html
<html>
  <p>Hello world!</p>
</html>
```

Obrázek 2 - Odpověď v HTTP/1.0, přejato z [5], upraveno autorem

Kromě hlaviček byla verze 1.0 rozšířena o 2 nové HTTP metody. K již zavedenému GET se přidaly metody HEAD a POST. HEAD metoda sloužila k vyžádání hlavičky od serveru bez nutnosti posílat samotný obsah/tělo dokumentu. Metoda POST umožnila zasílání dat na server, předchozí verze totiž umožňovala data ze serveru pouze získávat/stahovat (metoda GET). [6] Více o metodách na straně 11.

Již v průběhu implementace HTTP/1.0 probíhaly paralelně přípravy na novou a již standardizovanou verzi HTTP/1.1. [5]

2.2.3 HTTP/1.1

První návrh standardu pro verzi HTTP/1.1 byl zveřejněn jen necelý rok po verzi 1.0. [5] Oficiální standardizace HTTP/1.1 však přišla až v červnu 1999 dokumentem RFC-2616. [7]

Verze 1.1 z větší části zachovala strukturu verze 1.0, ale přišla s několika vylepšeními. V první řadě přinesla nové HTTP metody, konkrétně: OPTIONS, PUT, DELETE, TRACE a CONNECT. Tyto nové metody např. umožnily jednodušší manipulaci dat, komunikaci se serverem a zjišťování chyb. Více o metodách na straně 11.

Kromě nových metod umožnila tato verze znovupoužitelnost jednoho síťového spojení. Předchozí verze vyžadovaly nové síťové spojení pro každý požadavek + odpověď. V případě, že jedna stránka obsahovala vícero vložených částí (textů, obrázků apod.), znamenalo to, že pro každý obrázek byl zaslán nový požadavek a tím i vytvořeno nové spojení. Verze HTTP/1.1 umožnila v rámci jednoho spojení zaslat několik požadavků, což přineslo zrychlení načítání a snížilo síťovou zátěž.

Další důležitou novinkou bylo zřetězené zpracování (tzv. pipelining). Nový požadavek mohl být odeslán ještě dříve, než přišla odpověď na předchozí

požadavek. V předchozích verzí se vždy muselo čekat na stažení celé odpovědi. Toto vylepšení přineslo ještě větší zrychlení načítání stránek.

Mezi dalšími novinkami bylo umožněno přenášet soubory o předem neznámé velikosti, nebo vyžádat pouze části odpovědi, dále pak např. vybírání jazyka, kódování, typu a formátu odpovědi a další. [8]

```
GET /en-US/docs/Glossary/Simple_header HTTP/1.1
Host: developer.mozilla.org
User-Agent: Mozilla/5.0 Gecko/20100101 Firefox/50.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://developer.mozilla.org/en-US/docs/Glossary/Simple_header

200 OK
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Wed, 20 Jul 2016 10:55:30 GMT
Etag: "547fa7e369ef56031dd3bff2ace9fc0832eb251a"
Keep-Alive: timeout=5, max=1000
Last-Modified: Tue, 19 Jul 2016 00:59:33 GMT
Server: Apache
Transfer-Encoding: chunked
Vary: Cookie, Accept-Encoding

(content)
```

Obrázek 3 - Požadavek a odpověď v HTTP/1.1 [5]

Verze 1.1 se rychle rozšířila a stala se dominantní HTTP verzí na celém internetu. Vydržela bez větších změn po dobu 15 let až do roku 2014, kdy byl popsán návrh verze HTTP/2. Od roku 2016 pozvolna ustupuje její využívání, v březnu 2019 bylo přibližně 50 % HTTP požadavků verze 1.1 a 50 % verze 2. [9]

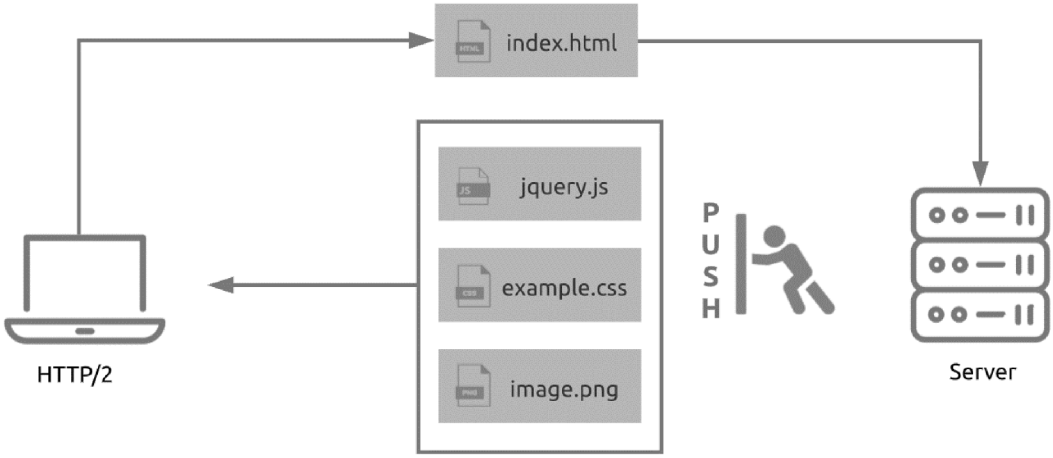
2.2.4 HTTP/2

Verze HTTP/2 byla zveřejněna pod RFC-7540 v květnu 2015. [10] Postupně se stává hlavní verzi HTTP, v červnu 2022 bylo 68 % všech HTTP požadavků odesláno přes verzi 2. [9]

Od roku 1999 narostlo množství zařízení využívající HTTP do řádů čísel tehdy z daleka neočekávaných. Miliardy zařízení různých tvarů a velikostí vyžadují čím dál více dat za co nejkratší čas. Verze 2 tak získala 2 základní podmínky/požadavky – zachovat kompatibilitu s HTTP/1.1 a co nejvíce snížit čas odezvy (zrychlit načítání stránek). [8] Kvůli požadavku na zachování plné kompatibility s HTTP/1.1 nebyly do verze 2 přidány žádné nové metody, ani nebylo rozšířeno základní chování, většina změn se soustředila na zrychlení odezvy. Byla přidána komprese HTTP hlaviček, vylepšeno tzv. zřetězené zpracování, přidána zcela nová funkce názvem "HTTP/2 Server Push" a další.

Na rozdíl od verze 1.1, která přijímá a odpovídá ve formátu prostého textu, verze 2 využívá binární formát, což umožňuje nové optimalizace, které nejsou v prostém textu možné. Cesty ke zrychlení se našly i přes snížení velikosti samotného požadavku a odpovědi – čím menší požadavek/odpověď, tím rychlejší přenos. Komprese dat byla v HTTP podporována již dříve, ale HTTP hlavičky se vždy odesílaly nekomprimovaně. Nová verze umožnila komprimovat i samotné HTTP hlavičky, čímž snížila velikost přenášených dat. Vylepšení se dočkal i tzv. pipelining, který umožňoval zasílat několik požadavků najednou bez nutnosti čekání na předchozí odpověď. Nově pojmenováno jako Multiplexování (Multiplexing), umožňuje v rámci jednoho síťového spojení zažádat nejen o pár částí stránky, ale rovnou o všechny části.

Webové stránky se během let vylepšily, již se nejedná jen o čisté HTML, přidaly se i skripty a kaskádové styly. Tyto části je obvykle nutné co nejrychleji načítat, jelikož bez nich často stránky nemusejí fungovat korektně. Přechozí verze načetly HTML soubor a následně žádaly o připojené soubory (skripty, styly apod.), což znatelně prodlužovalo dobu prvního vykreslení stránky. Nová funkce "HTTP/2 Server Push" přidala serverům možnost očekávat, že klient bude vyžadovat i připojené části. Server je díky tomu schopen tyto části odeslat klientovi ještě dříve, než o ně stihne klientská aplikace zažádat. [11] Viz. následující Obrázek 4 – Server Push [11].



Obrázek 4 – Server Push [11]

Klient žádá stránku index.html, kterou následně server zasílá. Server však nečeká na požadavky o soubory jquery.js, example.css a image.png, ale automaticky je zasílá klientu.

HTTP/2 přinesl vylepšení, které umožnily zrychlit načítání webových stránek, jeho budoucnost je však nejasná. Nelze s jistotou říci, zda se stane "hlavní" HTTP verzí, běžící na 99 % zařízeních, či bude rovnou postupně nahrazován nově popsanou verzí HTTP/3.

2.2.5 HTTP/3

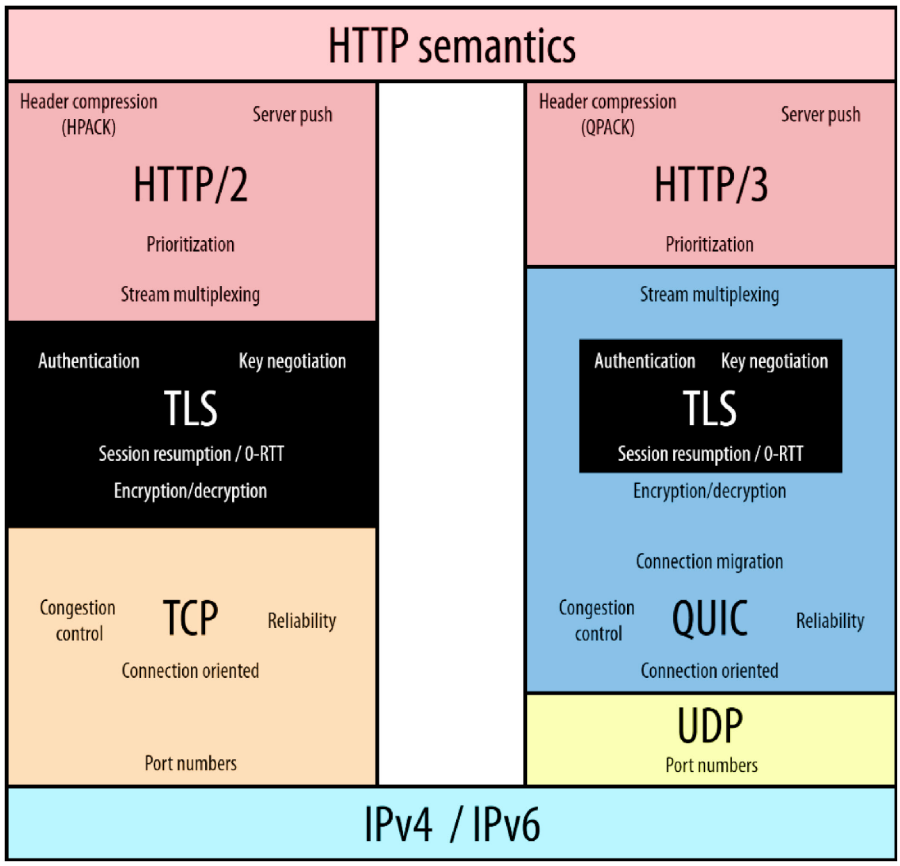
O překonání verze 1.1 a 2 se v následujících letech pokusí nová verze HTTP/3, jež byla oficiálně popsána v červnu 2022 pod RFC-9114. [12] Za vznik HTTP/3 lze poděkovat společnosti Google, která experimentovala se svým protokolem QUIC pro použití v HTTP. Dříve nazývanému protokolu HTTP-over-QUIC bylo v listopadu 2018 schváleno přejmenování na HTTP/3. [13] Hlavní rozdíl v HTTP/3 oproti všem předchozím verzím představuje použití již zmíněného protokolu QUIC.

2.2.5.1 QUIC

Protokol QUIC neoficiálně též jako *Quick UDP Internet* Connections (v českém překladu *Rychlé UDP Internetové Připojení*) byl představen veřejnosti v roce 2013 společností Google. Důvodem vzniku nového protokolu byla snaha o co nejvyšší zrychlení komunikace mezi zařízeními. Všechny dosavadní verze HTTP byly postaveny nad *Transmission Control Protokolem* (TCP). [14] TCP vyžaduje před úspěšným navázáním komunikace tzv. Trojcestný handshaking (odeslání 3 oddělených síťových

rámců pro potvrzení spojení), až poté je možné posílat data a ukončení spojení probíhá obdobným postupem. Tato nutnost potvrzování přináší částečné zpomalení komunikace, a proto využívá protokol QUIC *User Datagram Protocol* (UDP). Ten snižuje nezbytný počet cest mezi serverem a klientem, díky čemuž probíhá komunikace rychleji. [15]

K červenci 2022 bylo HTTP/3 (vč. QUIC) podporováno 75 % aktivně využívanými webovými prohlížeči [16]. Zároveň již 15 % ze všech webový požadavků podporovalo verzi 3 a vzhledem k nedávné standardizaci lze v následujících letech očekávat postupný růst v používání. [9]



Obrázek 5 – Porovnání HTTP/2 vs HTTP/3 [15]

2.3 Hypertext Transfer Protocol Secure

Vzhledem k tomu, že veškerá komunikace přes HTTP probíhá v nezašifrované podobě, vznikla již v roce 1994 první verze zabezpečeného protokolu *Hypertext Transfer Protocol Secure* (HTTPS), tehdy čistě pro prohlížeč od společnosti Netscape. Původní verze využívala v té době zcela nový kryptografický protokol pojmenovaný *Secure Sockets Layer*, zkráceně SSL, jejž byl vytvořen totožnou společností pro účely HTTPS. Verze SSL 1.0 nicméně nebyla nikdy veřejně vydána kvůli závažným bezpečnostním trhlinám, které

byly opraveny až ve verzi SSL 3.0 v roce 1996. Namísto vydání verze 4.0 bylo v roce 1999 rozhodnuto organizací IETF o vydání nového kryptografického protokolu TLS (*Transport Layer Security*). Verze TLS 1.0 byla poté popsána v RFC-2246. SSL protokol byl posléze označen jako zastaralý a vývoj pokračoval v rámci nových verzí TLS. [17] Nejnovější verze TLS 1.3 z roku 2018 je popsána v RFC-8446. [18]

Protokol HTTPS se stal rozšířeným natolik, že některé webové prohlížeče začaly nezabezpečené stránky (bez https) označovat jako nedostatečně bezpečné (např. prohlížeč Chrome od června 2018 a ostatní brzy následovaly). [19] V roce 2021 bylo přibližně 94 % webových požadavků odesláno právě přes HTTPS. [9]

2.4 Struktura

Každý HTTP požadavek a každý HTTP odpověď má předem přesně definovanou strukturu dle používané verze. Za účelem jednoduchého vysvětlení je v této sekci popsána struktura HTTP/2, která je identická s verzí HTTP/1.1.

Všechna komunikace mezi klientem a HTTP serverem probíhá pomocí HTTP zpráv. Zprávy se následně dělí na 2 typy – Požadavek (anglicky Request) a Odpověď (anglicky Response). Požadavek je vytvářen klientem, resp. klientskou aplikací (např. webovým prohlížečem) a následně je odeslán na server. Server poté reaguje HTTP odpovědí. Odpovědi mají z velké části obdobnou strukturu jako požadavky. [20] Na ukázkách níže si lze prohlédnout obvyklou strukturu požadavku (Zdrojový kód 1) a odpovědi (Zdrojový kód 2).

