
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**
**DEPARTMENT OF INFORMATION SYSTEMS**


# GENEROVÁNÍ APLIKACÍ V TYPESCRIPTU Z POPISU REST ROZHRANÍ

**TYPESCRIPT APPLICATION GENERATION FROM REST API DESCRIPTIONS**

**DIPLOMOVÁ PRÁCE**
**MASTER'S THESIS**


**AUTOR PRÁCE** — **Bc. SILVESTER LIPJANEC**
**AUTHOR**

**VEDOUCÍ PRÁCE** — **Ing. RADEK BURGET, Ph.D.**
**SUPERVISOR**



**BRNO 2020**

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

# Zadání diplomové práce


Student: **Lipjanec Silvester, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Generování aplikací v TypeScriptu z popisu REST rozhraní**
**TypeScript Application Generation from REST API Descriptions**
Kategorie: Informační systémy

Zadání:

1. Prostudujte existující jazyky a technologie pro obecný popis REST rozhraní, např. OpenAPI, WADL, případně další.
2. Seznamte se s principy tvorby klientských aplikací v aplikačních rámcích podporujících TypeScript. Zaměřte se zejména na Angular.
3. Navrhněte řešení pro automatické generování kostry aplikace zahrnující služby a definici datových struktur v jazyce TypeScript.
4. Implementujte navržené řešení pomocí vhodných technologií.
5. Demonstrujte použitelnost výsledného řešení na vhodné ukázkové aplikaci.
6. Zhodnoťte dosažené výsledky.

Literatura:

- Jansen, R. H.: Learning TypeScript 2.x: Develop and maintain captivating web applications with ease, 2nd Edition, Packt, 2018
- Ponelat, J. S.: Designing APIs with Swagger and OpenAPI, Manning, 2019

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Burget Radek, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 20. května 2020
Datum schválení: 16. října 2019

# Abstrakt

Táto práca sa zaoberá návrhom a implementáciou nástroja pre generovanie častí klientskych aplikácií v jazyku TypeScript z popisu REST rozhrania. Cieľom nástroja je automatické vygenerovanie kostry klientskej aplikácie využívajúcej framework Angular, ktorá zahrňuje služby a dátové štruktúry, umožňujúce prístup ku koncovým bodom servera. Práca popisuje rozhrania vytvorené podľa architektonického štýlu REST, ako aj technológie používané k ich popisu. Nástroj bol implementovaný v jazyku TypeScript a využíva behové prostredie Node.js. Generovanie výstupných súborov zabezpečuje šablónovací systém Mustache.js. Výsledkom je nástroj umožňujúci generovanie zdrojového kódu, ktorý je jednoducho použiteľný v aplikácii využívajúcej framework Angular, na základe poskytnutej WADL alebo OpenAPI špecifikácie rozhrania.



# Abstract

This thesis deals with the design and implementation of a tool for generating parts of client applications in TypeScript language from the description of a REST interface. The goal of the tool is an automatic generation of an application skeleton which uses the Angular framework including data structures and services enabling access to server endpoints. The thesis describes the interfaces based on the REST architectural style, as well as the technologies used for their description. The tool was implemented in TypeScript language and uses the Node.js runtime. The output file generation is based on the Mustache.js template system. The result is a tool which allows the generation of source code based on the provided WADL or OpenAPI interface description, which can be simply used as a part of an Angular application.



# Kľúčové slová

API, REST, Swagger, OpenAPI, WADL, TypeScript, HTTP, Angular, Mustache.js, Node.js, generovanie aplikácie, systém šablónovania

# Keywords

API, REST, Swagger, OpenAPI, WADL, TypeScript, HTTP, Angular, Mustache.js, Node.js application generation, template system


# Citácia

LIPJANEC, Silvester. *Generování aplikací v TypeScriptu z popisu REST rozhraní*. Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Burget, Ph.D.

# Generování aplikací v TypeScriptu z popisu REST rozhraní

## Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Radka Burgeta, Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Silvester Lipjanec
31. mája 2020

## Poďakovanie

Týmto by som chcel poďakovať môjmu vedúcemu práce, ktorým bol Ing. Radek Burget, Ph.D., za konzultácie a odbornú pomoc poskytnutú pri tejto práci.

# Obsah

# Kapitola 1

# Úvod

Hlavným cieľom tejto práce je navrhnúť a implementovať nástroj pre automatické generovanie kostry aplikácií v jazyku TypeScript a to na základe popisu REST rozhrania. Kostra aplikácie by mala zahrňovať dátové štruktúry a služby, pomocou ktorých bude možné pristupovať ku koncovým bodom servera. Štruktúra vygenerovaného zdrojového kódu má byť koncipovaná na jeho využitie v klientskej aplikácii, ktorá využíva framework *Angular*. Zámerom tvorby takéhoto nástroja je uľahčenie a urýchlenie procesu programovania klientskych aplikácií, ktoré na komunikáciu so serverom využívajú REST rozhranie.

Webové služby, ktoré komunikujú pomocou protokolu SOAP, umožňujú svoju funkcionalitu popisovať pomocou jazyka WSDL. Zároveň existujú nástroje, ktoré tento popis umožňujú jednoducho a rýchlo vygenerovať. Pre rozhrania vytvorené podľa architektonického štýlu REST dlho žiadny spôsob popisu a automatického generovania neexistoval. Až následne začali vznikať nástroje ako *Swagger*, ktoré pridali možnosti popisu REST rozhraní. Nástroj *Swagger* sa postupne štandardizoval a ďalej vyvíja ako *OpenAPI*. Postupne začali pribúdať aj nástroje, ktoré popis rozhraní umožňovali generovať a naopak také, ktoré na základe popisu rozhrania umožňovali generovať funkčný zdrojový kód. V rámci tejto práce sú popísané existujúce nástroje, ktoré umožňujú generovanie zdrojového kódu aplikácií. Práca popisuje ich funkcie a nedostatky, a na základe nich navrhuje vlastnosti a funkcie, ktoré by mal navrhovaný nástroj ponúkať.

V rámci návrhu nástroja je zahrnutá aj časť zaoberajúca sa automatickou kontrolou zmien v použitej špecifikácii rozhrania. Vo výsledku môže tento nástroj vývojárom ušetriť množstvo času pri samotnej implementácii služieb pre prístup k serveru, ako aj pri prípadnom hľadaní a opravovaní chýb. Ak zmena v rozhraní nastala, môže vzniknúť potreba zmeny implementácie služieb alebo dátových štruktúr na strane klienta.

Na začiatku práce sa kapitola 2 zaoberá webovými aplikačnými rozhraniami a webovými službami. Popisuje existujúce jazyky a technológie používané k popisu webových služieb a aplikačných rozhraní. Predovšetkým sa zameriava na popis rozhraní typu REST a bližšie popisuje jazyk WADL a štandard *OpenAPI*. Kapitola 3 sa zaoberá princípmi tvorby moderných webových aplikácií, porovnáva tradičné webové aplikácie s aplikáciami typu *single-page* a prezentuje základné technológie využívané pri ich tvorbe. Kapitola 4 na začiatku prezentuje vlastnosti existujúcich nástrojov, ďalej špecifikuje požiadavky kladené na návrh nástroja a následne samotný návrh popisuje. Kapitola 5 prezentuje použité technológie pri implementácii nástroja. Ďalej popisuje štruktúru a konfiguráciu projektu, ako aj samotné implementačné detaily. Dôležitou časťou tejto kapitoly je aj popis štruktúry vygenerovaného kódu. Kapitola 6 popisuje priebeh testovania implementovaného nástroja. Taktiež prezentuje súhrn vlastností nástroja a navrhuje možné vylepšenia.

# Kapitola 2

# Technológie pre tvorbu webových aplikačných rozhraní

Po celé desaťročia, väčšina počítačových programov a webových stránok bola koncipovaná iba pre jeden typ koncového užívateľa - pre človeka. Bez ohľadu na to aký reťazec operácií program vykonal, jeho výsledok bol určený pre ľudí. Výstup bol ľuďmi spracovaný z užívateľského rozhrania, ktoré ho prezentovalo tak jednoducho, ako to bolo možné [6].

Postupom času sa začali vytvárať programy a aplikácie, ktorých výstup mal slúžiť ako vstup pre ďalšie programy, ktoré ich ďalej mohli využiť podľa vlastnej potreby. Web je plný dát a služieb najrôznejšieho druhu: mapy, sociálne siete, vyhľadávacie služby, cestovanie, hudba, počasie, šport, elektronický obchod, financie, fotografie a ďalšie. Namiesto inštalovania nezmyselného množstva programov do vlastného počítača stačí iba jeden - webový prehliadač, ktorý zabezpečí prístup k týmto dátam a službám pomocou aplikačného rozhrania [25]. Toto rozhranie nemusí byť užívateľsky prívetivé, ale musí efektívne a jednoducho spracovať dáta a poskytnuté služby [6].