```
1. GET / HTTP/1.1
2. User-Agent: PostmanRuntime/7.29.0
3. Accept: */*
4. Cache-Control: no-cache
5. Host: www.seznam.cz
6. Accept-Encoding: gzip, deflate, br
7. Connection: keep-alive
```

Zdrojový kód 1 – Ukázkový HTTP požadavek, vlastní zpracování

```
1.    HTTP/1.1 200 OK
2.    vary: Accept-Encoding
3.    content-type: text/html; charset=UTF-8
4.    cache-control: no-cache, no-store, must-revalidate
5.    pragma: no-cache
6.    x-frame-options: SAMEORIGIN
7.    content-security-policy: frame-ancestors 'self'
8.    content-encoding: gzip
9.    date: Thu, 07 Jul 2022 13:33:54 GMT
10.   x-envoy-upstream-service-time: 35
11.   server: envoy
12.   strict-transport-security: max-age=86400; preload
13.   transfer-encoding: chunked
14.
15.   <!DOCTYPE html>
16.   <html lang="cs">
17.   ... (obsah odpovědi byl zkrácen) ...
18.   </html>
```

Zdrojový kód 2 – Ukázková HTTP odpověď, vlastní zpracování

Struktura obvyklých HTTP zpráv je následující:

- První řádek – Popisný a povinný

První řádek zprávy (požadavku i odpovědi) popisuje základní a nejdůležitější část zpráv. V případě požadavku první klíčové slovo definuje použitou metodu zprávy (více v 2.4.2 Metody), v ukázce se jedná o metodu GET. Poté následuje lokace žádaného souboru na serveru (v ukázce "/", jenž symbolizuje hlavní stránku, též nazývaný index). Na závěr je definována žádaná verze HTTP. Touto verzí následně server odpovídá a následuje stavový kód (více v 2.4.3 Stavové kódy), v případě ukázky "200 OK". Popisný řádek musí být vždy jednořádkový.

- HTTP hlavičky – Nepovinné, ale žádané

Druhým řádkem HTTP zprávy obvykle začíná část věnovaná HTTP hlavičkám. Nejdůležitější hlavičku v případě požadavku představuje hlavička s názvem Host, která specifikuje server, na který je daný požadavek zasílán. V ukázce se jedná o doménu *www.seznam.cz*. Důležitá je i hlavička Accept, jež serveru oznamuje žádaný formát dat. Ukázka udává "*/*", což znamená libovolný formát. Server následně zvolí nejvhodnější formát podle typu dokumentu.

Nejvýznamnější hlavičku v HTTP odpovědi reprezentuje "Content-Type", který udává formát příchozích dat a případně i druh kódování. V ukázce se jedná to HTML s kódováním UTF-8. [20] Více o hlavičkách v sekci 2.4.1 Hlavičky.

- Prázdný řádek – Povinný

Prázdný řádek odděluje hlavičku zprávy od samotného těla. Oznamuje, že veškerá metadata již byla odeslána a následuje tělo zprávy či její konec (neobsahuje-li tělo).

- Tělo zprávy – Nepovinné

Po prázdném řádku může následovat samotné tělo zprávy v předem daném formátu. Z HTTP požadavků tělo obsahují metody jako POST, PATCH a PUT, a to pouze je-li třeba zaslat data na server, ostatní metody obvykle tělo neobsahují. V případě HTTP odpovědí zahrnutí a nezahrnutí těla bývá opačně, kdy CRUD metody vrací pouze stavové kódy bez samotného těla a metoda GET vrací tělo zprávy téměř vždy. [20]

2.4.1 Hlavičky

HTTP hlavičky (anglicky Headers) slouží k předávání dodatečných dat mezi klientem a server. Formát hlaviček je tzv. slovníkový, kdy každý klíč je od hodnoty oddělen dvojtečkou (např. "Content-Type: text/html", klíč je "Content-Type" a hodnota "text/html"). Klíče jsou tzv. case-insensitive neboli se nerozlišují velká a malá písmena (např. "CoNTenT-tYPE" a "Content-Type" představují stejný klíč. [21]

Hlavičky lze rozdělit do dvou kategorií, standardizované a nestandardizované. Část oficiální (standardizovaných) hlaviček včetně jejich popisu lze nalézt v dokumentu RFC-4229, postupně aktualizovaný seznam se nachází i na stránkách organizace *Internet Assigned Numbers Authority* (IANA) v [22].

Nestandardizované hlavičky museli až do června 2012 začínat předponou "X-", tato povinnost byla zrušena dokumentem RFC-6648. Vlastní hlavičky tak může definovat a využívat jak klient, tak i server. [23]

HTTP požadavky obvykle obsahují hlavičky k určení formátu žádaných dat (začínající na "Accept"), dále také hlavičky sloužící k autentizaci (hlavička "Authorization") anebo identifikaci používaného prohlížeče (hlavička "User-Agent"). Požadavek však může obsahovat mnohem více hlaviček. Součástí HTTP odpovědi bývají hlavičky týkající se obsahu zprávy, které začínají slovem "Content-". Např. "Content-Encoding" pro oznámení typu kódování, či "-Length" pro informování od délce zprávy. Dále může být uvedeno stáří dat v mezipaměti "Age" a mnohé další hlavičky. [21]

2.4.2 Metody

HTTP metody definují, které akce se mají vykonat na dané stránce. Každá metoda má různou implementaci, některé však mohou mít společné vlastnosti. Mezi základní vlastnosti patří idempotence a bezpečnost. [24]

Idempotentní metody se vyznačují tím, že při odeslání stejných požadavků za sebou vrátí server identické odpovědi bez změny stavu serveru. Odpovědi však mohou mít vracet jiné stavové kódy. Idempotentní metody by neměly způsobovat vedlejší efekty, vyjma jsou-li tyto efekty idempotentní. [25]

Bezpečné metody jsou takové HTTP metody, jež nemění stav na serveru. Jedná se o metody sloužící pouze ke čtení/získávání dat. Všechny bezpečné metody jsou zároveň idempotentní, avšak ne obráceně. [26]

Názvy metod jsou standartně psány velkými písmeny. Níže jsou krátce popsány jednotlivé HTTP metody. [24]

- CONNECT – navazuje komunikaci se severem přes specifikovaný port
- DELETE – idempotentní metoda, jež maže daná data na serveru
- GET – bezpečná metoda sloužící k získání dat ze serveru
- HEAD – bezpečná metoda, která vrací pouze hlavičky
- OPTIONS – bezpečná metoda k zjištění povolených metod
- PATCH – slouží k úpravě dat na serveru
- POST – hlavní metoda sloužící k odeslání dat na server
- PUT – idempotentní metoda k nahrání nových dat
- TRACE – bezpečná metoda, jež slouží ke kontrole příchozího požadavku

Přehledné srovnání jednotlivých metod lze nalézt v následujícím porovnání (Obrázek 6).

	CONNECT	DELETE	GET	HEAD	OPTIONS	PATCH	POST	PUT	TRACE
Požadavek obsahuje tělo	✗	✗	✗	✗	✗	✓	✓	✓	✗
Odpověď obsahuje tělo	✓	✓	✓	✗	✓	✓	✓	✓	✓
Bezpečná	✗	✗	✓	✓	✓	✗	✗	✗	✓
Idempotentní	✗	✓	✓	✓	✓	✗	✗	✓	✓

Obrázek 6 – Porovnání HTTP metod, přejato z [7], upraveno autorem

2.4.3 Stavové kódy

Každá HTTP odpověď obsahuje na svém začátku tzv. stavový kód. Jedná se o trojciferné číslo, jež oznamuje klientovi, zda byl požadavek zpracován úspěšně, či se při zpracování vyskytly problémy. Nejnovější definici stavových kódů poskytuje dokument RFC-9110 z června 2022. Za platné kódy lze považovat pouze stavové kódy v rozmezí 100 až 599 (včetně). [1] První číslo kódu udává kategorii, podle níž lze jasně určit, jakého informačního typu daný stavový kód je.

Dle RFC-9110 z [1] jsou kategorie stavových kódů následující:

- 1xx – Informační – Označuje prozatímní odpověď, že daný požadavek byl serverem přijat a očekává se jeho pokračování
- 2xx – Úspěch – Udává úspěšné zpracování požadavku, nejčastějším zástupcem je kód "200 OK"
- 3xx – Přesměrování – Udává, že žádaný zdroj se nachází na jiné adrese a je nutné přesměrování požadavku
- 4xx – Chyba na straně klienta – Označuje klientskou chybu (např. nedostatečná oprávnění, neexistující adresa, nesprávná syntaxe apod.)
- 5xx – Chyba na straně serveru – Informují, že server nebyl schopný zpracovat požadavek z interních důvodů

3 Webové API

Webové API či webové služby lze rozdělit do hlavních „technologií“ či směrů, kterými se webové služby ubírají. Mezi tyto technologie patří SOAP, REST a GraphQL. Každá služba využívá k přenosu určité formáty dat, které jsou popsány v následující části.

3.1 Formáty dat

Jelikož klienty a servery mohou být různého typu, s jinými operačními systémy a platformami, je nutné, aby komunikovaly pomocí standardizovaného jazyka a formátu. Dané formáty dat musí mít předem danou strukturu pochopitelnou pro klienty i servery. Tyto formáty musí disponovat schopností převedení libovolné vstupní datové struktury (objekt, text, číslo…) do daného formátu, tzv. *serializace*. Zároveň musí být možné data opět převést do původní podoby, tzv. *deserializace*. Níže jsou popsány dva nejpoužívanější základní formáty dat.

3.1.1 JSON

JSON neboli *JavaScript Object Notation*, představuje jeden z nejvyužívanějších formátů pro výměnu webových dat. Jedná se o standardizovanou notaci, která je popsána v RFC-8259 i EMCA-404. Díky standardizaci je přesně učena struktura a podoba dat. Data jsou reprezentována textově a nezávisle na jazyce, jsou však lehce čitelná lidmi i stroji. Struktura může být v podobě párů klíč a hodnota či seznamu hodnot. Klíč je vždy reprezentován textovou hodnotou (string). Hodnoty mohou být vyjádřeny objektem, polem, číslem, textem či hodnotou true/false/null. [27]

```
{
  "Image": {
      "Width":  800,
      "Height": 600,
      "Title":  "View from 15th Floor",
      "Thumbnail": {
          "Url":    "http://www.example.com/image/481989943",
          "Height": 125,
          "Width":  100
      },
      "Animated" : false,
      "IDs": [116, 943, 234, 38793]
    }
}
```

Obrázek 7 – Ukázka JSON formátu [27]

S formátem JSON se lze setkat u HTTP požadavku i odpovědi. Např. HTTP POST požadavek může obsahovat tělo zprávy ve formátu JSON a odpověď na HTTP GET může být taktéž tělem formátována do JSONu.

3.1.2 XML

Dalším důležitým zástupcem formátu dat v rámci webového API je formát XML. Celý název tohoto formátu zní *Extensible Markup Language*. XML byl standardizován v roce 1998 společností *World Wide Web Consortium* (W3C) jakožto součást standardu SGML (*Standard Generalized Markup Language*). Cílem bylo umožnit přenos dat po webu pomocí generického SGML, obdobně jako je to možné u HTML. [28] Podobně jako u JSON formátu je u XML výhodou lehká čitelnost stroji, které jej mohou i jednoduše zpracovat. XML je do určité míry i poměrně jednoduše čitelný člověkem. Obsah XML má jasně danou logickou strukturu. Veškeré entity jsou formátovány značkami začínající na "<" a končící ">". Každý dokument musí obsahovat právě jednu kořenovou entitu (tzv. *root*). Každý element může v sobě zanořovat další elementy a sám může obsahovat ještě atributy, jejichž hodnoty musí být definovány v uvozovkách. XML lze též reprezentovat pomocí datové struktury strom. [28]

```xml
<books>
    <book>
        <title lang="cs">R.U.R.</title>
        <year>1920</year>
        <authors>
            <author>
                <name>Karel Čapek</name>
                <year>1890</year>
            </author>
        </authors>
    </book>
    <book>
        <title lang="cs">Staré pověsti české</title>
        <year>1894</year>
        <authors>
            <author>
                <name>Alois Jirásek</name>
                <year>1851</year>
            </author>
        </authors>
    </book>
</books>
```

Obrázek 8 – Ukázka XML formátu, vlastní zpracování

3.2 SOAP

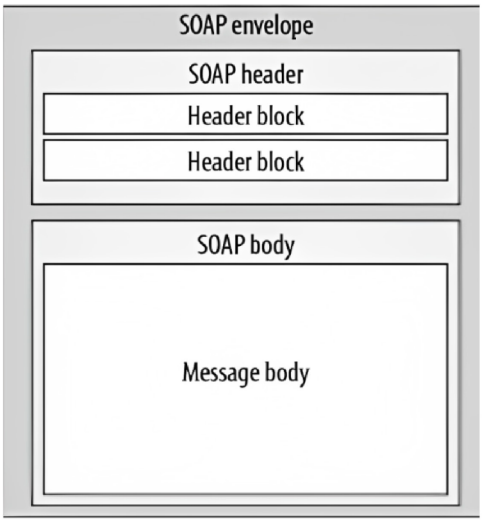
SOAP neboli *Simple Object Access Protocol* je protokol sloužící k výměně jasně strukturovaných dat v distribuovaném prostředí. Komunikace v tomto protokolu využívá zpráv, které jsou definovány v již dříve zmíněném XML formátu. Samotné přenášení zpráv může probíhat různými protokoly, nejčastěji však využívá primárně HTTP. Protokol by navržen s důrazem na nezávislost na platformách i s důrazem na rozšiřitelnost. [29] Díky nezávislosti na platformě a využití formátu XML je možné, že server i klient mohou využívat rozdílné technologie a programovací jazyky. První verze protokolu nese označení 1.1 a specifikace byla vydána konsorciem W3C v květnu 2000. [30] Tato verze se však nedočkala oficiální standardizace a byla označena pouze jako návrh. I přesto se lze setkat s poměrně častými implementacemi této verze. Oficiální doporučení od W3C připadlo až novou tzv. „druhou edici" (Second Edition) nesoucí označení verze 1.2. Specifikace verze 1.2 byla zveřejněna v dubnu 2007. [29] Pracovní skupina pod W3C udržovala tuto specifikaci až do července 2009, kdy byla skupina rozpuštěna. Od rozpuštění skupiny lze specifikaci SOAP verze 1.2 považovat za první, jedinou a prozatímně poslední verzi SOAP. [31]

SOAP ve své specifikaci definuje přesnou podobu zasílaných zpráv mezi klienty a servery. Zprávy jsou formátovány do XML a pro přenos může být využit libovolný protokol. [29] Nejčastěji se využívá dříve zmíněné HTTP, avšak kvůli způsobu předávání zpráv je v praxi využívána hlavně HTTP metoda POST. Zjednodušená komunikace probíhá tak, že klient vytvoří SOAP požadavek, zabalí ho do tzv. SOAP obálky a tento požadavek odešle na požadovanou URL adresu. Server přijme SOAP obálku, rozbalí ji, vykoná požadovanou činnost a odpověď zabalí opět do nové obálky, kterou odešle zpět. [32]

3.2.1 Zprávy

Obdobně jako u HTTP je i u SOAP komunikace dělena na požadavek (request) a odpověď (response). Oba tyto druhy lze zařadit pod tzv. SOAP zprávy. Každá zpráva má jasně definovanou strukturu podle specifikace pro SOAP. [29] Zpráva se skládají z několika částí v rámci jmenného prostoru nazvaného *soap-envelope* a to, např. "https://www.w3.org/2003/05/soap-envelope/", v rámci internetu lze nalézt mnohé rozdílné adresy, jež se však obsahově neliší. Základními elementy tvořící SOAP zprávu je obálka (SOAP envelope), hlavičky (SOAP headers), tělo (SOAP body) a informace

o chybě (SOAP fault), tyto elementy se však sami skládající z dalších elementů. [29] Obrázek 9 znázorňuje základní schématickou strukturu SOAP zprávy.



Obrázek 9 – Struktura SOAP zprávy [32]

Jednotlivé zprávy vždy musí obsahovat právě jeden kořenový element. Tento kořen vyjadřuje tzv. obálka (SOAP envelope).

3.2.1.1 Obálka

Jak už název napovídá, obálka slouží k zaobalení obsahu SOAP zprávy, čímž označuje začátek a konec zprávy samotné. Počáteční značka obálky také svými atributy obvykle definuje veškeré jmenné prostory celé zprávy. Obálka může obsahovat hlavičky. Pokud je obsahuje, musí se všechny nacházet v právě jednom elementu nazvaném *Header* a tento element se musí nacházet před elementem těla obálky. Obálka zároveň musí obsahovat právě jeden element s tělem zprávy nazvaným *Body*. [32]

3.2.1.2 Hlavičky

Jedná se o nepovinný element obálky. Pokud jsou nějaké hlavičky definovány, musí vždy být obsaženy uvnitř blokového elementu *Header*. Tento blokový element musí být v rámci obálky na prvním místě, před tělem. Hlavičky poskytují mechanismus na rozšiřování samotné SOAP zprávy, obdobně jako hlavičky u HTTP. Blokový element hlaviček může obsahovat libovolný počet různorodých elementů hlaviček. Zároveň může obsahovat specifické atributy, např. k učení role, způsobu kódování apod. [29]

3.2.1.3 Tělo

Element těla reprezentuje povinnou a nejdůležitější část obálky. Každá obálka musí obsahovat právě jeden element nazvaný *Body*. Obsahem tohoto elementu mohou být jakákoliv validní data v XML formátu. [32] V případě požadavku jsou přes tělo předávány informace o požadovaném endpointu a o žádané operaci. Součástí těla by měly být i všechny hodnoty pro parametry, které daná operace vyžaduje. Naopak v případě odpovědi jsou skrze tělo předávány výstupní informace o výsledku z vykonané operace. [29]

3.2.1.4 Chyby

Element chyb je předávám pouze v SOAP odpovědích, kde při vykonávání žádané operace došlo k chybě. Tento element se značí značkou s názvem *Fault* a je vždy obsažen uvnitř elementu těla odpovědi. Každá chyba musí být složena minimálně ze 2 dalších elementů, které jsou vypsány v bodech níže. [29]

- Code – Jedná se o povinný element skládá z následujících částí
  - Value – Povinný element reprezentující chybový kód
  - Subcode – Volitelný element definující chybový pod-kód
- Reason – Povinný element, který obsahuje informace o chybě ve formě textu
- Node – Volitelný element ukazující na element, který mohl způsobit chybu
- Role – Volitelný element identifikující roli elementu, který byl zpracováván v době, kdy se vyskytla chyba
- Detail – Volitelný element, který obsahuje dodatečné detailnější informace k chybě

Mezi specifikované chybové kódy patří *VersionMismatch*, který oznamuje, že požadavek obsahuje elementy špatné verze SOAP. Další chybový kód *MustUnderstand* informuje, že požadavek obsahoval element či atribut elementu, který nebyl serverem rozpoznán. Mezi další kód patří i Sender a Receiver, kde první informuje o tom, že zpráva nebyla správě formátována, či obsahovala nedostatečná data k vykonání požadavku. Kód Receiver definuje chybu, která nastala při samotném zpracování požadavku. [29] Kromě předem definovaných chybových kódů, může odesílatel vložit do odpovědi i vlastní chyby, kde kód i detaily mohou být jakékoliv. Jedinou podmínku tvoří nutnost u těch vlastních chyb definovat jmenný prostor s definicí chyby. [32]

3.2.2 Jmenné prostory

Jmenné prostory představují v XML odkazy na definici daného elementu. Odkaz na jmenný prostor se obvykle definuje v kořenovém elementu, vlastní definice však mohou obsahovat i jednotlivé elementy. Jmenné prostory jsou od názvů elementů odděleny dvojtečkou a u definice je určen názvem a hodnotou oddělenou rovnítkem. [29] Definice jmenného prostoru může vypadat obdobně následujícímu příkladu: *xmlns:env="http://www.w3.org/2003/05/soap-envelope"* a použití by bylo následující *<env:Header>* v tomto případě by se jednalo o jmenný prostor s názvem *env*.

3.2.3 WSDL

Aby bylo možné zvenčí zjistit strukturu webové SOAP služby bez nutnosti přístupu ke zdrojovému kódu byla vytvořena technologie WSDL neboli *Web Services Description Language*. Jedná o popisný jazyk v XML, který umožňuje klientům zjistit celkovou strukturu webové služby včetně vstupů a výstupu. WSDL není závislý na programovacím jazyku a ani platformě, díky čemuž je možné WSDL zpracovávat v jakékoliv programu na libovolné platformě. Kromě toho existují nástroje, které z WSDL dokumentu vygenerovat zdrojový kód klienta. [32]

První verzi specifikace k WSDL 1.1 vydala společnost W3C březnu 2001. Jednalo však pouze o koncept a obdobně jako SOAP 1.1, nebyla verze WSDL 1.1 oficiálně schválena ke standardizaci. Následující verze 1.2 byla v průběhu vývoje přejmenována na WSDL 2.0. Pracovní skupina pod konsorciem W3C vydala verzi 2.0 v červnu 2007, tato verze byla již oficiálně schválena a standardizována. Kvůli poměrně zdlouhavému vývoji verze 2.0 začala být využívána již verze 1.1. V praxi je tedy možné se setkat s oběma verzemi. [33]

Dle [32] je pomocí WSDL popisováno všechno, co daná webová služba dělá, jak to dělá a jak uživatelé mohou danou službu využívat. Mezi jednu z hlavních výhod využití WSDL patří:

- WSDL umožňuje jednoduchou tvorbu a udržování webových služeb
- WSDL umožňuje jednoduché využití webové služby
- WSDL umožňuje jednodušší implementaci změn ve webové službě
- Díky WSDL je možné vygenerovat celé klienty pro webovou službu

Využití WSDL však přináší i své nevýhody, mezi které patří i například nevyřešené verzování. V případě větších změn ve webové službě není určen způsob jakéhokoliv verzování. [32]

WSDL dokument je soubor, který v XML formátu popisuje všechny operace dané webové služby, jména a datové typy vstupních i výstupních parametrů včetně kompletního popisu datového typu výstupů všech operací. Kromě toho obsahuje i URL adresu dané webové služby. Díky tomu, že WSDL obsahuje veškeré informace o dané webové službě, je možné využití tzv. *code-generators* – neboli generátorů kódu. Tyto generátory přijímají na vstupu validní WSDL dokument a umožňují vygenerování kompletního zdrojového kódu, který bude volat danou webovou službu. Zároveň existují i technologie, které z existující webové služby umí vygenerovat WSDL dokument. Například platforma .NET od společnosti Microsoft umožňuje automatické generování WSDL dokumentů ke všem vytvářeným webovým službám. [32] Obdobně umožňuje vývojové prostředí *Visual Studio* od stejné společnosti vygenerovat z WSDL dokumentu zdrojový kód pro volání dané služby.

3.2.3.1 Struktura

Jak již bylo zmíněno, i přes chybějící oficiální standardizaci, je stále využívaná verze WSDL 1.1. V následujícím popsání struktury WSDL však bude popisována verze WSDL 2.0 dle oficiální specifikace od konsorcia W3C. [33]

Kořenový element je ve WSDL označován názvem *definitions*. Tento kořen obsahuje všechny hlavní jmenné prostory využívané v celém souboru a atribut *targetNamespace*, který definuje URL adresu pro určení příslušnosti elementů nejvyšší úrovně ke jmennému prostoru. Obsahem kořenového mohou být například následující elementy z oficiální specifikace [33].

- Documentation (Dokumentace)

Tento volitelný element obsahuje popis ve formátu lidsky čitelného textu či ve formátu XML s vnořenými elementy pro detailnější popis. Struktura tohoto elementu není pevně daná a je otevřena libovolnému rozšíření. Element dokumentace se odlišuje tím, že může být vnořen do jakéhokoliv jiného elementu a díky tomu se může v celém WSDL dokumentu vyskytovat mnohokrát. [33]

- Types (Typy)

V tomto elementu jsou definovány veškeré použité datové typy, jež jsou použity v dané webové službě. Definovány by měly být pouze vlastní či komplexní datové typy, včetně všech vlastností a názvů. Definice základních typů (text, číslo apod.) není nutná. [33]

- Interface (Rozhraní)

Komponenta rozhraní popisuje sekvenci zpráv, které webová služby přijímá či odesílá. V rozhraní jsou definovány všechny operace dané webové služby. Pro každou operaci je zde definován název datových typů vstupu a výstupu. [33]

- Binding (Vazby)

Obsahem elementu vazeb je definována implementace detailů nezbytných k připojení k webové službě. Každá vazba propojuje ve svém obsahu rozhraní, službu a operace dané služby. Zároveň určuje použitý protokol pro komunikaci (např. HTTP) a typ zpráv (např. SOAP). [33]

- Service (Služby)

V této komponentě jsou popsány koncové body na jejichž adrese se nachází daná webová služba. Každý element služby obsahuje název služby, její URL adresu a napojení na rozhraní s vazbou. [33]

3.3 REST

REST neboli *Representational State Transfer*, na rozdíl od SOAP nepředstavuje protokol, ale architekturu pro distribuovaná prostředí. Distribuovaná prostředí si lze v kontextu této práce představit jako části programů na vícero počítačích komunikujících přes síť/internet. Jeden z autorů protokolu HTTP (Roy Thomas Fielding) popsal v roce 2000 v rámci své disertační práce *Architectural Styles and the Design of Network-based Software Architectures* architekturu REST. [34]

3.3.1 Pravidla

Architektura se snaží vyřešit problém škálovatelnosti webu pomocí pravidel či omezení (z anglického *constraints*), která jsou rozdělena do 6 kategorií. [34]

3.3.1.1 Klient-Server

Jedná se o jednu z nejdůležitějších částí v definici REST. Pravidlo udává nutnost rozdělení na 2 části – klientskou a serverovou. Server disponuje daty, která předává klientům. Klient má poté za úkol získaná data reprezentovat v uživatelském rozhraní. Díky tomuto dochází ke zjednodušení serverové části, jelikož reprezentace dat je přenesena na klienty. [34]

3.3.1.2 Jednotné rozhraní

Velký důraz je kladen i na pravidlo s názvem *Uniform Interface* (jednotné rozhraní). Vzhledem k předchozímu pravidlu je zřejmé, že mezi sebou musí komunikovat různé části systému. Ale aby části byly schopny komunikace, je nutné definovat jednotné rozhraní, přes které bude komunikace probíhat. [34] V rámci tohoto pravidla jsou přidány ještě další pravidla, které jsou vypsány níže – pro zjednodušení jsou použity zkrácené názvy.

- Identifikace zdroje – Každý zdroj musí mít jedinečný identifikátor (např. URI)
- Manipulace se zdroji – Udává, že se zdrojem by se mělo pracovat skrze jeho reprezentace a ne přímo
- Samo-popisující zprávy – Jednotlivé zprávy by měly obsahovat všechny nezbytné informace, včetně hlaviček
- Hypermédia – Reprezentace zdroje by měla obsahovat odkazy na související zdroje

3.3.1.3 Vrstvený systém

Pravidlo spočívá v rozdělení do hierarchických vrstev, kde jednotlivé vrstvy mohou komunikovat pouze se sousedními vrstvami (pouze s nižší nebo vyšší vrstvou). Díky tomuto vrstvy vykonávají pouze svůj „jednoduchý“ úkol a dochází k obecnému zjednodušení a zpřehlednění.

3.3.1.4 Mezipaměť (*Cache*)

Některá data je možné ukládat do mezipaměti. V případě opětovného požadavku není nutné např. složitý úkol vykonávat znovu, ale rovnou vrátit uložený výsledek. Toto pravidlo je výhodou převážně u zdrojů, které se často nemění. Díky ukládání do mezipaměti dochází k vysokému zvýšení výkonu.

3.3.1.5 Bezstavovost

Pravidlo bezstavovosti značí, že server si nepřechovává stav mezi jednotlivými požadavky, a proto všechny požadavky musí obsahovat kompletní data k jejich vykonání. Stav mezi požadavky může být ukládán pouze na klientovi, díky čemuž je ušetřen výkon serverů. [34]

3.3.1.6 Kód na vyžádání

Toto volitelné pravidlo umožňuje, aby server klientu na požádání poslal spustitelný kód. Pod tímto kódem si lze představit různé programy, skripty, pluginy. Dříve bylo toto pravidlo využíváno technologiemi např. Java applet či Adobe Flash. Vzhledem k vysokým bezpečnostním rizikům bylo od těchto technologií upuštěno a nejpoužívanějším kódem na vyžádání se stal Javascript. [34]

3.3.2 Webové služby

V kontextu webových služeb je REST úzce spojený s HTTP. Vzhledem k tomu, že autor architektury REST je i jedním z autorů protokolu HTTP, lze mezi těmito technologiemi nalézt určitou podobnost, např. bezstavovost, použití mezipaměti apod. I díky tomu lze uvést, že REST je ideální k použití s HTTP, či obráceně. Lze se setkat s označením *RESTful*, jedná se o technologie, které splňují všechny požadavky architektury REST. [34]
Pro práci se zdroji a daty je využíváno HTTP metod. REST pomocí HTTP umožňuje kompletní CRUD operace – vytvoření, čtení, úpravy a mazání. HTTP metody lze použít například následovně:

- GET – Získání dat
- POST, PUT – Vytvoření dat
- PATCH, POST – Úprava dat
- DELETE – Mazání dat

Toto použití však není povinné a jednotlivé systémy mohou využít vlastní implementace. Teoreticky lze vytvořit plnohodnotnou webovou službu splňující CRUD za použití pouze dvou HTTP metod – GET pro získání a POST pro vše ostatní.

REST žádným způsobem neomezuje reprezentaci dat, a proto lze využívat jakýchkoliv datových formátů. Nejpoužívanějším formátem pro reprezentaci dat v RESTful webových službách je však JSON.

3.3.3 Zabezpečení

Obdobně jako u SOAP lze zlepšit zabezpečení systému využívajícího RESTful API (webové služby) použitím zabezpečeného protokolu HTTPS místo HTTP.

Webové služby lze zabezpečit přidáním nutnosti přihlášení (autentizace). Jinak řečeno, přidat nutnost, aby byl uživatel přihlášený, jinak nebude moci přistupovat k API. Toto lze zajistit například vytvořením koncového bodu (*endpoint*), který přijímá přihlašovací údaje a vrací unikátní token. S tímto tokenem lze poté přistupovat k dalším částím webové služby. Další častou možností bývá vytvoření tzv. API klíče v administraci. Pod tímto API klíčem lze poté přistupovat k vybraným endpointům.

3.3.4 Ukázky

Požadavek a odpověď pro získání dat může vypadat následovně.

```
// Požadavek
GET /users?id=136

// … HTTP hlavičky …

// Odpověď
{
  "data": [
    {
      "id":136,
      "firstname":"Karel",
      "lastname":"Novák",
      "profese":"Účetní"
    }
  ],
  "paging": {
    "page":1,
    "total":1587
  }
}
```

Zdrojový kód 3 – Ukázkový RESTful GET požadavek, vlastní zpracování

Požadavek žádá vrácení uživatelů a dle parametrů (id=136), žádá pouze uživatele s ID rovno 1. V odpovědi je vrácen uživatel s žádaným ID a informace o stránkování, včetně počtu všech záznamů.

Níže se nachází ukázka požadavku na vytvoření knihy.

```
// Požadavek
POST /books
{
  "title":"Staré pověsti české",
  "author":"Alois Jirásek"
}

// … HTTP hlavičky …

// Odpověď
{
  "id":536,
  "title":"Staré pověsti české",
  "author":"Alois Jirásek"
}
```

Zdrojový kód 4 – Ukázkový RESTful POST požadavek, vlastní zpracování

V tomto případě jsou zasílány na endpoint */books* informace o knize k vytvoření a server vrací uložené údaje včetně unikátního čísla vytvořeného záznamu.

Požadavek na smazání vytvořené knihy by mohl vypadat následovně.

```
// Požadavek
DELETE /books/536

// … HTTP hlavičky …

// Odpověď
OK
```

Zdrojový kód 5 – Ukázkový RESTful DELETE požadavek, vlastní zpracování

3.4 GraphQL

Graph Query Language, zkráceně GraphQL je dotazovací jazyk vyvinutý společností *Meta Platforms* (dříve *Facebook*). Byl vytvořen v roce 2012 a původně sloužil pro interní produkty společnosti. V roce 2015 byl zpřístupněn veřejnosti a v roce 2019 byla vytvořena organizace *GraphQL Foundation*, jejíž úkolem je spravovat specifikaci pro GraphQL. [35]

Obdobně jako u jiných technologií, tvoří GraphQL rozhraní, díky kterému získává nezávislost na platformách i programovacích jazycích. Hlavní a nejvýraznější výhodou je v GraphQL možnost si vyžádat části různých objektů bez nutnosti stahovat objekty celé. Klient si může požádat o jakékoliv dostupné části. Může vyžádat například pouze názvy knih a rok narození autora a server mu tyto údaje vrátí. Server tak neposílá zbytečné informace, které klient nepotřebuje. Zároveň odpadává nutnost posílat několik požadavků k získání informací k souvisejícím objektům. [35] V dříve uvedeném příkladě tak není nutné zasílat požadavek na objekty knih a oddělený požadavek autory, vše bude totiž vráceno v rámci jednoho požadavku a jedné odpovědi.

Aby klient mohl zažádat o data, potřebuje mít k dispozici informace o dostupných objektech. Informace o všech objektech dostupných na daném koncovém bodě GraphQL jsou uloženy a poskytovány přes GraphQL schéma. Definovány jsou zde všechny datové typy a objekty používané v dané službě, včetně všech vlastností a dostupných modifikačních metod. Každý požadavek musí být validní vůči GraphQL schématu. Nelze si např. vyžádat neexistující objekty. Výhodou schématu je i existence nástrojů, které umožňují z GraphQL schématu vygenerovat datové třídy v daném programovacím jazyce.

```
type Book {
   name: String!
   isbn: String!
   author: Author
}

type Author {
   firstname: String!
   lastname: String!
   birthdate: Date
   books: [Book]
}

type Query {
  books: [Book]
  authors: [Author]
}

type Mutation {
  addBook(title: String!, isbn: String!, author: String): Book
}
```

Zdrojový kód 6 – Ukázkové GraphQL schéma, vlastní zpracování

V ukázce Zdrojový kód 6 jsou definovány 2 objektové typy. Type kniha obsahuje povinné atributy jméno, ISBN a volitelný atribut Autor. Tento atribut je odkazem na druhý definovaný objektový typ. Autor má povinné atributy křestní jméno, příjmení a volitelné datum narození a seznam knih, které napsal. Typy Query a Mutation jsou speciální typy operací.

GraphQL definuje 3 druhy operací – dotazy, mutace a subskripce. Dotazy slouží k vyžádání dat. Ve schématu typ dotazu definuje na nejvyšší úrovni všechny druhy objektových typů, které lze vyžádat. Ukázka Zdrojový kód 7 obsahuje požadavek na vrácení názvů knih a roku narození jejich autora. V dolní části je ukázka, jak by mohla vypadat odpověď ze serveru.

```
query GetBooks {
   books {
      name
      author {
         birthdate {
            year
         }
      }
   }
}

// Odpověď
{
  "data": {
    "books": [
     {
       "name": "Staré pověsti české",
       "author": {
         "birthdate": {
             "year": 1851
         }
       }
     },
     ... další záznamy
    ]
  }
}
```

Zdrojový kód 7 – Ukázkový GraphQL dotaz, vlastní zpracování

GraphQL mutace mají podobnou strukturu jako dotaz (také jsou definovány ve schématu) ale na rozdíl od dotazů slouží k úpravě dat. Úpravou dat je v tomto případě rozuměna modifikace, přidání i smazání záznamů. Kromě toho je možné při volaní mutace také nadefinovat, která data má server vrátit. Na ukázce níže lze vidět příklad přidání nové knížky *R.U.R.* od Karla Čapka a vyžádání názvu dané knihy. Server následně vrací informace o vytvořené knize.