Koncept aplikačného programového rozhrania (*Application programming interface*, ďalej skrátene API) začal vznikať už v 60. rokoch minulého storočia. Programátori začali vytvárať knižnice pre procedurálne jazyky a zdieľať ich s ostatnými. Tí ich ďalej mohli využívať bez znalosti ich vnútornej implementácie.

V 70. a 80. rokoch so vznikom sieťovo prepojených počítačov začali vznikať prvé sieťové API. Vývojári mohli vystaviť funkcionalitu implementovaných služieb na sieť a zároveň využívať iné knižnice pomocou volania vzdialených procedúr (*Remote Procedure Calls*, ďalej skrátene RPCs).

V 90. rokoch so vznikom internetu chcelo mnoho spoločností štandardizovať spôsob vytvárania a vystavovania API. Štandardy ako COBRA (*Common Object Request Broker Architecture*), COM (*Component Object Model*), DCOM (*Distributed Component Object Model*) a veľa ďalších sa usilovali o to, aby sa stali konceptom pre vystavovanie služieb cez sieť. Ich problémom bolo to, že boli príliš komplexné na správu, vyžadovali využívanie rovnakého programovacieho jazyka na oboch stranách a niekedy aj lokálnu inštaláciu určitých častí vzdialených služieb.

Na prelome tisícročia začali vznikať nové štandardizované formy prístupu k vzdialeným službám cez web (web API). Protokol SOAP (*Simple Object Access Protocol*) a XML (*Extensible Markup Language*), neskôr REST (*Representative State Transfer*) a JSON (*JavaScript Object Notation*) umožňovali jednoduchý prístup k vzdialeným službám bez obmedzenia

programovacieho jazyka alebo závislostí kódu klienta. Použité informácie v tejto sekcii boli prevzaté z [13].

Na začiatku tejto kapitoly je čitateľ oboznámený s webovými službami a aplikačnými rozhraniami. Sekcia 2.3 sa zameriava na popis vlastností REST rozhraní a následne v sekcii 2.4 sú prezentované technológie používané pri implementácii REST rozhraní. Sekcia 2.5 popisuje existujúce jazyky a technológie určené pre popis REST rozhraní.

## 2.1 Webové služby

Webové služby sú systémy, ktoré vystavujú svoju funkcionalitu na internete a na komunikáciu využívajú štandardizované XML dokumenty. Týmito systémami môžu byť programy, objekty, správy, databázy alebo akékoľvek iné pokročilé biznis funkcie. Tieto programy zasielajú XML dokumenty ako správy webovým službám po sieti a prípadne aj získavajú odpovede vo forme XML dokumentov. Webové služby využívajú protokoly a štandardy, ktoré definujú formát správ a rozhraní, na ktoré sú správy zasielané. Ďalej popisujú spôsoby mapovania správ na programy a mechanizmy na získavanie informácií o rozhraniach. Webové služby k prenosu a transformácii dát z alebo do programov, vyžívajú technológie ako XML, WSDL a SOAP.

Webové služby oproti Webu ako takému nepracujú na princípe vystavovania zdrojov. Typická webová služba je vystavená na jednu URI adresu a nedefinuje žiadne prepojenia na iné služby. Protokol HTTP využívajú iba ako prostriedok na zabezpečenie prenosu správ a nevyužívajú jeho ďalšiu sémantiku. Informácie v tejto sekcii boli prevzaté z [20] a [25].

### XML

XML (*Extensible Markup Language*) je značkovací jazyk, ktorý umožňuje popisovať štruktúrované dáta. Pomocou XML je možné definovať elementy popisujúce dáta a následne ich zoskupovať do štruktúr podľa ich významu. V rámci webových služieb je pomocou XML definovaný formát správ, štruktúra dát posielaných v rámci týchto správ, ale aj spôsob akým sú samotné služby definované. Keďže štandard XML umožňuje definovanie vlastných elementov, musí existovať spôsob ako zabezpečiť, že komunikujúce strany interpretujú všetky elementy rovnakým spôsobom. Z tohto dôvodu je potrebná definícia schémy, ktorá určuje sémantiku dát. Následne je potrebné zabezpečiť jej použitie oboma komunikujúcimi stranami.

### WSDL

WSDL (*Web Services Description Language*) je jazyk založený na XML, ktorý popisuje z čoho webová služba pozostáva. Poskytuje mechanizmus, pomocou ktorého je možné popisovať rozhrania webových služieb. WSDL umožňuje jednoznačným spôsobom reprezentovať dátové typy využívané v rámci správ, taktiež operácie, ktoré majú byť vykonané na základe správy a popisovať mapovanie správ na prenos po sieti. Každá z týchto informácii je definovaná pomocou jedného majoritného elementu, ktorý môže byť definovaný ako separátny XML dokument. Tie môžu byť následne importované a spojené do výsledného popisu webovej služby. Pre definíciu typov sa typicky využívajú XML schémy. Operácie sa typicky mapujú na metódy alebo programy, ktoré webové služby implementujú. Samotné mapovanie popisuje protokoly a transportné prostriedky využívané na prenos dát k službám.

Ručné vytváranie WSDL popisu webových služieb je možné, ale zdĺhavé a náročné. Z tohto dôvodu existujú nástroje, ktoré umožňujú popis webových služieb generovať automaticky. Takmer všetky WSDL sú v dnešnej dobe generované nástrojmi a ďalej sú používané inými nástrojmi. Klientská aplikácia potom môže na základe WSDL popisu volať webové služby, ako keby boli jej súčasťou.

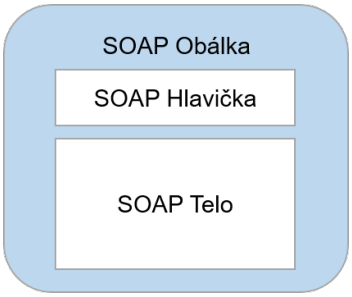
Rôzne nástroje môžu WSDL generovať a interpretovať rôznym spôsobom. Preto musí byť zaručené, že klient a server budú využívať rovnakú sadu nástrojov, čím sa medzi nimi vytvára úzka väzba. Webové služby však boli zamýšľané tak, aby mohli byť vystavené na sieti a klient ich mohol použiť bez ohľadu na používaný softvér.

### SOAP

SOAP (*Simple Object Access Protocol*) je protokol, ktorý definuje mechanizmus komunikácie medzi webovými službami. Protokol vytvára komunikačný model, pomocou ktorého zabezpečuje, že správy v podobe XML dokumentov budú prevedené od odosielateľa k prijímateľovi. Protokol pomocou XML určuje formát správ a spôsob, akým budú správy posielané po sieti a ako budú dostupné na spracovanie cieľovým počítačom. Na prenos správ po sieti najčastejšie využíva aplikačný protokol HTTP, ale využitie iného transportného protokolu nie je vylúčené. Protokol taktiež určuje pravidlá pre mapovanie na využitý transportný protokol.

SOAP správa pozostáva z troch základných častí, ktoré ilustruje obrázok 2.1:

- **Obálka** - Povinná časť správy, ktorá určuje jej začiatok a koniec.
- **Hlavička** - Nepovinná časť, ktorá popisuje vlastnosti používané pri spracovaní správ. Nesie informácie doplňujúce samotnú správu, napríklad bezpečnostné tokeny, informácie o smerovaní alebo identifikátory transakcií.
- **Telo** - Povinná časť obsahujúca XML dáta, ktoré zahrňujú odosielanú správu.




Obr. 2.1: Znázornenie základných častí SOAP správy.

## 2.2 Webové aplikačné rozhranie

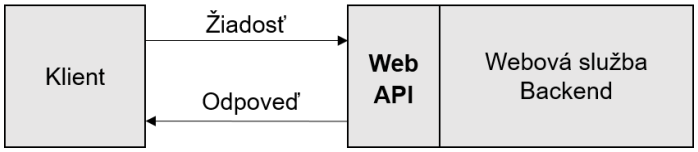
Aplikačné programové rozhranie (*Application programming interface*, ďalej skrátene API) je rozhranie, ktoré poskytuje program iným programom alebo ľuďom a v prípade web API celému svetu cez internet. Aj keď je API navrhnuté na spoluprácu s iným softvérom je navrhnuté tak, aby bolo pochopiteľné pre človeka - vývojára, ktorý API používa.

API vznikli z potreby výmeny informácií medzi poskytovateľmi dát, ktorí sú schopní vyriešiť určitý problém a ďalšími, ktorí nemusia tráviť čas na vyriešení rovnakého problému

samostatne. Sú napríklad schopní vložiť interaktívnu mapu na webovú stránku bez toho, aby si sami museli vytvoriť Google Mapy alebo využiť prihlasovanie užívateľov za pomoci Facebook login, bez potreby implementovať vlastné prihlasovanie. Za použitia API z takýchto špecializovaných platforiem majú ostatní možnosť vytvoriť nové doplňujúce funkcie alebo produkty jednoducho a rýchlo. Použité informácie v tejto sekcii boli prevzaté z [13].