```
mutation AddBook {
  addBook(name: "R.U.R.", isbn: "978-80-7390-062-5", author: "Karel Čapek") {
    name
  }
}
// Odpověď
{
  "data": {
    "addBook": {
       "name": " R.U.R."
    }
  }
}
```

Zdrojový kód 8 – Ukázková GraphQL mutace, vlastní zpracování

V případě použití GraphQL přes HTTP se využívá pouze metod GET a POST. Pro dotazy se využívá metoda HTTP GET a pro mutace (vč. vrácení dat) se využívá HTTP POST. V případě mazání záznamů tedy není využívána metoda DELETE, ale pouze POST. [35] Za další nevýhodu lze vnímat malou podporu verzování. Přidávání nových vlastností je jednoduché, stačí dané změny přenést do schématu a kompatibilita se starší verzí zůstane zachována. V případě nutnosti změny či odstranění, ať už vlastností či celých typů, dotazů či mutací, nastává problém. Kvůli zachování kompatibility není ve většině případů možné změny instantně aplikovat. Je nutné informovat o změnách a aktualizovat všechny klientské aplikace.

GraphQL přináší velké výhody v možností vlastního definování žádaných dat a stejně jako u RESTful API využívá v přenosu dat převážně formát JSON. Obdobně jako jiné technologie k API má GraphQL i své nevýhody – např. absence verzování, ukládání do mezipaměti či nemožnost omezení požadavků. GraphQL je využitelný v mnoha případech, není však zcela univerzální a použitelný pro všechny možné systémy.

4 Dynamicky definované API

Dynamicky definované webové API je nadstavba nad REST API, která umožňuje dynamicky (za běhu aplikace) definovat nové koncové body API neboli endpointy. V rámci této práce je v rámci zjednodušení dynamicky definované API nazýváno též jako dynamické API. Dynamicky definované API lze považovat za architekturu API, která je navržena jako vysoce rozšiřitelná a univerzální. Implementace architektury musí splňovat požadavky popsané v kapitole 4.2. Po splnění požadavků může informační systém implementující architekturu využívat výhod dynamicky definovaného API zmíněných v kapitole 6.1.

4.1 Využití

V informačním systému (IS) implementujícím dynamicky definované API je umožněno za běhu informačního systému definovat nové koncové body webového API. V klasických IS implementujících RESTful API vyžaduje přidání nového koncového bodu zkompilování celé aplikace a následně nasazení na server. Proces přidání nového endpointu může trvat i několik týdnů, než se změny z vývoje dostanou až na produkční server. U větších a komplexnějších systémů může tento proces trvat i celé měsíce. Proces případných změn v již existujících koncových bodech může trvat ještě déle v rámci zachování kompatibility se staršími verzemi.

Dynamické API umožňuje proces správy koncových bodů výrazně urychlit. Přidání nových či úprava stávajících koncových bodů lze provést téměř instantně. Všechny koncové body jsou v dynamicky definovaném API abstraktně zachovány v uložišti a při volání daného bodu dochází k načtení definic a vykonání požadovaných činností. Dynamické koncové body API nejsou tedy součástí zdrojového kódu systému, ale jsou dodatečně načítány za běhu. Dynamické endpointy jsou schopny vykonat libovolnou činnost obdobně jako standartní koncové body. Implementující informační systém může dynamické API libovolně upravovat či rozšiřovat, měly by však být stále dodrženy následující požadavky.

4.2 Požadavky

K umožnění dynamického definování webového API je nezbytné nadefinování několika požadavků, bez kterých není možná implementace. Jedná se o obecné požadavky,

kde v případě jejich splnění lze usuzovat, že je možné v dané technologii provést implementaci z podstatné (či dokonce celé) části. Tyto požadavky jsou rozděleny do jednotlivých kategorií a jsou detailněji rozepsány níže. Požadavky jsou uvedeny v obecné rovině a lze je v případě potřeby mírně upravovat.

4.2.1 Technologické požadavky

Existuje velké množství různorodých technologií, platforem, programovacích jazyků apod. Vzhledem k nezávislosti dynamického API na platformě je možné využití libovolných technologií a nástrojů. Technologie pro implementaci musí splňovat všechny nároky z následujícího bodového seznamu:

- Webový server

Používaná technologie musí být schopna využívat služby webového serveru a komunikovat s ním. Webový server musí správně zajišťovat komunikaci mezi počítači pomocí protokolu HTTP. Verze HTTP není omezena, avšak ideální je verze HTTP/2 a vyšší. Autentizace na úrovni webového serveru není nutná, poskytuje-li ji, je nezbytné povolit anonymní autentizaci či nadefinovat vlastní (pro ověření API klíčů). Ukládání odpovědí do mezipaměti (cache) je doporučeno pro dynamické API vypnout. Mezi podporované webové servery lze zařadit *Apache*, *Microsoft-IIS*, *Node.js* a další.

- REST API

Používaná technologie a webový server musí podporovat webové API. Vzhledem k tomu, že dynamické API je navrženo jako nástavba na REST API, je nezbytná podpora REST API technologií i serverem.

- Middleware

Jedná se o podmíněný požadavek. V případě, že použitý webový server umožňuje definici vlastní autentizace, není nutná podpora middlewaru mezi webovým serverem a vyvíjeným informačním systémem. Vlastní autentizace by však měla spolehlivě zaručit ověření příchozího požadavku i API klíče. Využití middleware je však doporučeno k definování vlastních rozšíření informačního systému. Pokud webový server vlastní autentizaci nepodporuje, je využití middlewaru nezbytné.

- Mezipamět (Cache)

Jde o volitelný, avšak silně doporučený požadavek. Použitá technologie by měla umožňovat ukládat libovolné objekty do mezipaměti. Ukládání do mezipaměti by však mělo probíhat v rámci vyvíjené aplikace (informačního systému), ne na úrovni webového serveru. Ukládání do mezipaměti umožňuje výrazné zrychlení vykonání požadavku v dynamicky definovaném API. K ukládání dat do mezipaměti lze využít interní nástroje používané technologie (pokud je poskytuje), např. *MemoryCache* v *.NET*. Možné je ale využití i externích uložišť, např. *Redis*.

- Služby na pozadí

Další volitelný požadavek tvoří možnost vytváření služeb/úkolů na pozadí. Jedná se o speciální procesy, které běží na pozadí hlavní aplikace (informačního systému). Může se jednat o dlouho trvající procesy, opakující se procesy či o procesy, které není nutné začleňovat do primárního chodu systému a tedy např. logování, *Cron* úkoly apod. Služby na pozadí nemusí být nezbytně součástí vyvíjeného systému, ale mohou stát odděleně ve formě separátních aplikací neboli agentů. Tyto aplikace mohou být vytvořeny v libovolném programovacím jazyce a pouze vykonávají svůj kód na základně pokynů z hlavní aplikace, které následně vrátí výsledek. Přenos dat mezi hlavní aplikací a oddělenými agenty však může být složitý.

4.2.2 Požadavky na API

Předchozími požadavky je již určeno, že používaná technologie musí podporovat REST API. Ke správné funkcionalitě dynamického API je nezbytné nadefinovat dodatečné požadavky přímo na původní webového API. Pro první požadavek mohou v závislosti na použité technologii nastat následující 2 situace.

Pokud daná technologie umožňuje za běhu aplikace načíst nové koncové body API (endpointy), pak je tento požadavek splněn a lze přistoupit ke kontrole dalších požadavků. Načítání nových endpointů je možné u určitých technologií, které využívají směrování podle cest v souborovém systému. Směrování API podle souborů se stalo trendem převážně v Javascriptových frameworcích, např. *Next.js*, *Gatsby.js* a jiné. Tento požadavek však vyžaduje možnost vytvoření nového endpointu za běhu aplikace, což většina technologií nenabízí.

Pokud vybraná technologie neumožňuje vytvoření nových koncových bodů za běhu aplikace, pak je nutné, aby technologie umožňovala tzv. *Catch all* API cesty. *Catch all* cesta je speciální druh koncového bodu API, jenž udává, že veškeré požadavky v dané URL sub-cestě, které nejsou směrovány na jiný existující endpoint, budou zpracovány v tomto endpointu. Příklad: *Catch all* endpoint je definovaný jako "localhost/api/*". Všechny požadavky jdoucí na URL adresu "…/api/<cokoliv>" (např. "…/api/knihy/", "…/api/autor/edit") budou zpracovány právě zmíněným koncovým bodem. Je nezbytné, aby daný endpoint přijímal požadavky ve všech HTTP metodách. Tento endpoint poté přebírá celý požadavek včetně všech jeho parametrů. Obvykle je nutné jednotlivé parametry z požadavku získat dodatečně, stejně jako tělo požadavku. Parametry požadavku nelze přijímat jako parametry pro metodu daného endpointu, jelikož struktura a datové typy nejsou předem známy. V tomto případě jsou jednotlivé dynamicky definované endpointy odlišeny pomocí unikátních URL cest. Z přijímaného požadavku je tak přečtena celá URL adresa, na kterou byl odeslán, z té se získá sub-cesta podle které je identifikován dynamicky definovaný API endpoint a následně je zavolán.

Pokud není možné vytváření nových endpointů za běhu aplikace a zároveň daná technologie neumožňuje definovat *Catch all* endpointy, pak není možné využití dynamicky definovaného API a je nutné zvolit jinou technologii.

4.2.3 Požadavky na uložiště

Vzhledem k faktu, že koncové body jsou v dynamickém API definovány za běhu, je nutné jejich definice ukládat. Jediný obecný požadavek v této kategorii je tak tvořen nutností vybrané technologie spolupracovat s libovolným druhem uložiště. Uložiště by mělo být dostatečně rychlé a permanentní (data by měla být zachována i při vypnutí aplikace). Je možné kombinování vícero různorodých uložišť. Data mohou být například ukládána v souboru na disku, ale pro rychlý přístup mohou být uložena v mezipaměti aplikace. V uložišti by měly být uloženy hlavně definice jednotlivých endpointů, ale zároveň by mělo být uložiště přizpůsobeno pro možné budoucí rozšiřování. Struktura definic endpointů je předem známá a uložiště může být již předem připraveno (např. již vytvořené databázové tabulky). Způsob reprezentace dat v uložišti není žádným způsobem omezován. Doporučeno je využití relační databáze (např. *MySQL*, *MSSQL* a jiné), dle potřeby je však možné využít i odlišných druhy databází (např. *NoSQL*).

4.3 Napodobované funkcionality

Některé vlastnosti a zásady normálního REST přístupu k API nemohou být v dynamickém API splněny, a proto je nezbytné danou funkcionalitu napodobit. Dodatečně simulované funkce se snaží co nejvěrněji nahradit původní schopnosti, ale dosažení plně identického chování není v současném stavu možné. Problematice napodobované funkcionality se detailněji věnují následující kapitoly.

4.3.1 Možnost seskupování

V mnoha implementacích klasického REST API bývají jednotlivé endpointy sdruženy podle API řadičů (anglicky controllers). Řadiče jsou obvykle představovány ve formě objektových tříd, které poskytují endpointům další nezbytné informace. Jednotlivé koncové body bývají definovány metodami v dané třídě API řadiče. Vstupní parametry metody jsou poté i vstupními parametry koncového bodu. Použité technologie poté obvykle zajišťují automatické napojení a převedení příchozích dat z požadavku do jednotlivých instancí tříd daných parametrů. Ukázkový řadič pro manipulaci s knihami (*BooksController*) může obsahovat metody například na přidávání nových knih (*CreateNew(Book book)*), získávání stávajících (*GetBooks()*) či mazání starých (*DeleteBook(int id)*). Každá metoda je jedním koncovým bodem s určitou HTTP metodou. Zároveň jsou endpointy zapouzdřeny v daném řadiči.

K zachování této funkcionality je tedy nutné přidat možnost seskupení koncových bodů i dynamicky definovaném API. Tohoto je docíleno vytvořením nového abstraktního objektu pro skupiny endpointů, anglický název *EndpointGroups*. Tyto skupiny slouží k logickému začlenění obdobně jako řadiče. Na rozdíl od nich je v případě skupin povoleno, aby jakýkoliv koncový bod byl součástí několika skupin. Endpoint zároveň nutně nemusí být součástí nějaké skupiny. Jednotlivé skupiny by však měly být od sebe odlišitelné. Jak již bylo zmíněno, *EndpointGroups* jsou pouze logickým rozčleněním, které samo o sobě neposkytuje přímou funkcionalitu jako některé řadiče. Více o skupinách lze nalézt v kapitole 4.4.5 - Skupiny endpointů.

4.3.2 Problematika verzování

Mnoho REST webových služeb umožňuje verzování API. Obecně, verzování API nastává v momentech, kdy je rozhodnuto, že původní API již neplní dostatečně svůj účel a je třeba

ho změnit. Změny mohou být malého a velkého rázu, kdy nová verze představuje plně předělané API. Mezi výhody verzování lze zařadit možnost vytváření zpětně nekompatibilních změn v API. Pokud zůstane zachována původní verze API, budou stále fungovat aplikace používající původní verzi. Do nové verze lze poté zapracovat tzv. zlomové změny (anglicky breaking change) a aplikace mohou postupně přejít na novou verzi API. Množství verzí API není nikterak omezováno a postupem času může progresivně docházet k odstraňování starých a již nepodporovaných verzí. Určení verze API bývá nejčastěji z URL adresy anebo pomocí HTTP hlaviček požadavků. V případě použití verze v URL adrese může vypadat ukázková adresa třeba následovně: "https://api.google.com/**v3**/books", důležitá je zde část "/v3/", která značí, že jde o volání API verze tři. Pro každé verzované webové API by měla vždy být uvedena i výchozí verze. Pokud není v požadavku explicitně uvedena žádaná verze, je použita výchozí (obvykle nejnovější) verze.

Pokud daná služba umožňuje specifikaci verze pomocí HTTP hlaviček, musí být předem oznámeno jaké hlavičky přijímá. Zvolení názvů hlaviček je tedy zcela v režii webové služby, může se jednat například o hlavičku "X-API-Version", "API-Version" či jen "Version". V případě neuvedení hlavičky je postup totožný jako při neuvedení v URL adrese, aneb je použita výchozí verze API.

V dynamicky definovaném API bylo verzování povoleno zavedením nového objektu endpoint akcí. Dané akce jsou odlišeny verzí a jsou přiřazeny k jednomu endpointu. Daný endpoint zároveň nese informaci o jedné výchozí verzi. V momentě zavolání určitého koncového bodu, dochází k hledání endpoint akce dle specifikované verze, a pokud není verze explicitně zadána, je hledána akce dle výchozí verze. Po nalezení dochází k vykonání úkolu zadaného v nalezené akci. Akce koncových bodů jsou detailněji rozepsány v kapitole 4.4.2 – Endpoint akce.

4.3.3 Označení zastaralým

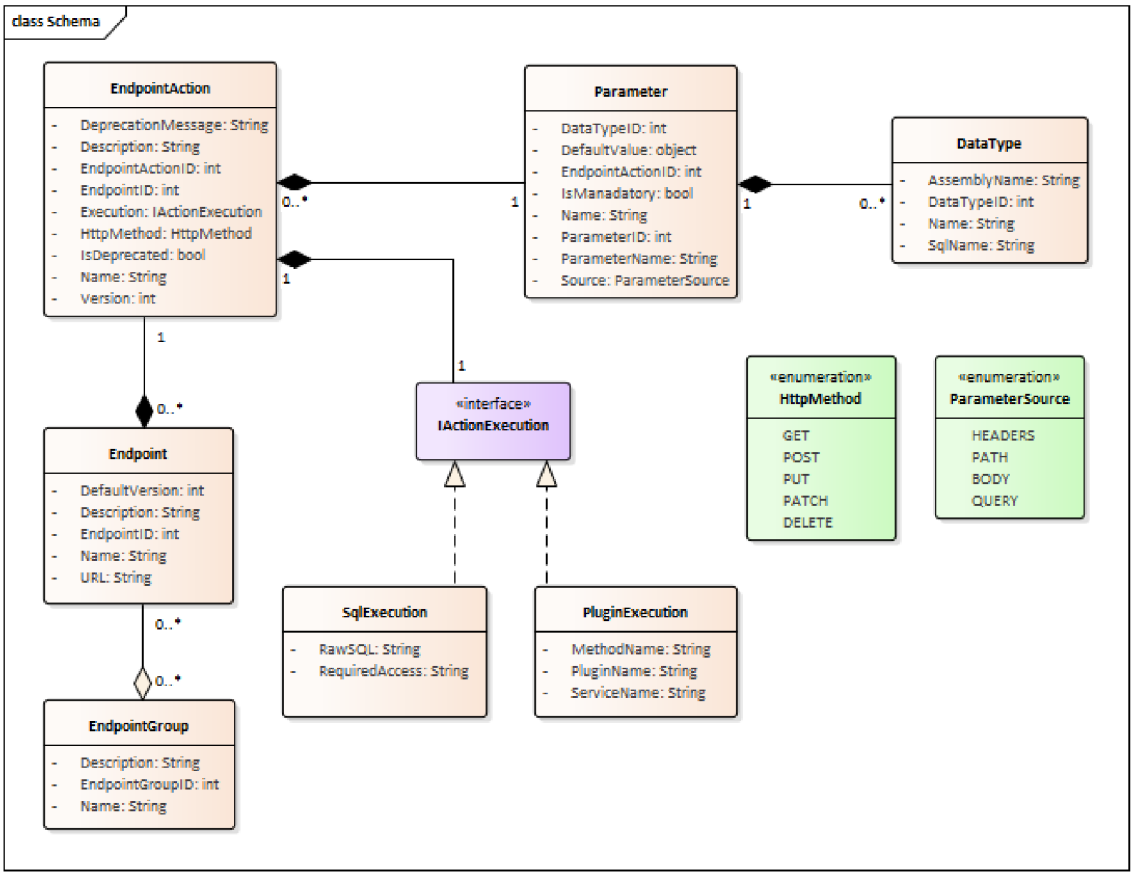
Často je nutné aplikace neustále rozvíjet, přidávat nové funkcionality, opravovat chyby apod. Obdobně je nutné progresivně aktualizovat i webové služby, resp. strukturu API a jejich objektů. Verzování webového API se věnuje předchozí kapitola 4.3.2 – Problematika verzování. U verzování je nezbytné pochopit, že průběhem času bývá neudržitelné zachování podpory všech starých verzí API. V některých případech může být

nutné provést změny, které ovlivní objekty původního API, pak se buď musí změnit původní API, či bude muset být odstraněno a nahrazeno novým. Někdy může i naléhavé odstranění staré verze API kvůli bezpečnostním rizikům. Staré verze však obvykle není možno rovnou odstranit, jelikož by přestaly fungovat navázané systémy a aplikace. Nejdříve proběhne označení daného koncového bodu jako zastaralý (anglicky obsolete/deprecated). Toto označení bývá ještě doplněno dodatečnou zprávou s informacemi o důvodu a možnostech použitelných alternativních endpointů. Označením se dává na vědomí vývojářům napojených systémů a aplikací, že daná verze by se již neměla používat a měli by přejít na verzi novější. V některých větších systémech bývají o změnách v API informováni externí vývojáři skrze blogy či přímé emaily.

V dynamicky definovaném API je označování zastaralých endpointů řešeno pomocí rozšíření endpoint akcí. Každou akci koncového bodu je možno označit jako zastaralou a přidat i dodatečnou informační zprávu o důvodu označení a o možnostech jiných použitelných akcí. V případě volání endpoint akce, která je označena jako zastaralá, měla by být ke každé odpovědi přidána informace o použití zastaralého koncového bodu včetně informační zprávy. Informace o zastaralé akci by měly být dostupné i administraci jednotlivých koncových bodů.

4.4 Schéma

V rámci dynamického API mohou být jednotlivé definice endpointů reprezentovány pomocí objektů. Reprezentace se netýká pouze endpointů, ale je nutné vyobrazit a pracovat s jednotlivými parametry i datovými typy. U každého objektu je nutné vyobrazit všechny důležité vlastnosti, jako například názvy, HTTP metody, adresy a další. Dynamicky definované API je však navrženo jako velice rozšiřitelné a každý objekt může být libovolně upravován a rozšiřován. Jednotlivé implementace dynamického API se tak mohou výrazně lišit. Jednu z mnoha možností návrhu dynamicky definovaného API vyobrazuje následující diagram (Obrázek 10).



Obrázek 10 – Schéma návrhu dynamického API, vlastní zpracování

Diagram vyobrazuje pouze základní elementy s jejich vlastnostmi. Jedná se pouze o obecné objekty nezbytné pro hlavní chod dynamicky definovaného API. Další aplikace/informační systémy mohou schéma volitelně rozšiřovat dle svých potřeb. Možné je rozšíření například přidáním objektu API klíčů, či objektů sloužících k napojení k již existujícímu systému.

Středobodem základního schématu je reprezentace endpointu a jeho akcí. Endpointy mohou obsahovat libovolné množství akcí, které jsou dále složeny z parametrů, kde každý parametr musí být určitého datového typu. Jakožto rozšíření je ve schématu ukázána možnost skupiny koncových bodů. Této a další problematice se věnuje přechozí kapitola 4.3 – Napodobované funkcionality.

4.4.1 Endpointy

Endpointy jsou hlavním objektem v dynamickém API. Jedná se základní objektovou reprezentaci koncového bodu v REST API převedenou a rozšířenou k dynamicky definovanému API.

Účel

Endpointy jsou reprezentaci koncových bodů ve webových službách implementujících RESTful API. Jedná se tedy o znázornění obdoby klasických koncových bodů. Objekt endpointů obsahuje pouze nejdůležitější vlastnosti pro chod dynamicky definovaného API a může být dále rozšiřován.

Vlastnosti

- EndpointID – přirozené číslo, povinné, unikátní

Představuje unikátní identifikátor daného koncového bodu. Kromě jednoznačné identifikace slouží tato vlastnost k jednoduchému vyhledávání specifického koncového bodu. Zároveň se jedná o přípravu pro implementaci schématu v databázi.

- Name (název) – text, povinný

Jedná se o vlastnost sloužící k rozlišení koncových bodů z uživatelského pohledu. Název by měl odpovídat účelu a funkci daného koncového bodu. Zároveň by měl být srozumitelný pro interní i externí vývojáře, neb se jedná vlastnost, která je veřejně přístupná.

- Description (popis) – text, volitelný

Popis je volitelným parametrem, který slouží přidání dodatečných informaci k danému koncovému bodu. Obdobně jako u názvu se jedná o veřejně přístupnou vlastnost, a tedy by obsah popisu měl být srozumitelný i pro externí vývojáře.

- URL – text, povinný, unikátní

Adresa URL udává část cesty URL, pod jakou je dostupný daný koncový bod. Cesta musí být unikání. V případě, že by bylo více koncových mělo stejnou cestu URL, nebylo by možné poté rozlišit, který koncový bod je volán. Je nezbytné zdůraznit, že definována by měla být pouze část cesty URL, ne celá adresa. URL adresa by měla být nastavena v implementovaném systému a cesta z koncového bodu je pouze příponou k hlavní adresa. Příkladem: V systému je nadefinována hlavní adresa jako "api.google.com/dynamic/", cesty v jednotlivých koncových bodech by mohly vypadat následovně "books/create", "books/edit" apod.

- DefaultVersion (výchozí verze) – přirozené číslo, povinné

Tato vlastnost definuje výchozí verzi API. V případě volání endpointu bez explicitně definované žádané verze API bude použita nastavená výchozí verze. K nastavené výchozí verzi by měla být vždy dostupná akce koncového bodu dané verze. Problematice verzování se detailněji věnuji kapitola 4.3.2.

Vazby

Každý koncový bod má relaci k endpoint akcím. Může existovat koncový bod, který nemá definovanou žádnou akci. V případě volání takového endpointu, však není možné nic vykonávat a systém implementující dynamické API by měl vrátit chybovou hlášku. Množství endpoint akcí není omezeno maximem, ale každá akce musí být odlišené verze. Koncové body mohou být zároveň součástí několika skupin endpointů. Množství není žádným způsobem omezeno a může nastat i situace, kdy koncový bod nebude součástí žádné skupiny. O problematice seskupování si lze více přečíst v kapitole 4.3.1.

4.4.2 Endpoint akce

Endpoint akce představují nedílnou součást koncových bodů. Vyjadřují samotný exekuční obsah koncového bodu, aneb co endpoint dané verze vykoná při zavolání. Zároveň představují nezbytný objekt pro umožnění verzování API.

Účel

Pomocí rozdělení objektu endpointů na endpoint akce je možné pod jedním koncovým bodem (URL adresou) vykonávat různé akce. Bez rozdělení by většina vlastností endpoint akcí byla součástí endpointů samotných a nebylo by možné definovat nové verze pod stejnou adresou URL. V endpoint akcích je zároveň definován průběh vykonávání práce daného koncového bodu. Jinými slovy je v jednotlivých akcích zadáno, co má být vykonáno při zavolání, čímž je i zadán účel dané akce.

Vlastnosti

- EndpointActionID – přirozené číslo, povinné, unikání

Jde o unikátní identifikátor dané akce koncového bodu. Jelikož endpoint může být složen z více akce, je nutné je od sebe odlišit. Z toho důvodu má každá akce své identifikační číslo. Tuto vlastnost lze použít jako primární klíč v případě ukládání do databáze.

- EndpointID – přirozené číslo, povinné

Jedná o identifikátor odkazující na koncový bod, jehož je součástí. Každá akce musí být součástí právě jednoho koncového bodu, a tak je i tato vlastnost povinná. V případě převodu do databáze, by byla tato vlastnost cizím klíčem.

- Name (název) – text, povinný

Obdobně jako u koncových bodů se i u akcí jedná se o vlastnost sloužící k rozlišení jednotlivých akcí z uživatelského pohledu. Název by měl také odpovídat účelu a funkci. Zároveň by měl být srozumitelný pro interní i externí vývojáře.

- Description (popis) – text, volitelný

Popis je stejně jako u endpointů volitelným parametrem, který slouží přidání dodatečných informaci. Obdobně jako u názvu by obsah popisu měl být srozumitelný i pro externí vývojáře.

- Version (verze) – přirozené číslo, povinné

Tato vlastnost specifikuje, které verze je daná akce koncového bodu. V rámci akcí jednoho endpoint by měla být verze vždy unikátní. Nesmí se tedy pod jedním koncovým bodem vyskytovat více akcí téže verze. První akce daného endpoint je vždy verze číslo 1. Další akce mají vždy verzi o jedna vyšší než akce poslední. Po vytvoření by již nemělo být umožněna změna verze.

- HttpMethod (HTTP metoda) – výčtový typ *HttpMethod*, povinný

Reprezentace HTTP metody udává, která HTTP metody musí být využita při volání dané akce. Bude-li akce volána jinou HTTP metodou, měl by systém implementující dynamické API vrátit chybu o špatně zvolené metodě. Pro vlastnost je použit výčtový typ, který obsahuje povolené HTTP metody. V ukázkovém případě se jedná pouze o *GET*, *POST*, *PUT*, *PATCH* a *DELETE*. Ostatní metody podporovány nejsou. Systémy si však mohou podporované metody nadefinovat samy.

- Execution (průběh akce) – objekt, povinný

Průběh akce je abstraktním rozšířením, které definuje, co se má vykonat při volání akce. Průběh může být libovolného typu a v diagramu je znázorněný rozhraním *IActionExecution*. Ukázkový průběh je vyobrazen 2 objekty. Jeden průběh může být například skrze exekuci SQL. V takovém případě je vykonán SQL dotaz, který je uložen

v definici. K SQL je nutná i přístupová úroveň, pod kterou má být daný dotaz spouštěn. Alternativou k SQL je spuštění externího kód, který je obsažen v tzv. pluginech. Více o plugin systému je popsáno v kapitole 4.5. V takovém případě je uložen název plugin, celý název dané služby v plugin a celý název metody v dané službě. V obou případech jsou předávány vstupní parametry z požadavku určeného v definici. Výstup z obou metod je poté předáván jako odpověď z koncového bodu.

- IsDeprecated (je zastaralý) – boolean, pouze ano/ne

Jedná se o jednoduchou vlastnost ano/ne, která značí, zda je daná akce označena jako zastaralá. V případě, že je akce zastaraná, mělo by po vykonání akce odesláno varování, že akce je zastaraná a neměla by se již používat. Více o problematice zastarání je popsáno v kapitole 4.3.3.

- DeprecationMessage (zpráva o zastaralé akci) – text, volitelný

Poslední vlastnost je aktivní pouze v případě, že je akce koncového bodu označena jako zastaralá. Ve zprávě mohou být obsaženy informace o náhradním použitelném koncovém bodu či dalších plánech s danou akcí. V případě, že není akce označena jako zastaralá, měla by tato zpráva být prázdná.

Vazby

Jak již bylo zmíněno výše, každá akce je součástí právě jednoho koncového bodu. Akce bez přiřazeného koncového bodu nesmí v systému dynamického API existovat. Druhou vazbu představuje navázání na parametry, přičemž se jedná o vstupní parametry pro průběh dané akce. Těchto parametrů může být libovolný počet.

4.4.3 Parametry

V rámci schématu jsou zachycovány pouze vstupní parametry. Reprezentují vstupní data pro průběh přiřazené akce. Parametry jsou důležitou součástí akcí koncových bodů. Skrze parametry jsou předávána vstupní data pro úspěšné vykonání činnosti volané akce. Každá akce musí být součástí právě jedné endpoint akce. Parametr bez přiřazené akce koncového bodu nesmí existovat. Současně musí být každý parametr nějakého datového typu. Datový typ je určen vazbou na objekt datových typů (viz. další kapitola).

Vlastnosti

- ParameterID – přirozené číslo, povinné, unikání

Jedná se o jednoznačný identifikátor parametru. V případě použití relační databáze, by se jednalo o primární klíč.

- EndpointActionID – přirozené číslo, povinné

Tato vlastnost odkazuje na připojenou endpoint akci. Jedná se o povinné napojení, a tedy nemůže existovat parametr bez navázané akce koncového bodu. V relační databázi jde o cizí klíč odkazují na objekt endpoint akce.

- DataTypeID – přirozené číslo, povinné

Obdobně jako u předchozí vlastnosti, odkazuje i tato na jiný objekt. Tentokrát však na objektovou reprezentaci datového typu. Pomocí této vazby je zadáno, jakého datového typu nabývá konkrétní parametr. Každý parametr musí mít přesně nastavený datový typ. V případě špatně zvoleného typu by nebylo možno předávat data akci koncového bodu. Taktéž by se v relační databázi jednalo o cizí klíč.

- Name (název) – text, povinný

Vlastnost názvu reprezentuje název vstupního parametru na vstupu z požadavku. Jedná se tedy o název, pod kterým jsou data přijímána od klienta. Název by neměl obsahovat mezery ani žádné speciální znaky. Příklad: Parametr s názvem "year" očekává, že z HTTP požadavku získá hodnotu pod daným názvem. Například z URL adresy "…/api/books?year=2023" bude získána hodnota 2023. Parametry jedné endpoint akce musí mít jedinečné názvy.

- ParameterName (název parametru) – text, povinný

Tato vlastnost se od normálního názvu liší tím, že reprezentuje název parametru na vstupu do exekuční metody. Hodnoty parametrů jsou tedy mapovány z "Name" na "ParameterName" a předávány pod správným názvem metodě dané endpoint akce. Příkladem: Metoda pluginu je definována jako "GetBooks(String author)". Parametr akce je nastaven s "Name = authorName" a "ParameterName = author". Akce koncového bodu tedy na vstupu od klienta z HTTP požadavku očekává parametr s názvem "authorName", např "…/api/books?authorName=Karel". Hodnota "Karel" je získána a předána jako

parametr s názvem "author" metodě "GetBooks". Tato vlastnost také musí mít v rámci jedné endpoint akce jedinečné názvy.

- Source (zdroj) – výčtový typ *ParameterSource*, povinný

Zdroj udává, v jaké části HTTP požadavku se má daný parametr nacházet. Pokud se v dané částí požadavku parametr se správným názvem nenachází, bude označen jako nenalezený. Další chod se dále odvíjí podle vlastnosti, zda je povinný (viz. další bodová odrážka). Zdroj je reprezentován výčtovým typem, který může nabývat hodnot *HEADERS*, *PATH*, *BODY* a *QUERY*. *HEADERS* značí, že parametr se bude nacházet v hlavičkách přijímaného HTTP požadavku. *PATH* udává, že parametr bude brán z URL adresy při volání požadavku. Podmínkou k tomuto zdroji je nutnost specifikovat proměnnou přímo v samotné URL adrese koncového bodu. *BODY* označuje, že parametr se musí nacházet uvnitř těla požadavku. A poslední možnost *QUERY* udává, že parametr bude brán z query parametrů v URL adrese.

- IsMandatory (je povinný) – boolean, pouze ano/ne

V případě je parametr nastaven jako povinný, je nutné, aby příchozí požadavek obsahoval ve správném zdroji validní hodnotu. V případě, že požadavek neobsahuje daný parametr, který je označen jako povinný, pak by měl systém implementující dynamické API vrátit chybu. Pokud parametr není nastaven jako povinný (je volitelný), pak musí obsahovat výchozí hodnotu.

- DefaultValue (výchozí hodnota) – objekt, volitelný

Výchozí hodnota reprezentuje hodnotu, která má být použita, pokud je parametr označen jako volitelný a zároveň není v požadavku obsažena jiná hodnota parametru. Pokud je parametr volitelný, ale v HTTP požadavku se nachází klientem specifikovaná hodnota, pak je výchozí hodnota ignorována a je použita poskytnutá hodnota. V případě, že je parametr označen jako volitelný, musí obsahovat výchozí hodnotu. Hodnota musí odpovídat přiřazeném datovému typu.

4.4.4 Datové typy

Jedná se o objektovou reprezentaci vlastních datových typů používaných v parametrech. Dynamicky definované API musí pro správné fungování vědět o datových typech všech parametrů. V případě špatně zvoleného datového typu není možné zaručit správnou

funkcionalitu systému. Systém může obsahovat předem definované datové typy. Mezi předem očekávané typy se mohou řadit celá čísla (integer), textové řetězce (string), logické hodnoty (boolean), reálná čísla (double, float) a jiné. Na rozdíl od ostatních objektů mohou datové typy existovat i bez přiřazených parametrů. Dokonce je možné, že datový typ bude využíván několika parametry najednou. Systém by měl ale implementovat funkcionalitu, která zajistí, že dlouhodobě nevyužívané datové typy budou promazány v rámci zajištění úspory místa na disku. Jedinou vazbou je tedy napojení na parametry. Samozřejmostí je i možnost dalších rozšíření dle požadavků implementujícího systému.

Vlastnosti

- DataTypeID – přirozené číslo, povinné, unikání

Identifikátor datového typu umožňuje jednoznačné odlišení jednotlivých datových typů. V případě relační databází by tato vlastnost mohla být použita jako primární klíč.

- Name (název) – text, povinný, unikátní

Název představuje v tomto případě uživatelsky přívětivý název datového typu. Může tedy obsahovat libovolné speciální znaky i mezery. Podmínkou však je, že název musí být unikátní, aby v případné administraci nedocházelo k záměnám datových typů. Názvy mohou být například následující "Celé číslo (int)", "Datum", "Datum a čas". Zvolení anglický názvů by bylo výhodnější. Nejlepší variantou může být použití klíčů pro případnou lokalizaci.

- AssemblyName (název v balíčku) – text, povinný

Tato vlastnost vyjadřuje celý název datového typu v balíčku. Musí se jednat o přesný název, jelikož pomocí těchto hodnot jsou vyhledávány přímo datové typy v použité technologii. Názvy nemusí být unikátní, v dynamickém může být definováno více instancí datových typů se stejným názvem. Příkladem tohoto může být typ datumu a času. Existují technologie, které neumožňují odděleně definovat datum a čas. V takovém případě by mohly být definované datové typy s "*Name* = Datum" a "*Name* = Datum a čas", přičemž u obou instancí by byl název balíčku nastaven jako "System.DateTime". Z přechozí příkladu je i zřejmé, jak mohou vypadat hodnoty pro názvy v balíčkách, např. "System.String", "System.Int32", "SomePackage.XYZ.Abc" apod.

- SqlName (název v SQL) – text, volitelný