### Rozhrania typu žiadosť - odpoveď

Rozhrania typu žiadosť - odpoveď vystavujú dáta a služby skrz HTTP server. Web API definuje koncové body (*endpoints*), ku ktorým má klient prístup. Klienti zasielajú HTTP žiadosti (*HTTP requests*) na tieto koncové body a server im odpovedá HTTP odpoveďou (*HTTP response*). Ako vidieť na obrázku 2.2, Web API stojí v popredí webovej služby vystavenej serverom a priamo naslúcha a odpovedá na žiadosti klienta [15]. Formát odpovede môže byť rôzny, najčastejšie sa však jedná o JSON alebo XML dáta.




Obr. 2.2: Znázornenie postavenia Web API mezi klientom a serverom. Web API je tvárou webových služieb backendu - odpovedá na žiadosti klienta. Preložené z [15].

Sú tri hlavné API formy využívajúce tento princíp vystavovania služieb iným aplikáciám a to REST, RPC a GraphQL.

## 2.3 Aplikačné programové rozhrania typu REST

REST je akronym pre *Representational State Transfer*. Je to architektonický štýl, ktorý bol prezentovaný v roku 2000 Royom Fieldingom v jeho dizertačnej práci[1]. Fielding vo svojej dizertačnej práci stanovuje vlastnosti, ktoré môže mať webový systém a poukazuje na vlastnosti, ktoré robia web úspešným. Ďalej vyberá obmedzenia, ktoré z generického sieťového systému urobia systém podobný webu. Tieto obmedzenie sú formálnou architektonickou definíciou, ktorá zachytáva podstatu webu [27]. Fielding tieto obmedzenia zoskupil do 6 kategórií: [15]

- **Klient-Server** (*Client-Server*) - Web je systém založený na modeli klient-server. Klient aj server sú dve nezávislé časti, ktoré môžu byť implementované a nasadené samostatne. Pri tom môžu používať rôzne jazyky a technológie, po dobu kým využívajú jednotné rozhranie.
- **Jednotné rozhranie** (*Uniform Interface*) - Všetky interakcie medzi komponentami webu (klientmi, servermi a sieťovými sprostredkovateľmi) závisia na jednotnosti ich rozhrania. Fielding vo svojej práci identifikuje štyri obmedzenia kladené na jednotné rozhranie:

[1] https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

1. **Identifikácia zdrojov** - Taktiež možno nazvať ako adresovateľnosť. Každý zdroj (*resource*) môže byť adresovaný jednoznačným identifikátorom URI (*Uniform Resource Identifier*[2]). Stav zdroja sa môže meniť, ale URI ostáva rovnaké.
2. **Manipulácia so zdrojmi skrz ich reprezentácie** - Server reprezentuje stav zdroja klientovi zaslaním jeho reprezentácie. Klient môže zmeniť stav zdroja zaslaním reprezentácie na server.
3. **Seba-popisujúce správy** (*Self-descriptive messages*) - Okrem samotnej správy môžu obsahovať metadáta k sprostredkovaniu dodatočných informácií o stave zdroja, formátu zdroja alebo jeho veľkosti. Požadovaný stav zdroja môže byť špecifikovaný ako súčasť žiadosti od klienta. Aktuálny stav zdroja môže byť reprezentovaný v rámci správy odpovede od servera.
4. **Hypermédia** - Stav zdroja obsahuje odkaz na iný súvisiaci zdroj. Odkazy nasmerujú užívateľa skrz web zmysluplným a riadeným spôsobom a tým umožňujú zmenu stavu aplikácie. Toto obmedzenie sa v určitých zdrojoch definuje ako HATEOAS (*Hypermedia as the engine of application state*).

- **Vrstevný systém** (*Layered System*) - Vrstevný systém umožňuje neviditeľne vkladať medzi klienta a server dodatočné sieťové vrstvy ako napríklad brány (*gateways*) alebo proxy, za účelom zvýšenia bezpečnosti, ukladania odpovedí do vyrovnávacej pamäte cache alebo vyváženia výkonu (*load balancing*).

- **Vyrovnávacia pamäť** (*Cache*) - Zabezpečuje možnosť ukladania odpovede do vyrovnávacej pamäte (*cache*). Tým umožňuje znížiť dobu čakania na odpoveď, zvýšiť dostupnosť a spoľahlivosť aplikácie a kontrolovať zaťaženie webového servera.

- **Bezstavovosť** (*Statelessness*) - Webový server si neuchováva stav klientskej aplikácie. Stav aplikácie si pamätá iba klient, ktorý musí pri každej interakcii so serverom zahrnúť v rámci dát relevantné informácie o kontexte aplikácie. Ak klient v danom momente od servera nič nežiada, potom server o existencii klienta nevie.

- **Kód na požiadanie** (*Code-on-demand*) - Server má možnosť odoslať spustiteľný kód klientovi, ktorý musí byť schopný získanému kódu rozumieť a vykonať ho. Toto obmedzenie je považované za nepovinné. Príkladom môžu byť technológie webového prehliadača ako Java applety[3], JavaScript alebo Flash[4].

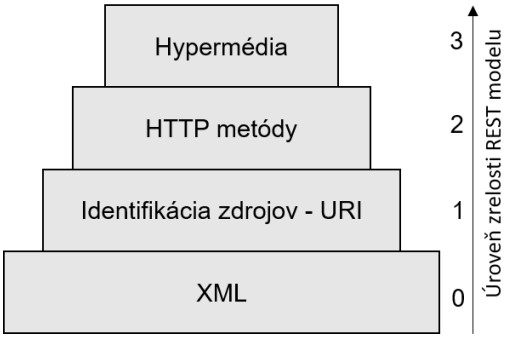
## 2.4 Úrovne zrelosti REST modelu

Tento model je užitočný pretože technológie, na ktoré poukazuje, sú reálnym príkladom implementácie REST obmedzení. Koncept zrelosti REST modelu je inšpirovaný podľa [26] a znázorený na obrázku 2.3.

### Úroveň 0

Na tejto úrovni sa využívajú volania vzdialených procedúr (RPCs) alebo typicky webové služby SOAP, ktoré boli spomínané v úvode tejto kapitoly. Všetky vystavené funkcie sú

[2] https://tools.ietf.org/html/rfc3986
[3] https://www.javatpoint.com/java-applet
[4] https://www.adobe.com/sk/products/flashplayer.html




Obr. 2.3: Úrovne zrelosti REST modelu vyjadrujúce, ako je implementovaná služba súhlasná s REST modelom, ktorý bol popisovaný Fieldingom. Úroveň je priradená na základe technológií použitých pri jej implementácii. Inšpirované podľa [12].

dostupné z jedného URI za použitia HTTP POST metódy. V tele správy sa posielajú XML fragmenty obsahujúce príkazy pre vykonanie procedúr.

### Úroveň 1

Toto riešenie aplikuje softvérový princíp oddelenia zodpovednosti (*Separation of concerns*[5]) a tým znižuje vysokú komplexnosť jedného zdroja. Každý zdroj je adresovaný jedinečným identifikátorom URI, avšak rovnako ako na úrovni 0, využíva sa iba jedna HTTP metóda.

#### Zdroje a ich identifikácia pomocou URI

Zdrojom môže byť čokoľvek na čo by sa klient mohol chcieť odkazovať - umelecké dielo, nejaká informácia, operácia nad dátami, abstraktný pojem alebo fyzický objekt, ktorý nemôže byť poslaný cez internet. Zdrojom môže byť aj kolekcia iných zdrojov. Klienti preto nemôžu pracovať so zdrojmi priamo. Namiesto toho pracujú s reprezentáciou zdrojov vo forme dokumentov v špecifickom formáte, ktorá obsahuje informácie o zdroji [25]. Jeden zdroj môže mať rôzne reprezentácie pre rôznych klientov, bez nutnosti zmeny jeho identifikátora.

Formát URI by sa mal držať určitých zásad. Základom je určenie archetypov zdrojov, ktoré pomáhajú vyjadriť ich typickú štruktúru a chovanie. REST API je vytvorené zo štyroch základných archetypov zdrojov: [15]

- **Dokument** - Je inštancia objektu alebo databázový záznam. V URI je vyjadrený podstatným menom v jednotnom čísle.
- **Kolekcia** - Adresár zdrojov riadený serverom. Pre názov kolekcie sa používa podstatné meno v množnom čísle.
- **Sklad** - Úložisko zdrojov riadené klientom. Pre názov skladu sa taktiež používa podstatné meno v množnom čísle.
- **Kontrolér** - Modeluje koncept spustiteľných procedúr. Slovesná fráza alebo sloveso sa používa ako názov v URI.

[5] https://medium.com/machine-words/separation-of-concerns-1d735b703a60

Jednotlivé časti URI sú oddelené lomkou (/) a ich názvy sú volené zmysluplne. Na prvý pohľad má byť jasne zrozumiteľné, ako je navrhnutá hierarchická štruktúra zdrojov [15]. V tabuľke 2.1 sú uvedené príklady URI a ich príslušné archetypy.

| URI | Archetyp zdroja |
|---|---|
| `http://api.example.org/leagues/seattle` | Dokument |
| `http://api.example.org/league` | Kolekcia |
| `http://api.example.org/users/1234/favorites/alonso` | Sklad |
| `http://api.example.org/students/morgan/register` | Kontrolér |

Tabuľka 2.1: Príklady identifikátorov URI pre zdroje a uvedenie príslušného archetypu, ktorý reprezentujú. Inšpirované podľa [15].