Jedná se o volitelné rozšíření, ve kterém je pro každý datový typ navíc definován celý název typu v jazyce SQL. Může se jednat například o hodnoty "INT", "VARCHAR", "DATE" apod. Systémy, které nebudou využívat exekuční obsah v SQL nemusí tuto vlastnost zahrnovat. Jedná se pouze o ukázku možností dynamicky definovaného API. Jednotlivé implementace mohou libovolně upravenou strukturu.

4.4.5 Skupiny endpointů

Skupiny koncových bodů slouží k logickému rozčlenění jednotlivých endpointů do skupin. Koncové body mohou být součástí několika skupiny a stejně tak mohou skupiny obsahovat několik endpointů. Existence skupiny bez žádných koncových bodů je povolena. Problematice seskupovaní se podrobněji věnuje kapitola 4.3.1.

Vlastnosti

- EndpointGroupID – přirozené číslo, povinné, unikání

Vlastnost představuje unikátní identifikátor dané skupiny. V případě užití relačních databází by se jednalo o primární klíč.

- Name (název) – text, povinný, unikátní

Název skupiny musí být vždy unikání, aby se zamezilo možnost záměny jednotlivých skupin. Hodnota může obsahovat mezery i libovolné speciální znaky. Skupina by však měla být uživatelsky přívětivě nazvaná.

- Description (popis) – text, volitelný

V rámci rozšíření je umožněno skupině nastavit i popis, kde mohou být specifikovány libovolné dodatečné informace. Může zde být obsažen například účel skupiny.

4.5 Plugin systém

Akce koncových bodů mohou vykonávat libovolnou činnost. Tyto činnosti mohou být obsaženy v tzv. pluginech neboli zásuvných modulech. Jedná se o balíček tříd splňující rozhraní, které poskytuje implementovaný informační systém. Systém je následně schopný zásuvné moduly načíst a volat jednotlivé metody z modulu dle definice v dynamickém API. Obecný systém pro zásuvné moduly může být jakýkoliv, ale v této kapitole bude zaměřen pouze v kontextu pro dynamicky definované API.

4.5.1 Průběh vývoje

Před samotným vývojem pluginu je nezbytné určení účelu neboli co bude daný plugin dělat. Jednotlivé pluginy by měly být vytvořeny v totožné technologii, jako je systém samotný. Informační systémy mohou podporovat načítání modulů v jiných technologiích, v takovém případě je možné použití podporované technologie. Projekt se zdrojovým kódem modulu musí načíst knihovnu, již poskytuje daný informační systém. Knihovna obvykle obsahuje všechna důležitá rozhraní a služby. Součástí knihovny by měly být i důležité veřejné objekty systému, se kterými může plugin pracovat. Po naimportování sdílené knihovny může začít implementace nezbytných rozhraní. V rámci implementace by měl být splněn účel pluginu. V průběhu vývoje byl měl být modul samozřejmě průběžně testován, aby se předešlo jakýmkoliv neošetřeným výjimkám a chybám. Po kompletní implementaci a úspěšném otestování, může být zásuvný model zkompilován a připraven k nasazení. Některé informační systémy mohou podporovat interní "obchody", kam může být modul nahrán a následně automaticky nainstalován na jednotlivé instance. Pokud systém obchody nenabízí, je možnost implementovat vlastní načítání skrze souborový systém. Veškeré implementace jsou zcela v režii vyvíjeného informačního systému.

4.5.2 Společné rozhraní

Jak bylo zmíněno výše, zásuvné moduly musí splňovat společné rozhraní. Rozhraní je definování hostujícím informačním systémem skrze sdílenou knihovnu. V rámci rozhraní jsou definované povinné metody, které musí daný modul implementovat. Definice může být jednoduchá ve stylu jedné metody, která přijímá požadavky a vrací odpověď. Definování tímto stylem by ale mohlo být nedostatečné a je doporučeno použít vlastní komplexnější řešení dle potřeb. Součástí společných rozhraní by mělo být rozhraní, ve kterém budou definovány informace o samotném pluginu. Mezi těmito informacemi může být název, autor, verze pluginu apod. Rozhraní jsou taktéž zcela v režii vyvíjeného systému, pravidla pro dynamicky definované API jej nikterak neomezují.

4.5.3 Výhody a nevýhody

Mezi hlavní výhody patří možnost skoro neomezeného rozšíření. V pluginech lze vykonávat libovolně komplexní činnost, která může posunout exekuci koncového bodu

na novou úroveň. Uvnitř zásuvného modulu může být vykonáváno skutečně téměř cokoliv. Tato možnost může být ale i nevýhodou. Je nutné zvážit bezpečnostní rizika. Neznámý modul totiž může přinést i nevyžádaný škodlivý kód. Před použitím daného pluginu je nezbytná kontrola. Kontrola může být v podobě antiviru či v podobě kontroly přes kompetentní osobu (např. vývojáře). Jako další nevýhodu lze považovat ztrátu kontroly ze strany vývojářů systému (tzv. core tým). V momentě, kdy je umožněn vývoj plugin externím vývojářům, ztrácí core tým kontrolu nad možnými změnami skrze pluginy.

4.6 SOAP

Jakožto nadstavba nad REST API dynamicky definované API v základu nepodporuje SOAP. Ale díky velké rozšiřitelnosti je možné vytvořit rozšíření, které bude SOAP napodobovat. Jak již zmíněno v kapitole 3.2 věnované SOAP, má tento protokol jasně danou strukturu požadavků a odpovědí. Dynamické API umožňuje přijímání i odesílání dat v libovolném formátu. Na vstupu přijímaná data ve formátu XML není složité rozebrat na důležité části a získat z nich informace potřebné k vykonávání (např. název koncového bodu a parametry pro akci). Výstup může být také opět poměrně jednoduše složen a odeslán ve správném formátu SOAP zprávy. Vygenerování WSDL dokumentu je možné díky seznamu definicí všech koncových bodů, akcí, parametrů i datových typů. Hlavní problém představuje problematika verzování, které není v SOAP z určité částí podporováno. V rámci jedné SOAP webové služby může být nadefinován právě jeden koncový bod. Jednotlivé SOAP metody mohou být reprezentovány pomocí endpoint akcí, avšak musí být upravena pravidla pro verzování. Výsledkem může být fungující SOAP API, které co nejvíce napodobuje skutečné SOAP.

5 Implementace

Pro lepší představu byla v rámci této bakalářské práce vytvořena velice jednoduchá implementační ukázka dynamického API. Zdrojové kódy jsou přiloženy k elektronické verzi této bakalářské práce. Pro implementaci byla využita technologie *.NET 7* od společnosti Microsoft. Primárním programovacím jazykem je tedy jazyk C#. Framework *.NET 7*, přesněji jeho součást s názvem *ASP.NET*, je jednou z mnoha technologií, které splňují všechny technologické požadavky definované v kapitole 4.2.1. *ASP.NET* umožňuje obsluhu webových služeb, avšak za běhu aplikace není možné vytvářet nové koncového body. Je ale dovoleno definovat *Catch all* koncové body, čímž jsou splněny požadavky na API z kapitoly 4.2.2.

V rámci ukázky byla využita relační databáze *MSSQL*, která taktéž pochází z dílny společnosti Microsoft. Vybraná databázová technologie splňuje požadavky na uložiště z kapitoly 4.2.3. Zvolení technologií od stejné společnosti poskytuje velké výhody vzhledem k vzájemné kompatibilitě a jednoduchého propojení. Pro zjednodušení komunikace mezi aplikací a databází je v ukázce využito objektově relační mapování pomocí technologie *Entity Framework*, původem také od společnosti Microsoft.

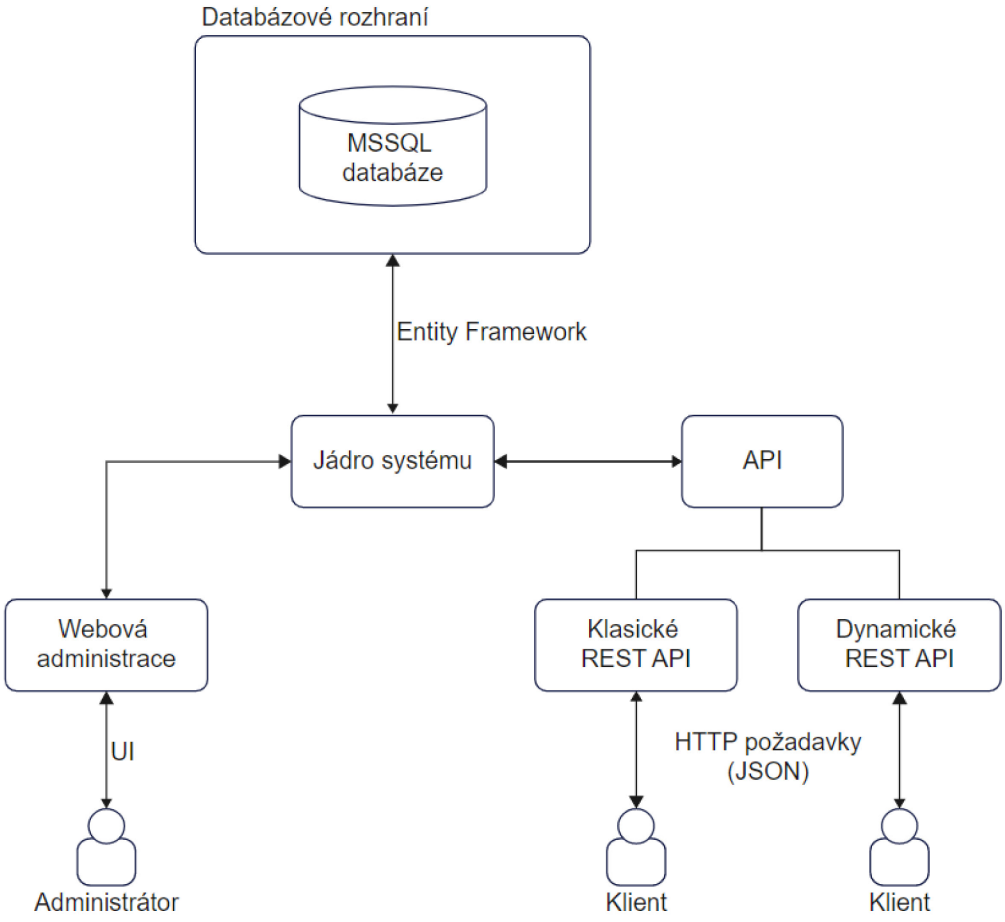
Aby bylo možné jednoduše vytvářet nové koncové body, je vhodné, aby měla hostitelská aplikace uživatelské prostředí. Toto prostředí může mít podobu konzolovou nebo grafickou. Pro ukázku bylo zvoleno grafické uživatelské prostředí v rámci webu. Pro vykreslování webových stránek je použit webový framework *Blazor* od Microsoftu. Pro snadnější a vizuálně pohlednější prostředí je instalována knihovna *MudBlazor*, která obsahuje mnoho již hotových komponent se styly.

V ukázce probíhá komunikace s API pouze za použití formátu JSON. Jiné formáty nejsou pro zjednodušení podporovány. Manipulace s JSON dokumenty probíhá skrze integrovanou knihovnu *System.Text.Json*, která umožňuje rychlou serializaci i deserializaci textu.

5.1 Architektura

Obrázek 11 níže vyobrazuje diagram architektury ukázkové implementace dynamicky definovaného API. Hlavní součást v implementovaném systému představuje jeho jádro. Součástí jádra jsou různé hlavní služby nezbytné pro chod informačního systému,

včetně služby dispečera pro distribuci požadavků jednotlivým komponentám. Pro zjednodušení je však v diagramu jádro vyjádřeno abstraktně. Skrze jádro také probíhá veškerý přístup k datům v databázi i veškerá komunikace mezi dalšími částmi systému. Správa databázových dat probíhá pomocí *Entity Frameworku* skrze databázové rozhraní. S jádrem následně přímo komunikuje webová administrace a webové API. Webová administrace je přístupná pomocí webového uživatelského prostředí administrátorům. Webové API je rozděleno na klasické (nedynamické) REST API a dynamické REST API. K oběma typům se připojení uskutečňuje pomocí HTTP požadavků a odpovědí s formátem JSON. K API se mohou připojovat libovolné klientské aplikace mající dostatečné oprávnění.



Obrázek 11 – Architektura ukázkové implementace, vlastní zpracování

5.2 Požadavky ke spuštění

Pro spuštění implementované ukázky dynamicky definovaného API na vlastním zařízení je nutné, aby byl nainstalován následující software:

- Windows 10+
- SDK pro .NET 7+
- Visual Studio 2022+
- MSSQL Server 2019+
- Internet Information Services (IIS) nebo IIS Express
- Doporučeno: SSMS nebo jiný SW pro správu databáze

V případě verzí se jedná o nejstarší doporučené verze. Symbol plus (+) za verzí daného softwaru nahrazuje text "nebo novější".

Soubor s řešením (.sln) musí být otevřen v programu *Visual Studio*, kde se následně načte samotný ukázkový projekt. Projekt má standardní strukturu s množstvím komentářů u nejdůležitějších částí, orientace ve zdrojových kódech by tak neměla být složitá. Před spuštěním je doporučeno spustit příkaz "dotnet restore", který stáhne a doinstaluje chybějící knihovny, poté by již mělo být možné ukázkovou aplikaci spustit.

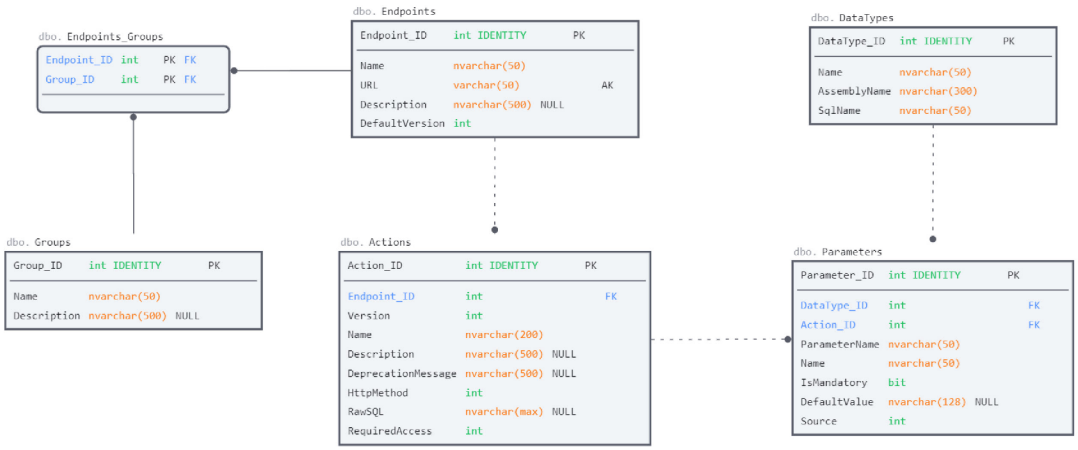
5.3 Databáze

Jak již bylo zmíněno v předchozích kapitolách, pro jednoduchou ukázku implementace dynamicky definovaného API byla zvolena databáze *MSSQL*. Datová struktura použitých tabulek je vyobrazena na obrázku níže. Hierarchicky nejnižší tabulka je tabulka datových typů. Každý datový typ je složen ze svého unikátního identifikátoru, názvu, názvu v sestavení a názvu v SQL. Stejně jako u všech tabulek je pro unikátní identifikátory zvolen datový typ celého čísla s automatickou inkrementací. Vlastnost názvu představuje název, který by měl být uživatelsky srozumitelný. Oproti tomu zbylé vlastnosti musí být pouze strojově srozumitelné. Název v sestavení představuje přesný identifikátor typu v použité technologii (*.NET*). Např. Celá čísla jsou v jazyce C# reprezentována třídou *System.Int32*. Obdobně název v SQL reprezentuje přesný název datového typu v jazyce SQL, a tedy například pro celá čísla je datový typ nazván jako *INT*. Datové typy jsou spojeny relací 1:N s tabulkou parametrů.

Parametry reprezentují parametr daného koncového bodu. Tabulka obsahuje přesný název parametru, který musí být totožný s názvem v SQL i s názvem přijímaném HTTP požadavku. Dále pak název, který by měl být uživatelsky přívětivý. Další vlastnost tvoří informace o povinnosti parametru – zda je povinný, a pokud ne, tak jaká je výchozí hodnota. Poslední vlastnost tvoří zdroj, odkud je parametr přijímán. Každý parametr je součástí právě jedné endpoint akce. Zatímco akce mohou mít několik parametrů. Relace mezi těmito tabulkami je tedy 1:N.

Jednou z nejdůležitějších entit v ukázkovém systému jsou akce koncových bodů. Každá akce má definovanou svou verzi, která umožňuje verzování API. Dále pak název akce, popis a případná zpráva o zastarání dané akce. Nedílnou součástí je i nastavení HTTP metody, pod kterou má být daná akce přijímána a úroveň přístupu, pod kterou má být vykonána. Hlavní vlastnost je poté SQL dotaz ve formě nezpracovaného textu. Každá akce je součástí právě jednoho koncového bodu a zároveň koncové body mohou být složeny z několik akcí. Mezi endpoint a akcemi je relace 1:N.

Koncové body neboli endpointy jsou složeny z uživatelsky přívětivého názvu, popisu, výchozí verze akcí a částí URL adresy, pod kterou je dostupné volání daného koncového bodu. Pro lepší strukturování je přidána relace M:N s tabulkou skupin, která je složena z názvu a popisu.



Obrázek 12 – Struktura ukázkové databáze, vlastní zpracování

5.4 Ovládání

Ovládání aplikace je již dle architektury rozděleno na 2 části. První část tvoří webová administrace, která je složena z několika důležitých stránek. Hlavní stránka slouží

ke správě koncových bodů. V základním pohledu je tvořena seznamem všech dynamicky definovaných koncových bodů. Jednotlivé koncové body lze mazat pomocí příslušného tlačítka u daného endpointu. V tomto pohledu je také možné definovat endpointy nové a editovat stávající. Pro vytváření nového endpointu či editaci existujících je využívána stejná webová komponenta. V této komponentě lze nastavit všechny vlastnosti endpointů. Zároveň je umožněno vytvářet nové endpoint akce.

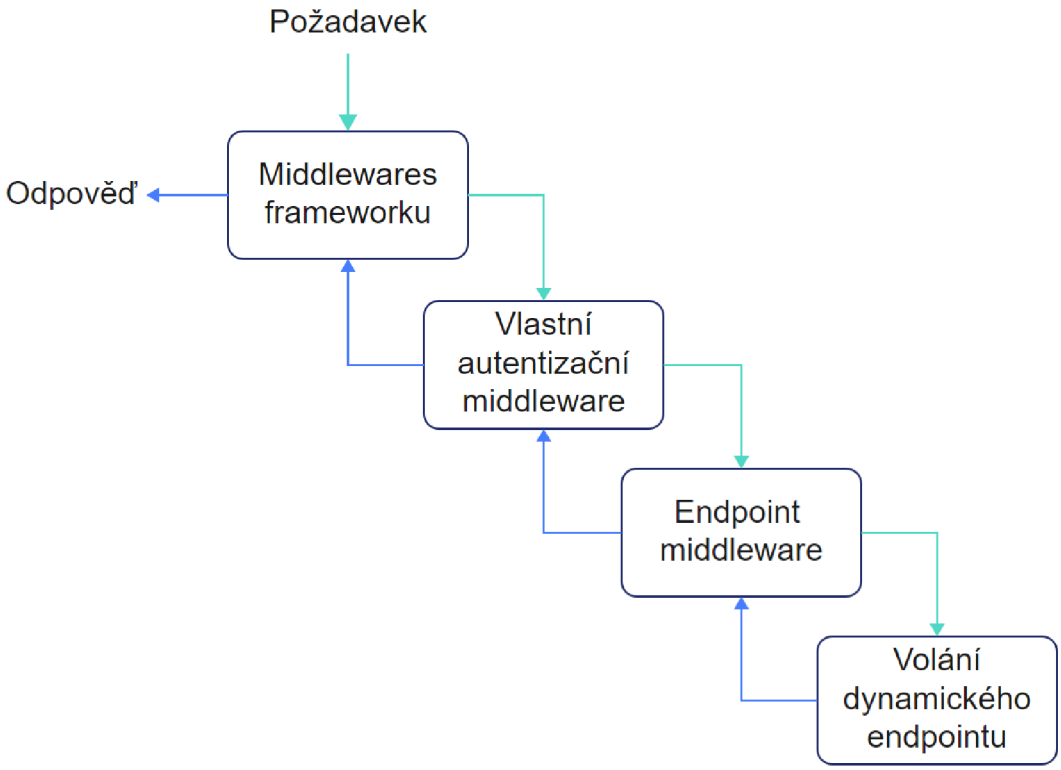
Úprava stávajících či vytváření akcí nových je součástí druhé z důležitých stránek webové administrace. Stejně jako o koncových bodů, je u akcí uveden kompletní seznam všech definovaných endpoint akcí. Akce je možné taktéž mazat, upravovat či vytvářet nové. Pro vytváření i úpravy je opět využita společná komponenta, která obsahuje textová pole pro všechny vlastnosti akcí koncových bodů. Nedílnou součástí editačního formuláře je se pole pro vložení SQL kódu, který se má vykonat při zavolání dané akce. Obsahuje-li použitý SQL příkaz parametry, je nezbytné parametry nadefinovat v tabulce pod příkazem. Každý použitý parametr musí obsahovat přesný název z SQL příkazu i přesný datový typ. Každý datový typ musí být reprezentovatelný jak v jazyce SQL, tak i v C#. Je nezbytné uvést přesný název v sestavení (např. *System.Int32*, *System.String* apod.) i přesný název v SQL (např. *INT*, *VARCHAR*).

Druhá část ukázkového systému (aplikace) představuje webové API. Volání dynamického API je dostupné na adrese "{URL_aplikace}**/api/**{URL_endpointu}". V takovém případě je volána akce dle nastavené defaultní verze na endpointu. Pro explicitní volání jiné verze, je možné volat adresu "**api/v**{verze}/{URL_endpointu}". Pokud žádaná verze endpoint akce existuje, bude zavolána, v opačném případě bude použita defaultní verze. Klasická verze API, tedy REST API, kde jsou koncové body přímou součástí systému, je dostupná na adrese "{URL_aplikace}/**webapi**/{různé koncové body}". Jestliže by bylo nutné vytvářet či upravit endpointy v klasickém API, bude nezbytné vytvořil celé nové zkompilování a nasazení projektu.

5.5 Middleware

V kontextu této práce je pod pojmem middleware myšlen specifický software, který tvoří novou vrstvu při zpracovávání požadavků. V technologii *.NET* je middleware tvořen třídou, které jsou při vykonávání předány 2 parametry. První je instance třídy *HttpContext*, která obsahuje informace o požadavku i odpovědi. Druhý parametr tvoří

odkaz na zavolání dalšího middlewaru. Každý middleware si může určit, zda bude exekuce aktuálního požadavku pokračovat či má skončit (např. chybovou hláškou). Při přijetí HTTP požadavku jsou nejdříve postupně volány middlewary přímo implementované v technologii, například ke směrování, autentizaci, či k obsluze nečekaných výjimek. Po zavolání posledního middleware následuje vykonání samotného koncového bodu. Poté následuje vrácení řízení middlewarům v obráceném postupu a na závěr vrácení HTTP odpovědi. Průběh zpracování požadavků je vyobrazen na obrázku níže.



Obrázek 13 – Průběh požadavků s middlewary, vlastní zpracování

V rámci implementované ukázky jsou dostupné 2 middlewary. První slouží jako ukázka možného autentizačního middlewaru při použití ověřování pomocí API klíčů. Kód nejdříve zjistí, zda je požadavek směrován na adresu API a zda je v požadavku obsažena hlavička s API klíčem. Poté následuje ověření daného klíče a případné vrácení chyby.