### Úroveň 2

Nadväzuje na úroveň 1 a pre každý jedinečne identifikovaný zdroj poskytuje zmysluplné štandardizované operácie (HTTP metódy). Výhodou tohto riešenia je, že namiesto jednej HTTP POST metódy, ktorej význam bol „vykonať čokoľvek“, sa používajú špecifické operácie, ktoré majú vlastný význam.

#### HTTP metódy a operácie CRUD

API klienti môžu komunikovať s API zasielaním rôznych HTTP správ. Použitá metóda informuje server o tom, aká operácia sa má nad zdrojom vykonať. Rôzne HTTP metódy vyvolané nad rovnakým URI poskytujú rôznu funkcionalitu. Štandard HTTP[6] definuje osem rôznych typov správ: GET, POST, DELETE, PUT, HEAD, OPTIONS, CONNECT a TRACE. Metóda PATCH, ktorá je taktiež použiteľná pre REST API je definovaná mimo tento štandard[7]. Tabuľka 2.2 ukazuje príklad využitia rôznych typov HTTP metód nad kolekciou a entitou a popisuje operácie, ktoré budú za použitia danej metódy vykonané.

HTTP metódy GET, POST, PUT, PATCH a DELETE reprezentujú takzvané CRUD operácie, teda operácie vytvorenia (*Create*), čítania (*Read*), aktualizácie (*Update*) a mazania (*Delete*). Tabuľka 2.3 ukazuje typické použitie rôznych typov HTTP metód v REST API a popisuje príslušné CRUD operácie, ktoré budú za použitia danej metódy vykonané. Pre jednotlivé CRUD operácie sa typicky používajú tieto metódy: [13]

- **Create** - Metódu POST klient využíva na vytvorenie nového zdroja, ak nepozná jeho URI. Stačí, ak pozná URI nadradenej kolekcie a URI nového zdroja získa ako odpoveď v hlavičke `Location`. Ak klient pozná URI nového zdroja, potom môže byť na jeho vytvorenie použitá metóda PUT.
- **Read** - Pre získanie reprezentácie zdroja sa používa idempotentná metóda GET. Jej sémantika je iba pre čítanie a nemá žiadne vedľajšie účinky. To znamená, že stav zdroja sa po žiadosti GET nikdy nezmení.

[6] https://tools.ietf.org/html/rfc2616
[7] https://tools.ietf.org/html/rfc5789

| HTTP metóda | Kolekcia | Entita |
|---|---|---|
| GET | Získanie zoznamu položiek. | Čítanie entity. |
| POST | Pridanie prvku do kolekcie. | Pridanie anotáciu k zdroju. |
| PUT | Pridanie prvku ku kolekcii. | Zápis konkrétnej entity. |
| DELETE | Odstránenie celej kolekcie. | Odstránenie entity. |
| HEAD | Získanie metadát. | Získanie metadát. |
| OPTION | Zistenie použiteľných HTTP metód. | Zistenie použiteľných HTTP metód. |

Tabuľka 2.2: Využitie rôznych typov HTTP metód nad kolekciou a entitou a popis operácií, ktoré budú za použitia danej metódy vykonané. Inšpirované podľa [25].

- **Update** - Pre nahradenie celého zdroja sa používa metóda PUT a metódu PATCH je možné použiť pre aktualizáciu časti existujúceho zdroja.
- **Delete** - Pre odstránenie existujúceho zdroja sa používa metóda DELETE.

| Operácia | HTTP metóda | URL: /users | URL: /users/U123 |
|---|---|---|---|
| Create | POST | Vytvorenie nového užívateľa. | Pridanie anotácie. |
| Read | GET | Zoznam všetkých užívateľov. | Získanie užívateľa U123. |
| Update | PUT/PATCH | Skupinová aktualizácia užívateľov. | Aktualizácia užívateľa U123. |
| Delete | DELETE | Odstránenie všetkých užívateľov. | Odstránenie užívateľa U123. |

Tabuľka 2.3: Typické použitie rôznych typov HTTP metód v REST API a popis príslušných CRUD operácií, ktoré budú za použitia danej metódy vykonané. Prevzaté z [13].

Mimo typické CRUD operácie je niekedy potrebné reprezentovať operácie, ktoré nie sú CRUD. V tomto prípade sú typicky využité nasledujúce prístupy [13]:

- Vyjadriť zamýšľanú operáciu v časti zdroja. Príkladom môže byť GitHub API[8], ktoré používa parameter `archived` ako vstupný parameter pri operácií úpravy repozitára, k vykonaniu archivácie repozitára.
- Zaobchádzať s danou operáciou ako s podzdrojom. V tomto prípade bude podzdroj archeotypu kontrolér.
- Pre operácie vyhľadávania použiť v časti URL sloveso, označujúce akciu nasledovanú dotazom (napr. `GET /search/code?q={query}`).

[8] https://developer.github.com/v3/repos/

### Úroveň 3

V rámci reprezentácie zdrojov sa pridávajú hypermédia obsahujúce hyperlinky - odkazy reprezentujúce vzťahy medzi prepojenými zdrojmi. Pri prístupe na koreňovú adresu API server vráti odpoveď, ktorá obsahuje odkazy na dostupné zdroje. Nasledovaním odkazov obsiahnutých v odpovediach od servera je možné riadene prechádzať štruktúrou webu, bez predchádzajúcej znalosti koncových bodov. Hlavným dôvodom využitia hypermédií je oddelenie poskytovateľa API od klienta, ktorý ho používa. Klient a server potom nemajú žiadnu spoločnú špecifikáciu rozhrania, ktorá by popisovala koncové body servera. Na rozdiel od toho, server musí popísať dostupné koncové body aplikácie kompletne pomocou vzťahov a ich odkazov. Klient potom musí vyhodnocovať odkazy a porozumieť vzťahom, ktoré reprezentujú [12].

Fielding poukazuje na to, že iba rozhranie spĺňujúce všetky obmedzenia popísané v sekcii 2.3 môže byť nazývané ako RESTful. Vo svojom blogu [10] kritizuje, že nestačí aby bolo rozhranie založené na HTTP a hovorí, že REST API musí byť riadené hypermédiami. Napriek tomu, že Fielding poukazuje na nesprávnosť používania termínu REST API, aj ja sa pri tvorbe tejto práce obmedzím na druhú úroveň REST modelu a nebudem brať do úvahy použitie hypermédií.

## 2.5 Existujúce jazyky a technológie pre popis REST rozhraní

Jazyky pre popis rozhraní sú formálne jazyky slúžiace k vytvoreniu štrukturovaného popisu (špecifikácie) rozhrania. Typicky je špecifikácia vytvorená ako dokument popisujúci služby API (*service description document*). Tento dokument obsahuje popis rozhrania a sémantiky aplikácie v zrozumiteľnej podobe pre programátora klientskej časti softvéru, ale aj pre nástroje, ktorých úlohou je automatické generovanie rôznych častí aplikácie. Príkladom môže byť generovanie knižníc pre prístup k rozhraniu pomocou rôznych programovacích jazykov.

### Jazyk WADL

WADL (*Web Application Description Language*) je jazyk založený na XML. Jeho názov je analógiou k jazyku WSDL (*Web Service Description Language*) využívaného k popisu SOAP a RPC služieb. WADL je navrhnutý tak, aby poskytoval strojovo spracovateľný popis webových aplikácií založených na HTTP [11].

Pomocou WADL možno popisovať nasledujúce aspekty: [11]

- **Množinu zdrojov** - Analógia s mapou stránok.
- **Vzťahy medzi zdrojmi** - Popis odkazov medzi zdrojmi.
- **Metódy aplikovateľné pre konkrétne zdroje** - Aplikovateľné HTTP metódy, očakávané vstupy a výstupy a ich podporované formáty.
- **Formáty reprezentácií zdrojov** - Podporované MIME typy[9] a použité dátové schémy[10].

[9] https://www.iana.org/assignments/media-types/media-types.xhtml
[10] https://www.w3.org/standards/xml/schema

V nasledujúcej časti práce bude vytvorený popis fiktívneho rozhrania s názvom *del.icio.us*, ktorý bol prevzatý z [25]. Na jednotlivých príkladoch bude rozoberané, ako je možné vytvoriť popis rozhrania pomocou jazyka WADL a zároveň bude popísaný význam jednotlivých častí.

WADL súbor popisujúci web API pozostáva z troch hlavných častí: definície zdrojov, definície metód a definície reprezentácií zdrojov. Príklad 2.1 ukazuje definíciu zdroja, ktorú rozhranie vystaví na adrese `https://api.del.icio.us/v1/posts/recent`. Pre tento zdroj je odkazom definovaná metóda `getRecentPosts`, ktorej definícia je uvedená v príklade 2.2. V rámci aktuálne popisovaného rozhrania musí byť každá žiadosť odoslaná klientom autorizovaná pomocou HTTP Basic autentifikácie[11]. Z tohto dôvodu je v definícii zdroja uvedený parameter `Authorization`, ktorý slúži pre informovanie klienta o potrebe vloženia parametera Authorization do HTTP hlavičky.

```
1  <?xml version="1.0"?>
2  <!-- This is a partial bootleg WADL file for the del.icio.us API. -->
3  <application xmlns="http://research.sun.com/wadl/2006/07">
4      <!-- The resource -->
5      <resources base="https://api.del.icio.us/">
6          <doc xml:lang="en" title="The del.icio.us API v1">
7              Post or retrieve your bookmarks from the social
8              networking website. Limit requests to one per second.
9          </doc>
10         <resource path="v1">
11             <param name="Authorization" style="header" required="true">
12                 <doc xml:lang="en">All del.icio.us API calls
13                 must be authenticated using Basic HTTP auth.</doc>
14             </param>
15             <resource path="posts">
16                 <resource path="recent">
17                     <method href="#getRecentPosts" />
18                 </resource>
19             </resource>
20         </resource>
21     </resources>
```

Výpis 2.1: Ukážka definície zdroja v jazyku WADL pre fiktívne rozhranie *del.icio.us*. V rámci elementu resource je uvedená definícia zdroja vrátane parametrov, metód a ďalších podzdrojov. Prevzaté z [25].

Element `method` v príklade 2.2 popisuje žiadosť (`request`) a odpoveď (`response`) HTTP metódy. Jeho atribút `name` vyjadruje použitú HTTP metódu a atribút `id` je ľubovoľne zvolený identifikátor metódy. Element `request` definuje ďalšie dva parametre, ktoré budú použité ako reťazec pri dotazovaní (tzv. *query string*[12], napr. `?tag=rest&count=20`). Element `response` popisuje reprezentáciu odpovede na túto metódu, ktorej definícia je uvedená v príklade 2.3. Element `fault` definuje chybový stav, do ktorého sa klient dostane, ak autorizácia neprebehne úspešne.

Časť WADL súboru v príklade 2.3 špecifikuje reprezentáciu zdroja. Atribút `mediaType` definuje, že MIME typ dokumentu odpovede je `text/xml`. Atribút `element` definuje, že

[11] https://tools.ietf.org/html/rfc7617
[12] https://en.wikipedia.org/wiki/Query_string

```
    <!-- The method -->
    <method id="getRecentPosts" name="GET">
        <doc xml:lang="en" title="Returns a list of the most recent posts." />
        <request>
            <param name="tag" style="form">
                <doc xml:lang="en" title="Filter by this tag." />
            </param>
            <param name="count" style="form" default="15">
                <doc xml:lang="en" title="Number of items to retrieve.">
                Maximum: 100
                </doc>
            </param>
        </request>
        <response>
            <representation href="#postList" />
            <fault id="AuthorizationRequired" status="401" />
        </response>
    </method>
```

Výpis 2.2: Časť súboru popisujúceho fiktívne rozhranie *del.icio.us* v jazyku WADL, ktorá popisuje použité metódy. Špecifikuje typ HTTP metódy, formát vstupu, výstupu a chybový stav pri nesprávnej autentifikácii. Prevzaté z [25].

obsah odpovede bude vnorený v rámci elementu `posts`. Element `param` vyjadruje, že v odpovedi sa v rámci elementu `posts` môžu nachádzať synovské elementy s názvom `post`. Atribút `path` je XPath výraz[13], ktorý môže klient využiť k získaniu všetkých príspevkov (elementov `post`) z výstupného XML súboru.

```
    <!-- The representation -->
    <representation id="postList" mediaType="text/xml" element="posts">
        <param name="post" path="/posts/post" repeating="true" />
    </representation>
</application>
```

Výpis 2.3: Časť súboru popisujúceho fiktívne rozhranie *del.icio.us* v jazyku WADL, ktorá popisuje reprezentáciu zdrojov. Špecifikuje formát odpovede a ponúka XPath výraz na prehľadávanie vnútorných elementov odpovede. Prevzaté z [25].

### Štandard OpenAPI

OpenAPI je štandard určený pre popis rozhraní založených na protokole HTTP - typicky REST rozhraní. Štandard OpenAPI umožňuje popísať celé rozhranie v jazyku YAML[14] alebo JSON[15] vrátane: [33]

- Dostupných koncových bodov a príslušných operácií.
- Vstupných a výstupných parametrov každej operácie.

[13] https://www.w3.org/TR/1999/REC-xpath-19991116/
[14] https://yaml.org/spec/1.2/spec.html
[15] https://www.json.org/json-en.html

- Autentifikačných metód.
- Kontaktných informácií, licencií, pravidiel používania a iných.

OpenAPI dokument umožňuje ľuďom aj počítačom pochopiť schopnosti služieb bez nutnosti prístupu k ich zdrojovému kódu, dokumentácii alebo potreby skúmania sieťovej prevádzky. Ak je popis správne definovaný, užívateľ dokáže pochopiť a komunikovať so vzdialenou službou bez toho, aby poznal jej implementáciu [17].

Popis môže byť vytvorený ručne, využitím nástrojov alebo vygenerovaný priamo z kódu. Po vytvorení OpenAPI dokumentu sa API stáva formálne popísaným. Typickým použitím OpenAPI dokumentu je vygenerovanie dokumentácie čitateľnej pre ľudí [24].

Príklad 2.4 ukazuje časť popisu rozhrania del.icio.us pomocou špecifikácie OpenAPI v jazyku YAML. Uvedené rozhranie del.icio.us je rovnaké fiktívne rozhranie, ktoré bolo popisované v predchádzajúcej časti pomocou jazyka WADL.

```yaml
openapi: 3.0.0
info:
  title: The del.icio.us API v1
  description: Post or retrieve your bookmarks from the social networking website
  version: 1.0.0
servers:
- url: https://api.del.icio.us
paths:
  /v1/posts/recent:
    get:
      description: Returns a list of the most recent posts.
      parameters:
      - in: query
        name: tag
        schema:
          type: string
        description: Filter by this tag.
      - in: query
        name: count
        schema:
          type: integer
          default: 15
          maximum: 100
        description: Number of items to retrieve.
      responses:
        '200':
          description: A~list of the most recent posts
          content:
            text/xml:
              schema:
                $ref: '#/components/schemas/posts'
        '401':
          $ref: '#/components/responses/AuthorizationRequired'
```

Výpis 2.4: Časť dokumentu popisujúceho fiktívne rozhranie *del.icio.us* v jazyku YAML podľa štandardu OpenAPI. Popisuje základné informácie o rozhraní a zobrazuje definíciu zdroja a metódy vrátane jej parametrov a príslušných odpovedí.

V príklade 2.4 je dátový typ odpovede metódy definovaný odkazom. Jeho definícia je ukázaná v príklade 2.5 v časti `components/schemas/posts`. Aj v tomto prípade je používaná *HTTP Basic* autentifikácia, ktorá je definovaná v časti `components/securitySchemes/Authorization`. V sekcii `security` sa definovaná *HTTP Basic* autentifikácia uplatňuje na celé API. Pre prípad neúspešnej autentifikácie je definovaná odpoveď `components/responses/AuthorizationRequired`, ktorá je použitá pomocou referencie v príklade 2.4 s chybovým kódom 401.

```yaml
components:
  securitySchemes:
    Authorization:
      description: All del.icio.us API calls must be authenticated using Basic
          HTTP auth.
      type: http
      scheme: basic
  responses:
    AuthorizationRequired:
      description: Authentication information is missing or invalid
      headers:
        WWW_Authenticate:
          schema:
            type: string
  schemas:
    posts:
      type: array
      items:
        type: object
        xml:
          name: 'post'
      xml:
        wrapped: true
security:
  - Authorization: []
```

Výpis 2.5: Časť OpenAPI dokumentu popisujúca komponenty fiktívneho rozhrania *del.icio.us*, ktoré môžu byť použité odkazom v rámci definície zdrojov a metód. Zároveň popisuje spôsob autentifikácie, chybový stav pri nesprávnej autentifikácii a v časti security uplatňuje definovanú autentifikáciu na celé API.