```
/// <summary>
/// Handle incoming request
/// </summary>
/// <param name="context">HTTP context</param>
/// <param name="next">Next action</param>
public async Task InvokeAsync(HttpContext context, RequestDelegate next)
{
    // Check if request is to API
    if (!context.Request.Path.StartsWithSegments("/api"))
    {
        // Request is not to API → ignore
        await next(context);
        return;
    }
    // Request to API → handle action

    // Check if headers contains API key
    if (context.Request.Headers.ContainsKey("api-key"))
    {
        // Get API key header
        var apiKeyHeader = context.Request.Headers["api-key"];

        // Look for API key in database
        var apiKey = await _apiKeyService.GetApiKeyAsync(apiKeyHeader);

        // Check if API key was found
        if (apiKey == null)
        {
            // API key not found
            context.Response.StatusCode = 401; // Unathorized

            await context.Response.WriteAsync("API key not valid");

            // Request is unauthorized → do NOT call next(...)
            return;
        }
        // Authorize API key
        await _apiKeyService.AuthKeyAsync(apiKey);
    }
    // Call next action
    await next(context);
}
```

Zdrojový kód 9 – Ukázkový middleware k API klíčům, vlastní zpracování

V případě druhého middlewaru se jedná o načtení dynamických koncových bodů, resp. informací o nich. Opět je kontrolováno, zda je požadavek směrován na adresu dynamického API, následně probíhá načtení endpointu dle URL cesty a při úspěšném načtení i uložení informací koncového bodu do mezipaměti přístupné dalšími službami. Ukázkový kód, ve kterém probíhá samotná kontrola a uložení informací o koncovém bodu, je k nahlédnutí níže.

```
/// <summary>
/// Look for endpoint and register it if found
/// </summary>
/// <param name="context"></param>
/// <returns>True if success</returns>
private async Task<bool> RegisterEndpointAsync(HttpContext context)
{
    // Ignore '/api'
    var path = context.Request.Path.Value[4..];

    // Remove slash at start and end
    path = path.Trim('/');

    // Replace version info
    path = ApiConstants.VersionRegex.Replace(path, string.Empty, 1);

    // Check if path is empty
    if (string.IsNullOrWhiteSpace(path))
    {
        // Endpoint was not found
        return await EndpointNotFound(context);
    }

    // Look for endpoint in database
    var endpointResult = await _endpointService.GetByUrlAsync(path);
    if (!endpointResult.IsSuccess)
    {
        // Error in database
        return await Error(context, endpointResult.Exception.Message);
    }

    // Get endpoint
    var endpoint = endpointResult.Item;
    if (endpoint == null)
    {
        // Endpoint was not found in database
        return await EndpointNotFound(context);
    }

    // Pass current endpoint to other services
    context.Items[ApiConstants.EndpointContextKey] = endpoint;

    // Return successs
    return true;
}
```

Zdrojový kód 10 – Ukázka registrace endpoint, vlastní zpracování

5.6 Důležité služby

V rámci implementované ukázky dynamického API je použito více vrstev různých služeb. Nejvyšší vrstva slouží k odstínění vrstev nižších a zároveň se stará o obsluhu neočekávaných výjimek. Tato vrstva je také implementací předem připravených rozhraní. Nižší vrstva se stará o validaci dat a komunikaci s datovou vrstvou. Datová vrstva je v ukázkovém projektu implementována skrze objektově-relační mapování za použití technologie *Entity Framework*.

Pro webovou administraci jsou připraveny služby nejvyšší vrstvy. Tyto služby umožňují vytváření, editaci, získávání a mazání všech dostupných entity evidovaných v ukázkovém systému. U některých služeb je dostupné i filtrování, například získání záznamu pouze podle jeho unikátního identifikátoru.

O zpracování požadavku na dynamicky definované koncové body se stará speciální služba nazvaná *ApiService*. V rámci této služby probíhá načtení a kontrola verzí koncových bodů. Načtení, validace parametrů a jejich následné mapování. Následuje vykonávání samotného SQL kódu v dané akci koncového bodu a ověření vrácených dat. Vykonávání SQL příkazu je ošetřeno proti útoku SQL injection za použití integrované knihovny *System.Data.SqlClient*. Výsledek je vracen ve formátu JSON. Pro znázornění je níže vyobrazena metoda sloužící k nalezení a vykonání endpoint akce dle poskytnuté verze.

```
public async Task<IActionResult> ExecuteAsync(Endpoint endpoint, int? version)
{
    if (!endpoint.Actions.Any())
    {
        // Endpoint contains no actions
        return new NotFoundObjectResult("No actions defined");
    }

    // If version is not explicitly defined, use the default version
    var usedVersion = (version == null) ? endpoint.DefaultVersion : (int)version;

    // Look for the action with the specified version
    var action = endpoint.Actions.FirstOrDefault(x => x.Version == usedVersion);
    if (action == null)
    {
        return new NotFoundObjectResult("Version not found");
    }