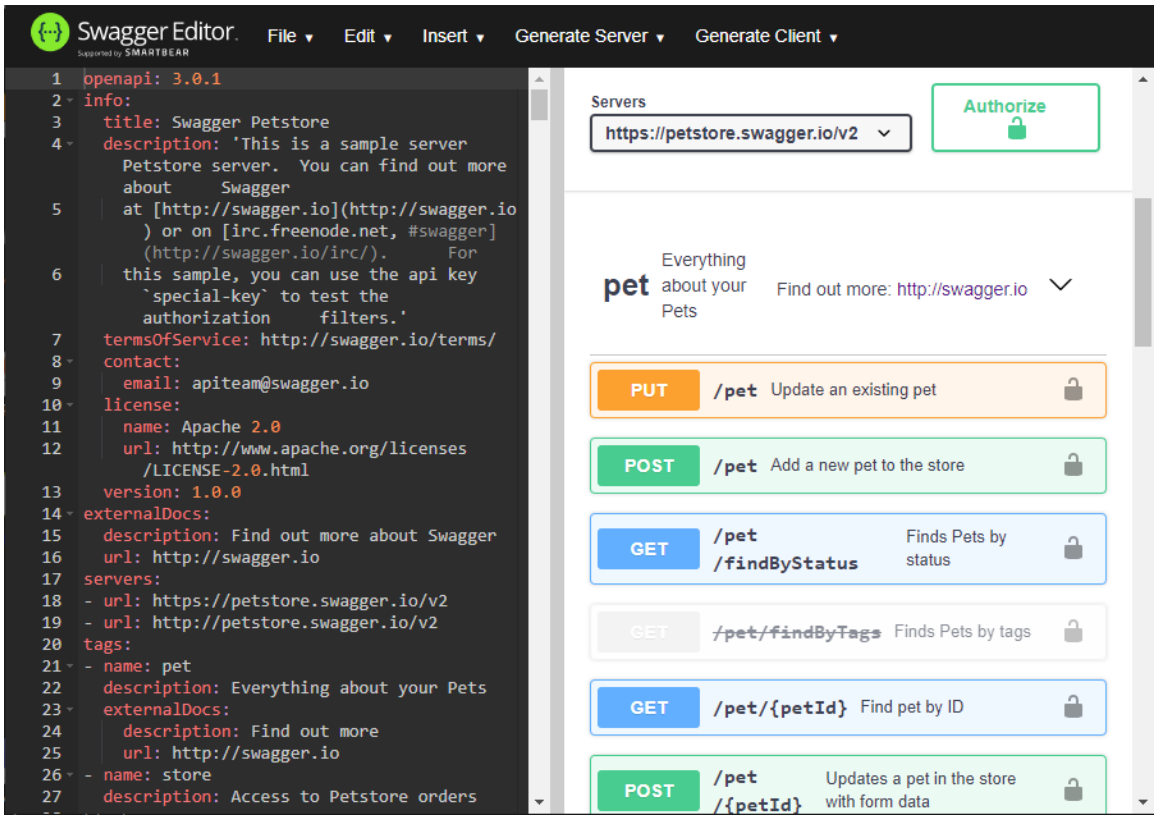
### Swagger

OpenAPI začalo prvotne vznikať ako sada *open source* nástrojov a špecifikácia, ktoré boli spoločne pomenované ako Swagger. Spoločnosť SmartBear spravujúca Swagger v roku 2015 oznámila, že podporuje vytvorenie novej spoločnosti sponzorovanej nadáciou Linux Foundation s názvom OpenAPI Initiative, ktorej úlohou bude rozšírenie Swagger špecifikácie [14]. Následkom toho bola špecifikácia Swagger premenovaná na OpenAPI [24].

Pojem Swagger sa v súčasnosti používa v súvislosti s nástrojmi, ktoré sú postavené na základe špecifikácie OpenAPI. Tieto nástroje pomáhajú pri návrhu, tvorbe, dokumentácii a práci s REST API.

Medzi hlavné Swagger nástroje patria: [33]

- **Swagger Editor** - Editor vo webovom prehliadači pre tvorbu návrhu, popisu a dokumentácie OpenAPI špecifikácie v jazyku YAML. Súčasťou editora je aj nástroj Swagger UI, ktorý zobrazuje definovanú špecifikáciu v reálnom čase (obr. 2.4).
- **Swagger UI** - Generuje interaktívnu dokumentáciu podľa OpenAPI špecifikácie.
- **Swagger Codegen** - Umožňuje generovanie serverových alebo klientskych častí kódu a dokumentácie podľa špecifikácie.




Obr. 2.4: Okno nástroja Swagger Editor používaného vo webovom prehliadači. V ľavej časti obrázka je vidieť textový editor, pomocou ktorého je možné vytvárať a upravovať špecifikáciu rozhrania v jazyku YAML. Súčasťou editora je nástroj Swagger UI, ktorý je vidieť na pravej strane. Ten graficky znázorňuje dokumentáciu špecifikovaného rozhrania v reálnom čase [30].

# Kapitola 3

# Charakteristika a princípy tvorby moderných webových aplikácií

Na moderné aplikácie sú v dnešnej dobe kladené vyššie nároky a očakávania užívateľov ako kedykoľvek predtým. Očakáva sa ich dostupnosť 24/7 z ktoréhokoľvek miesta na svete a ich použiteľnosť prakticky z akéhokoľvek zariadenia, bez ohľadu na použitý operačný systém alebo veľkosť obrazovky. Aby bol po aplikáciách požadovaný dopyt, aplikácie musia byť flexibilné, bezpečné a škálovateľné. Čoraz častejšie musia byť schopné zabezpečiť najrôznejšie užívateľské scenáre a efektívne komunikovať s webovými rozhraniami [31]. Na zabezpečenie všetkých týchto požiadaviek v súčasnosti postačuje jeden program - webový prehliadač. Webový prehliadač poskytuje prostredie pre beh aplikácií rôznych veľkostí. Webové aplikácie sa nemusia inštalovať a aktualizovať, sú prístupné z celého sveta a sú spustiteľné na takmer každom zariadení [9].

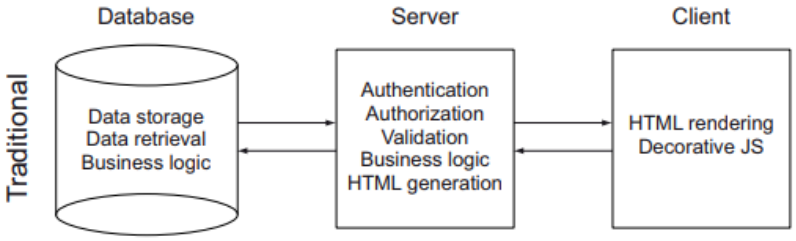
V tejto kapitole budú ukázané dva základné prístupy k tvorbe webových aplikácií. Prvým z nich je tradičný prístup popísaný v sekcii 3.1 a následne v sekcii 3.2 je ukázaný prístup k tvorbe jedno-stránkových aplikácií (*single-page applications*).

## 3.1 Tradičné webové aplikácie

Aplikácie, ktoré pracujú tradičným spôsobom, môžu byť nazývané aj multi-page aplikácie (MPA). Tieto aplikácie pre všetky dotazy, navigáciu a zmeny v aplikácii využívajú server. Každá nová operácia, ktorú klient vykoná, je zasielaná na server ako HTTP žiadosť. Žiadosť je na strane servera zachytená v prezentačnej vrstve a následne spracovaná v dátovej a biznis vrstve. Na obrázku 3.1 je možné vidieť, že väčšina aplikačnej logiky sa vykonáva na strane servera [31]. Na strane servera sa taktiež vykonáva výber nových pohľadov a ich generovanie. Zvyčajne je výsledný pohľad vytvorený zo šablóny, v ktorej sú definované zástupné miesta pre nové dáta. Po vložení nových dát do šablóny sa vytvorí nový pohľad, ktorý server odosiela vo svojej odpovedi klientovi. V prehliadači klienta dochádza k výmene pôvodnej stránky za novú a užívateľ vidí nový pohľad obsahujúci dáta o ktoré žiadal [9]. MPA aplikácie sú zvyčajne väčšie ako SPA, avšak využitím technológie AJAX[1] možno množstvo zasielaných dát medzi prehliadačom a serverom redukovať. Toto riešenie umožňuje obnovu iba niektorých častí aplikácie, ale na druhej strane pridáva komplexnosť aplikácie [19].

Najväčšou nevýhodou MPA aplikácií je, že sú pomalé. Z pohľadu biznisu to môže mať fatálne následky. Pomalé webové aplikácie môžu prinútiť užívateľov aby ich prestali použí-

[1] https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX




Obr. 3.1: Rozdelenie zodpovedností tradičnej webovej aplikácie. Aplikačná logika a generovanie pohľadov je umiestnené na strane servera. Klient iba zobrazuje pohľady, ktoré získal od servera. Prevzaté z [16].

vať a aby začali používať konkurenčné aplikácie. Jedným z dôvodov prečo sú MPA aplikácie pomalé je, že klasické MVC (*Model-View-Controller*) frameworky sú zamerané na prezentovanie stánok so statickým obsahom. Pri akcii užívateľa sa vymení celý obsah stránky (navigácia, nadpisy, text, pätička) napriek tomu, že sa vykonanou akciou nijako nezmenili [16].

## 3.2 Webové aplikácie typu single-page

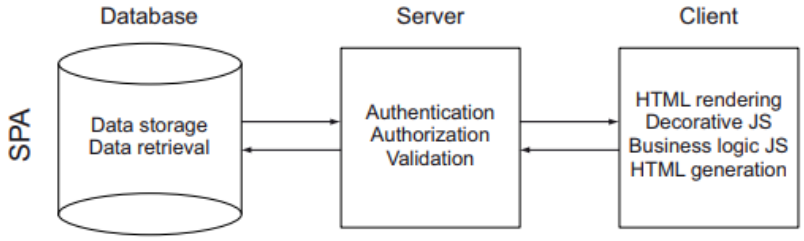
Cieľom vývojárov webových aplikácií je vytvorenie aplikácií, ktoré sa chovajú a vyzerajú ako desktop aplikácie. Spočiatku sa pre implementáciu takéhoto chovania používali technológie ako IFrames[2], Java applets, Adobe Flash a Microsoft Silverlight[3]. Neskôr sa začali vytvárať single-page aplikácie (SPA), ktoré umožňovali tento cieľ dosiahnuť za použitia jazykov integrovaných v prehliadači (JavaScript, HTML a CSS), bez nutnosti využitia ďalších doplnkov prehliadača alebo iných jazykov. Celkový návrh SPA architektúry je podobný MPA, ale má niekoľko zásadných rozdielov: [9]

- **Prehliadač neobnovuje celé stránky** - Celá SPA beží ako jediná webová stránka. Jednotlivé pohľady sú časti DOM[4], ktoré reprezentujú viditeľné oblasti stránky. Po úvodnom načítaní stránky sú všetky nástroje potrebné pre vytváranie a zobrazovanie pohľadov stiahnuté a pripravené na použitie. Nový pohľad je generovaný lokálne v prehliadači a následne dynamicky pripojený k DOM.
- **Prezentačná logika presunutá na strane klienta** - Na obrázku 3.2 je vidieť, ako sa logika generovania pohľadov presúva preč zo servera na stranu klienta.
- **Transakcie servera určené iba pre dáta** - Keďže generovanie pohľadov prebieha na strane klienta, v SPA sú typicky od servera požadované iba dáta pomocou asynchrónnych volaní.

SPA majú oproti tradičnými aplikáciám niekoľko výhod: [9]

- Ich správanie je podobné desktopovým aplikáciám napriek tomu, že bežia vo webovom prehliadači.
- Oddeľujú prezentačnú vrstvu od servera a tým umožňujú, aby boli obidve strany spravované a aktualizované samostatne.

[2] https://www.w3.org/TR/2011/WD-html5-20110525/the-iframe-element.html#the-iframe-element
[3] https://www.microsoft.com/silverlight/
[4] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model



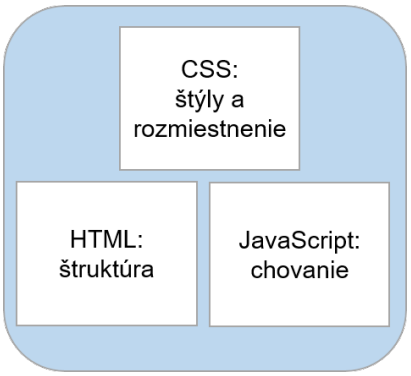

Obr. 3.2: Rozdelenie zodpovedností single-page aplikácie. Časť aplikačnej logiky a generovanie pohľadov sa presúva na stranu klienta. Prevzaté z [16].

- Žiadosti zasielané na server sú jednoduchšie a rýchlejšie, pretože sa zasielajú iba dáta namiesto celých HTML stránok.
- Sú rýchlejšie vďaka tomu, že sa na začiatku načítava iba kostra stránky a stránka sa postupne buduje pri navigácii v aplikácii.
- Sú jednoduchšie udržiavateľné vďaka oddeleniu kódu definujúceho štruktúru, štýly a chovanie do separátnych častí.

## 3.3 Technológie používané pri tvorbe klientskej časti webových aplikácií

Pri tvorbe klientskej časti webových aplikácií sa typicky využívajú tri základné jazyky, ktoré sú taktiež ilustrované na obrázku 3.3:

- **HTML** - Značkovací jazyk určený pre dokumenty zobrazované vo webovom prehliadači, ktorý popisuje ich sémantickú štruktúru.
- **CSS** - Popisuje spôsob zobrazenia HTML dokumentu vo webovom prehliadači vrátane rozloženia, farieb a použitého fontu.
- **JavaScript** - Imperatívny jazyk integrovaný vo webových prehliadačoch, umožňujúci tvorbu interaktívnych webových stránok.




Obr. 3.3: Základné technológie používané pri tvorbe klientskej časti webových aplikácií. Preložené z [9].

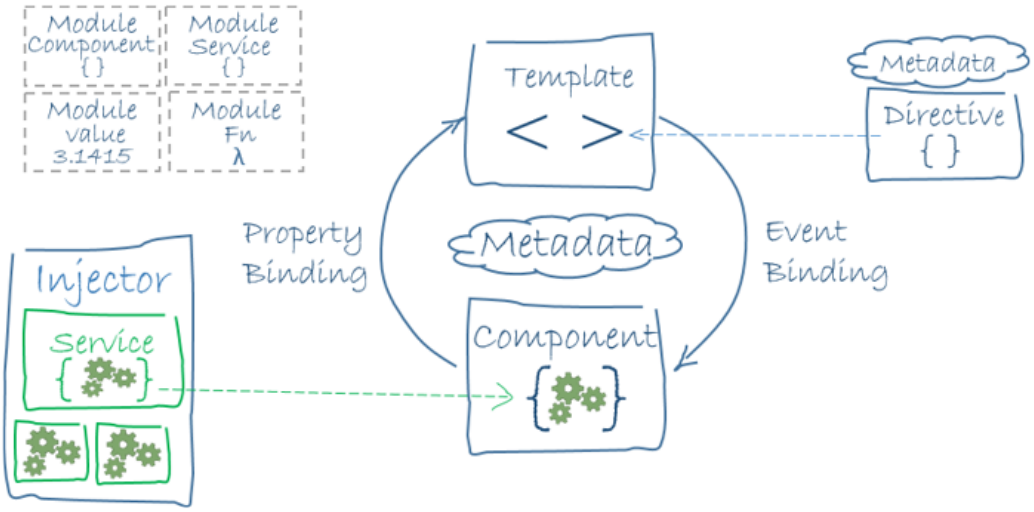
## 3.4 Framework Angular

V nasledujúcej sekcii je uvedený popis princípov frameworku Angular, ktorý bol vytvorený podľa [1], [4], [5] a [3].

Angular je platforma a framework určený pre tvorbu single-page klientskych aplikácií, za pomoci jazykov TypeScript, HTML a CSS. Angular definuje štruktúru aplikácií, čím uľahčuje prácu vývojárov v tíme a dovoľuje vývoj veľkých a udržiavateľných aplikácií.

Angular je orientovaný na prácu so samostatnými komponentami. Tie sú zoskupené do funkčných jednotiek nazývaných *ngModuly*, ktoré spoločne tvoria celú aplikáciu. Každá aplikácia používajúca Angular obsahuje minimálne jeden koreňový *NgModul*, ktorý zabezpečuje inicializáciu a načítanie aplikácie (tzv. *bootstrapping*). Koreňový modul typicky obsahuje ďalšie doplnkové moduly. Komponenty typicky definujú pohľady (*views*), ktoré sú usporiadané hierarchicky a medzi ktorými je možné definovať cesty pre navigáciu.

Na obrázku 3.4 sú zobrazené vzťahy medzi základným stavebnými prvkami frameworku Angular. Komponenta (*Component*) a šablóna (*Template*) spoločne vytvárajú pohľad (*View*). Dekorátor[5] *@Component()* komponente pridáva dodatočné informácie (*Metadata*) vrátane odkazu na príslušnú šablónu komponenty. V rámci šablón sa používajú direktívy a väzobné značky (*Binding marking*), ktoré modifikujú pohľady na základe logiky a dát programu. Komponenty využívajú služby (*Services*), ktoré sú vložené do komponenty prostredníctvom injektora (*Dependency injection*).




Obr. 3.4: Diagram zobrazujúci vzťahy medzi stavebnými prvkami frameworku *Angular*. Prevzaté z [1].

### Moduly

Modulový systém frameworku Angular využívajúci *NgModuly* je odlišný od modulového systému jazyka JavaScript[6]. V aplikácii je však možné využiť obidva systémy súčasne. *NgModuly* poskytujú kompilačný kontext pre ich komponenty, ktoré navzájom tento kontext zdieľajú. *NgModuly* sú triedy, ktoré sú označené dekorátorom `@NgModule()`. Tento dekorátor

[5] https://www.typescriptlang.org/docs/handbook/decorators.html
[6] https://exploringjs.com/es6/ch_modules.html

prijíma ako parameter objekt, ktorý sprostredkováva metadáta o module. Medzi základné vlastnosti tohto objektu patria:

- `declarations` - Určenie komponent, direktív, rúr (*pipes*) prislúchajúcich modulu.
- `exports` - Definovanie podmnožiny z declarations použiteľnej v iných moduloch.
- `imports` - Definovanie modulov, ktorých exportovanú funkcionalitu chce modul využiť.
- `providers` - Definovanie poskytovateľov služieb (*service providers*) a tokenov.
- `bootstrap` - Určenie koreňovej komponenty (*root component*).