    // Execute the action
    return await ExecuteAsync(action);
}
```

Zdrojový kód 11 – Ukázková metoda API služby, vlastní zpracování

5.7 Zabezpečení

Vzhledem k definici, že dynamicky definované API lze považovat za nadstavbu RESTful API, je možné využívat veškeré možnosti zabezpečení, které jsou možné i v samotné implementaci REST API. Kromě možností z nativního REST API lze při použití middlewarů nadefinovat vlastní úrovně zabezpečení. Lze tak vytvořit vlastní systém pro API klíče, či naimplementovat zcela novou možnost zabezpečení. Při vlastní implementaci je však nutné vzít ohled na všechna s tím spojená rizika. Totožně jako u klasického webového API není ani dynamicky definovaného API radno podceňovat bezpečnost. V implementované ukázce však v rámci zjednodušení nejsou implementovány uživatelské účtu, ani autentizace a ani autorizace. Ukázkový informační systém je určen k lokálnímu spuštění pouze na jednom zařízení, díky čemuž není nutné jakékoliv komplexní zabezpečení.

5.8 Praktické využití

Kromě výše zmíněné ukázkové implementace dynamicky definovaného API existuje i mnohem komplexnější a rozsáhlejší implementace architektury. Tato implementace je však proprietárním softwarem a není možné zveřejnění jejího obsahu vzhledem k nastavené licenci. Se souhlasem společnosti je však dovoleno zmínit, že komplexní implementace je vytvářena mezinárodní společností Lineup Systems zaměřující se na poskytování informačních systémů a nástrojů na správu reklam pro mediální společnosti. Implementace dynamicky definovaného API je zde poskytována jako součást projektu *Amplio*, což je informační systém sloužící výhradně vydavatelům k optimalizaci a správě předplatných pro koncové zákazníky. Komplexní implementace je poskytována všem zákazníkům, kteří si mohou zvolit, zda jej využijí či nikoliv. Některé specifické žádosti zákazníků jsou řešeny právě s pomocí dynamického API, které umožňuje rychlé splnění požadavků bez nutnosti vydávání a nasazování celé nové verze informačního systému.

6 Porovnání klasické vs. dynamické API

V rámci návrhu architektury dynamického API je nezbytné porovnat výkon, výhody a nevýhody s klasickými webovými služby implementující REST API. Je však nutné poznamenat, že jednotlivé implementace dynamicky definovaného API mohou být výrazně odlišné, zmíněné nevýhody se jich nemusí týkat a zároveň mohou disponovat i vyšším výkonem. Níže zmíněné výhody a nevýhody se tedy týkají obecného návrhu využívajícího základní implementaci dynamického API.

6.1 Výhody

Největší výhodu přináší dynamicky definované webové API v podobě možnosti vytvářet nové koncové body API za běhu informačního systému. Díky této vlastnosti je možné mnohem rychleji splnit požadavky zákazníků. Vyžádá-li si zákazník jakoukoliv změnu v API (přidání či úpravu) je možné žádanou změnu implementovat a po otestování rovnou nasadit na instanci systému u zákazníka. Zákazník v takovém případě nemusí čekat na další novou verzi systému. Požadavky tak mohou být splněny mnohem dříve než u změn v klasickém API.

Jak již bylo zmíněno, kromě rychlého splnění požadavků nevyžaduje dynamicky definované API nové nasazení celého systému. V případě klasických API, vyžaduje jakákoliv změna nové sestavení celého projektu s API, následné testování a nahraní celé nové verze. Dle využívaného stylu verzování může vydání nové verze informačního systému trvat i měsíce. Využití dynamického API tuto nutnost nového nasazení systému nevyžaduje.

Dynamicky definované webové API je tvořeno jako návrh architektury, jednotlivé informační systémy si mohou naimplementovat různé vlastní verze dynamického API. Architektura vyžaduje splnění několika obecných požadavků a při splnění je možné ji implementovat do jakéhokoliv existujícího informačního systému. Implementace v IS může být libovolná a vývojáři systému si mohou dle svého uvážení zvolit způsob napojení k již existujícího systému. S tímto bodem souvisí u možno různorodého rozšíření dynamického API. Implementace mohou architekturu libovolně rozšiřovat jakýmikoliv směry. Jednotlivé rozšíření mohou přidávat podporu nových technologií, připojovat další systémy apod.

6.2 Nevýhody

Obdobně jako každá architektura i technologie má i dynamicky definované webové API své nevýhody. Mezi jednu z hlavních nevýhod lze zařadit nerealizovatelnost vysoce komplexních koncových bodů. U klasického webového API lze vytvořit koncové bodu libovolné složitosti s libovolným počtem parametrů s libovolnými komplexními datovými typy. Dynamicky definované API umožňuje pouze takovou komplexitu, jakou dovolují abstraktní definice. Vhodné je také upozornit, že dynamické API není vhodné pro složité koncové body, které ve svém obsahu vyžadují složitou činnost, např. dlouhé validace, vícenásobné komunikace s dalšími systémy apod.

Za další nevýhodu lze považovat ztrátu přehledu z pohledu core vývojářů neboli vývojářů zodpovědných za běh jádra informačního systému. Do jádra IS lze totiž v některých případech zařadit i webové API. Obvykle tedy vývoj standartního webového API spadá právě do kompetence core vývojářů, kteří mají přehled o hlavní funkčnosti systému. V případě dynamického API ztrácí core vývojáři přehled o všech definovaných koncových bodech a nemohou zaručit bezproblémový běh API.

S přechozí nevýhodou souvisí i určitá ztráta kontroly nad webovým API. Ke ztrátě kontroly může docházet v případě připuštění externích vývojářů. Veškerý vývoj by měl jít v nejlepším případě alespoň přes kontrolu interních vývojářů. Riziko nastává i v případě, kdy dynamické endpointy začnou definovat nekompetentní osoby. Je důležité, aby o definovaných koncových bodech mělo povědomí co největší množství kompetentních osob, v opačném případě hrozí kompletní ztráta kontroly nad dynamicky definovaném webovém API.

Při vývoji jakéhokoliv softwaru je nezbytné testování a ověřování správné funkcionality. Testování jednotlivých dynamicky definovaných koncových bodů může být komplikované, neb neexistuje jednotný přístup k testování API. V ideálním případě je nutné tvořit automatizované testy, které však nejsou využitelné pro všechny případy. Z toho důvodu lze komplikované testování zařadit mezi nevýhody.

6.3 Výkon

V obecné rovině lze usuzovat, že výkon dynamického API bude mírně slabší než už klasického API. Ztráta výkonu se nachází převážně v samotném načítání definic koncových bodů. Po načtení definice je nezbytné zkontrolovat všechny vlastnosti, nalézt

správnou akci s parametry a správně dosadit parametry. Kontrola parametrů může také zpomalit exekuční čas vzhledem k nutnosti kontroly datových typu a správného mapovaní parametrů.

Načítání definic z uložiště lze výrazně zrychlit pomocí užití mezipaměti. Všechny, či jen často volané koncové body, lze načíst do mezipaměti (cache) aplikace a následný přístup k definicím je téměř okamžitý. Pro práci s mezipamětí lze použít libovolnou podporovanou technologii. V případě použití mezipaměti je však nutné přidání funkcionality pro synchronizaci změn. V případě změny nastavení koncového bodu je nezbytné, aby byly tyto změny propsány i do definice uložené v mezipaměti. Neaktualizované definice v mezipaměti mohou totiž vést k nečekanému chování.

Výkon dynamického API je zároveň vázán k výkonu hardwaru na serveru. Zvýšení výkonu lze dosáhnout pomocí využití lepšího hardwaru. Server s výkonnějším procesorem, větší pamětí, rychlým uložištěm a s rychlým síťovým připojením bude vykonávat příkazy mnohem rychleji. Použití výkonnějšího hardwaru však může přinést vysoké výdaje. Z toho důvodu není vhodné využívat drahých serverů pro všechny druhy systémů. Před samotným nasazením informačního systému je vhodné provést výzkum k očekávané vytíženosti a zjistit jaký je potřebný výkon.

Vzhledem k rozdílným implementacím dynamického API lze očekávat u každé implementace rozdílný výkon. Mohou existovat implementace s výkonem vyšším a některé s nižším. Další možností ke zvýšení výkonu je tak optimalizace zdrojového kódu, vyhledání slabých míst a jejich náprava.

7 Závěr

Cílem bakalářské práce bylo navržení možností k vytvoření dynamicky definovaného API. Cíle bylo dosaženo pomocí nastavení technologických a funkcionálních požadavků na informační systémy. S pomocí požadavků je vytvořena nadstavba nad architekturu REST, která umožňuje vytváření nových koncových bodů webových služeb za běhu systému bez nutnosti nasazování nové verze.

V rámci teoretické části je nejprve popsán protokol HTTP, který je využíván pro komunikaci s webovými službami. Následující kapitola se věnuje webovému API, kde jsou nejdříve charakterizovány formáty dat pro komunikaci a následně nejčastější přístupy k webovému API, konkrétně protokol SOAP, architektura REST a technologie GraphQL.

Po vzoru architektury REST je v praktické části definováno dynamicky definované API za využití omezení, přesněji požadavků na informační systém. V této kapitole je popsána celá nadstavba nad základní architekturou včetně nejdůležitějších funkcionalit a pro rozšíření jsou navrhnuty řešení pro případné plugin systémy či přidání podpory SOAP operací. Základní implementace dynamicky definovaného API je vysvětlena v páté kapitole, kde je vyobrazena architektura vytvářeného ukázkového informačního systému včetně struktury databáze, vyobrazení ovládání i ukázání zdrojových kódů nejdůležitějších služeb a middlewarů. V této kapitole je i popsána již existující a v praxi používaná implementace dynamicky definovaného API.

Ke zhodnocení výsledků slouží šestá kapitola s porovnáním. Vyjmenovány jsou zde výhody použití dynamického API, nicméně je nutné brát v úvahu i některé nevýhody, které jsou v této kapitole také kriticky zhodnoceny. Na závěr je v teoretické rovině i porovnán výkon klasických webových služeb s dynamicky definovanými službami.

Na základě zhodnocených nevýhod je možné pokračování výzkumu za účelem nalezení jejich řešení. Případné pokračování by také mohlo sloužit k rozšíření funkcionality, neb architektura je navržena velice otevřeně a je možné její libovolné rozšiřování.

8 Seznam použité literatury

[1] R. Fielding, Ed.- Greenbytes. RFC-9110 - HTTP Semantics. *IETF Datatracker.* [Online] IETF, 06. 06. 2022 [Citace: 17. 10. 2022] Dostupné z: https://datatracker.ietf.org/doc/html/rfc9110. RFC-9110.

[2] MDN constributors. HTTP. *MDN Web Docs.* [Online] Mozilla Corporation, 13. 5. 2022 [Citace: 5. 7. 2022] Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP.

[3] IBM Corporation. The components of a URL. *CICS Transaction Server for z/OS.* [Online] IBM, 27. 6. 2019 [Citace: 5. 7. 2022] Dostupné z: https://www.ibm.com/docs/en/cics-ts/5.1?topic=concepts-components-url.

[4] T. Berners-Lee - M. McCahill. RFC-1738 - Uniform Resource Locators (URL). *IETF Datatracker.* [Online] IETF, 1. 12. 1994 [Citace: 5. 7. 2022] Dostupné z: https://datatracker.ietf.org/doc/html/rfc1738. RFC-1738.

[5] MDN contributors. Evolution of HTTP. *MDM Web Docs.* [Online] Mozilla Corporation, 13. 5. 2022 [Citace: 5. 7. 2022] Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP.

[6] T. Berners-Lee - H. Frystyk. RFC-1945 - Hypertext Transfer Protocol -- HTTP/1.0. *IETF Datatracker.* [Online] IETF, 1. 5. 1996 [Citace: 5. 7. 2022] Dostupné z: https://datatracker.ietf.org/doc/html/rfc1945. RFC-1945.

[7] R. Fielding - T. Berners-Lee. RFC-2616 - Hypertext Transfer Protocol -- HTTP/1.1. *IETF Datatracker.* [Online] IETF, 1. 6. 1999 [Citace: 4. 7. 2022] Dostupné z: https://datatracker.ietf.org/doc/html/rfc2616. RFC-2616.

[8] Grigorik, Ilya. Brief History of HTTP. *High Performance Browser Networking.* [Online] O'Reilly Media, Inc, 2013 [Citace: 5. 7 2022] Dostupné z: https://hpbn.co/brief-history-of-http/.

[9] P. Meenan - B. Pollard. Report: State of the Web. *HTTP Archive.* [Online] http archive, 2022 [Citace: 5. 7. 2022] Dostupné z: https://httparchive.org/reports/state-of-the-web.

[10] M. Belshe - M. Thomson, Ed. RFC-7540 - Hypertext Transfer Protocol Version 2 (HTTP/2). *IETF Datatracker.* [Online] IETF, 14. 5. 2015 [Citace: 5. 7. 2022] Dostupné z: https://datatracker.ietf.org/doc/html/rfc7540. RFC-7540.

[11] NOTERMANS, THIERRY. How HTTP/2 can boost web performance. *Kadiska.* [Online] Kadiska, 17. 3. 2021 [Citace: 5. 7. 2022] Dostupné z: https://www.kadiska.com/blog-how-does-http-2-differ-from-http-1-1-to-boost-web-performance/.

[12] M. Bishop, Ed. - Akamai. RFC-9114 - HTTP/3. *IETF Datatracker.* [Online] IETF, 9. 6. 2022 [Citace: 5. 7. 2022] Dostupné z: https://datatracker.ietf.org/doc/html/rfc9114. RFC-9114.

[13] Cimpanu, Catalin. HTTP-over-QUIC to be renamed HTTP/3. *ZDNet.* [Online] ZDNET, A RED VENTURES COMPANY, 12. 11. 2018 [Citace: 6. 7. 2022] Dostupné z: https://www.zdnet.com/article/http-over-quic-to-be-renamed-http3/.

[14] Sharwood, Simon. IETF publishes HTTP/3 RFC to take the web from TCP to UDP. *The Register.* [Online] The Register | Situation Publishing Limited, 7. 6. 2022 [Citace: 6. 7. 2022] Dostupné z: https://www.theregister.com/2022/06/07/http3_rfc_9114_published/.

[15] Marx, Robin. HTTP/3 From A To Z: Core Concepts. *Smashing Magazine.* [Online] Smashing Media AG, 9. 8. 2021 [Citace: 6. 7. 2022] Dostupné z: https://www.smashingmagazine.com/2021/08/http3-core-concepts-part1/.

[16] Deveria, Alexis. Can I use HTTP/3 protocol. *Can I use.* [Online] Can I use, 1. 6. 2022 [Citace: 6. 7. 2022] Dostupné z: https://caniuse.com/http3.

[17] Walls, Colin. *Embedded software: the works.* Burlington: Newnes, 2006. ISBN 0-7506-7954-9.

[18] E. Rescorla. RFC-8446 - The Transport Layer Security (TLS) Protocol Version 1.3. *IETF Datatracker.* [Online] IETF, 10. 8. 2018 [Citace: 6. 7. 2022] Dostupné z: https://datatracker.ietf.org/doc/html/rfc8446. RFC-8446.

[19] Schechter, Emily. A secure web is here to stay. *Chromium Blog.* [Online] Google, 8. 2. 2018 [Citace: 6. 7. 2022] Dostupné z: https://blog.chromium.org/2018/02/a-secure-web-is-here-to-stay.html.

[20] MDN contributors. HTTP Messages. *MDN Web Docs.* [Online] Mozilla Corporation, 18. 2. 2022 [Citace: 7. 7. 2022] Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages.

[21] MDN contributors. HTTP headers. *MDN Web Docs.* [Online] Mozilla Corporation, 29. 3. 2022 [Citace: 7. 7. 2022] Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.

[22] Internet Assigned Numbers Authority. Message Headers. *IANA - Internet Assigned Numbers Authority.* [Online] Internet Assigned Numbers Authority, 8. 6. 2022 [Citace: 31. 3. 2023] Dostupné z: https://www.iana.org/assignments/message-headers/message-headers.xhtml.

[23] P. Saint-Andre - M. Nottingham. RFC-6648 - Deprecating the "X-" Prefix and Similar Constructs in Application Protocols. *IETF Datatracker.* [Online] IETF, 26. 6. 2012 [Citace: 8. 7. 2022] Dostupné z: https://datatracker.ietf.org/doc/html/rfc6648. RFC-6648.

[24] MDN contributors. HTTP request methods. *MDN Web Docs.* [Online] Mozilla Corporation, 13. 5. 2022 [Citace: 8. 7. 2022] Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

[25] MDN contributors. Idempotent. *MDN Web Docs.* [Online] Mozilla Corporation, 22. 7. 2022 [Citace: 28. 7. 2022] Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Idempotent.

[26] MDN contributors. Safe (HTTP Methods). *MDN Web Docs.* [Online] Mozilla Corporation, 18. 2. 2022 [Citace: 28. 7. 2022] Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Safe/HTTP.

[27] Bray, Timothy William. RFC-8259 - The JavaScript Object Notation (JSON) Data Interchange Format. *IETF Datatracker.* [Online] IETF, 13. 12. 2017 [Citace: 11. 11. 2022] Dostupné z: https://datatracker.ietf.org/doc/html/rfc8259. RFC-8259.

[28] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). *W3C.* [Online] World Wide Web Consortium (W3C), 26. 11. 2008 [Citace: 11. 11. 2022] Dostupné z: https://www.w3.org/TR/xml/.

[29] W3C. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). *W3C.* [Online] World Wide Web Consortium (W3C), 27. 4. 2007 [Citace: 11. 11. 2022] Dostupné z: https://www.w3.org/TR/soap12/.

[30] W3C. Simple Object Access Protocol (SOAP) 1.1. *W3C.* [Online] World Wide Web Consortium (W3C), 8. 3. 2000 [Citace: 12. 11. 2022] Dostupné z: https://www.w3.org/TR/soap11/.

[31] W3C. XML Protocol Working Group. *W3C.* [Online] World Wide Web Consortium (W3C), 10. 7. 2009 [Citace: 12. 11. 2022] Dostupné z: https://www.w3.org/2000/xp/Group/.

[32] Snell, James, Tidwell, Doug a Kulchenko, Pavel. *Programming Web services with SOAP.* 1st edition. Beijing : O'Reilly Media, 2002. ISBN 05-960-0095-2.

[33] W3C. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. *W3C.* [Online] World Wide Web Consortium (W3C), 26. 6. 2007 [Citace: 18. 11. 2022] Dostupné z: https://www.w3.org/TR/wsdl/.

[34] Massé, Mark. *REST API Design Rulebook.* 1st edition. místo neznámé: O'Reilly Media, 2011. ISBN 9781449310509.

[35] Facebook (Meta) Inc. GraphQL. *GraphQL specifications.* [Online] Meta (dříve Facebook), 26. 10. 2021 [Citace: 2. 1. 2022] Dostupné z: https://spec.graphql.org/October2021/.

9 Přílohy

Jedinou přílohou jsou zdrojové kódy ukázkové implementace. Zdrojové kódy jsou přiloženy k elektronické verzi této bakalářské práce.

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Jiří Falta**
Studium:	I2000347
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Možnosti dynamického definování webového API**
Název bakalářské práce AJ:	Possibilities of Dynamic Web API Definition

Cíl, metody, literatura, předpoklady:

Cíl: Cílem práce je popis, prozkoumání a vypracování možností převodu standardních staticky definovaných webových API na dynamicky definovaná API.

Osnova:

1. Úvod
2. Popis HTTP
3. Problematika webového API
4. Dynamicky definované webové API
5. Návrh řešení
6. Výsledky a závěr

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Mgr. Tomáš Kozel, Ph.D.
Datum zadání závěrečné práce:	24.1.2022