### Služby

Pre zabezpečenie dát a funkcií, ktoré priamo nesúvisia s konkrétnymi komponentami a ktorých funkcionalitu je možné využívať naprieč aplikáciou sa používajú služby. Služby, na rozdiel od komponent, nie sú združené so šablónami, ale komponentám môžu poskytovať svoju funkcionalitu. Sú to triedy označené dekorátorom `@Injectable()`, ktoré typicky zabezpečujú špecifickú funkciu. Dekorátor prijíma ako parameter objekt s vlastnosťou `providedIn`. Pomocou nej je možné definovať prostredie, v ktorom môže byť daná služba poskytovaná. Predvolená hodnota `'root'` znamená, že sa pre danú službu vytvorí jediná inštancia, ktorá bude poskytovaná ktorejkoľvek komponente, ktorá o ňu požiada. Ukážka takéhoto použitia je uvedená v kóde 3.1.

```
1 @Injectable({
2  providedIn: 'root',
3 })
```

Výpis 3.1: Ukážka registrácie poskytovateľa služby pomocou dekorátora a metadát služby na koreňovej úrovni aplikácie. Prevzaté z [5].

Každá služba, ktorej funkcionalita má byť využitá, musí mať aspoň jedného poskytovateľa. Okrem spomínanej registrácie priamo pomocou metadát služby, je možné poskytovateľa služby registrovať aj na úrovni *NgModulov* alebo komponent. Na úrovni *NgModulov* sa služby registrujú pomocou dekorátora `@NgModule()`. V takomto prípade sa vytvorí jedna inštancia služby pre všetky komponenty patriace do daného modulu. Ukážka takéhoto použitia je uvedená vo výpise kódu 3.2.

```
1 @NgModule({
2   providers: [
3   BackendService,
4   Logger
5  ],
6  ...
7 })
```

Výpis 3.2: Ukážka registrácie poskytovateľa služby pomocou dekorátora a metadát modulu na úrovni modulu. Prevzaté z [5].

Na úrovni komponent je možné poskytovateľa služby registrovať pomocou dekorátora komponenty `@Component()`. V takomto prípade bude pre každú inštanciu komponenty vytvorená nová inštancia danej služby. Príklad takéhoto použitia je uvedený vo výpise kódu 3.3.

```
@Component({
  ...
  providers:  [ HeroService ]
})
```

Výpis 3.3: Ukážka registrácie poskytovateľa služby pomocou dekorátora a metadát komponenty na úrovni komponenty. Inšpirované podľa [5].

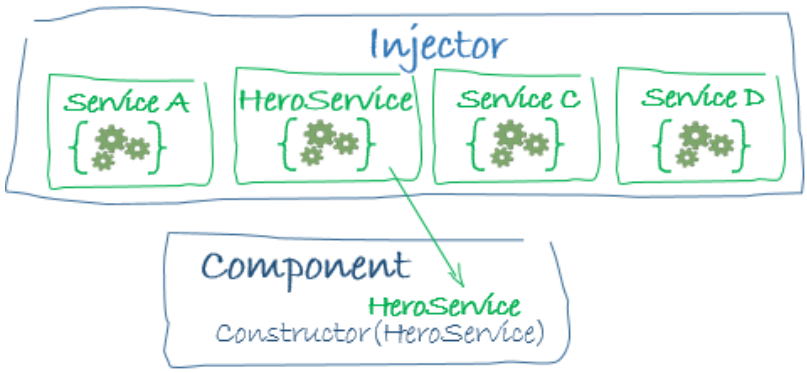
### Vkladanie závislostí

Systém vkladania závislostí (*Dependency injection*) sa využíva na sprostredkovanie služieb, funkcií alebo hodnôt pre ostatné triedy v aplikácii. Angular umožňuje vkladanie závislostí pre triedy, ktoré sú označené dekorátorom `@Injectable()`. Taktiež sa týmto dekorátorom označujú triedy, ktoré nejaké závislosti majú. Každá závislosť použitá v aplikácii musí mať registrovaného poskytovateľa (*provider*), ktorý informuje *Injector* o tom, ako získať alebo vytvoriť závislosť. Závislosti triedy (komponenty alebo služby) sa definujú v jej konštruktore, zadaním parametrov s rovnakým typom ako je typ danej závislosti (ukážka kódu 3.4).

```
constructor(private service: HeroService) { }
```

Výpis 3.4: Vkladanie závislostí (*Dependency injection*) pomocou konštruktora komponenty. Poskytuje informáciu o tom, že daná komponenta závisí na službe `HeroService`. Prevzaté z [5].

Framework Angular získava vytvorené inštancie závislostí z *Injectora* a po získaní všetkých potrebných závislostí pokračuje vo vykonávaní konštruktora. Proces vkladania závislosti (služby) je znázornený na obrázku 3.5.




Obr. 3.5: Znázornenie procesu vkladania závislosti, v tomto prípade služby s názvom `HeroService` do komponenty. *Injector* vytvorí inštanciu služby `HeroService` a poskytne jej inštanciu pre danú komponentu. Prevzaté z [5].

### Rozhrania, typy a dátové štruktúry

V jazyku TypeScript je možné dátové štruktúry reprezentovať pomocou rozhraní (*interfaces*[7]). Rozhrania ponúkajú mechanizmus ako definovať vlastnosti a metódy, ktoré musí objekt implementovať a týmto zároveň ponúkajú spôsob vytvorenia vlastných typov [28]. Pomocou rozhraní možno zabezpečiť kontroly, že potrebné dáta budú mať požadovanú štruktúru, resp. budú obsahovať vlastnosti určené rozhraním. Rozhrania sú definované kľúčovým slovom `interface` (názorná ukážka definície rozhrania je zobrazená vo výpise kódu 3.5) a pri ich definícii je možné využiť vlastnosti rôznych typov: [34]

- **Povinné** - Tieto vlastnosti musia byť v rámci dát daného typu vždy prítomné. V prípade neprítomnosti povinnej vlastnosti vznikne chyba pri typovej kontrole.
- **Voliteľné** - Vlastnosti, ktoré nie sú povinné. Ich neprítomnosť v rámci dát daného typu nespôsobí chybu pri typovej kontrole. V rámci rozhrania sú názvy voliteľných vlastností ukončené znakom `?`.
- **Iba na čítanie** - Hodnoty týchto vlastností je možné meniť iba pri vzniku objektu. Sú označené kľúčovým slovom `readonly` pred samotným názvom vlastnosti.
- **Indexovateľné typy** - Reprezentujú typy, ktoré je možné indexovať (napr. `a[10]` alebo `ageMap["daniel"]`). Využívajú označenia typov, ktoré je možné použiť pre indexovanie objektu, ako aj typ návratovej hodnoty pri indexácii.

```
1 interface IComplexType {
2   name: string;
3   address?: string;
4   readonly id: number;
5   [index: number]: string;
6 }
```

Výpis 3.5: Názorná ukážka definície rozhrania v jazyku TypeScript s názvom `IComplexType`, pozostávajúceho z povinnej vlastnosti `name`, voliteľnej vlastnosti `address`, vlastnosti iba na čítanie s názvom `id` a ďalšej indexovateľnej vlastnosti.

### Observables

Framework Angular využíva techniky vychádzajúce z návrhového vzoru *Observer*. Podstatou tohto návrhového vzoru je, že objekt (označovaný ako *subject*) spravuje zoznam závislých objektov (*observers*) a informuje ich o každej zmene stavu, resp. o nejakej udalosti. Oznámenie zároveň môže zahrňovať špecifické dáta, ktoré s týmto oznámením súvisí [22].

Využitie tohto princípu je vhodné pri implementácii komunikácie so serverom, k zabezpečeniu asynchrónnosti programu. Po zaslaní žiadosti na server je vhodné neblokovať činnosť programu čakaním na odpoveď, vykonávať ďalšie úlohy a pri získaní odpovede od servera sa vrátiť jej spracovaniu. Tento princíp je v rámci programovania označovaný pojmom reaktívne programovanie (*reactive programming*[8]). RxJS[9] (*Reactive Extensions for*

[7] https://www.typescriptlang.org/docs/handbook/interfaces.html
[8] https://en.wikipedia.org/wiki/Reactive_programming
[9] https://rxjs.dev/guide/overview

*JavaScript*) je knižnica poskytujúca funkcie pre tvorbu asynchrónnych programov a programov založených na udalostiach. Implementuje typ *Observable* a ďalšie funkcie potrebné pre jeho vytváranie a prácu s ním. Z tohto dôvodu je táto knižnica rozsiahlo využívaná v rámci frameworku Angular.

Funkcia, ktorá je určená na poskytovanie určitých hodnôt a ktorej návratový typ je *Observable* je deklaratívna. To znamená, že nebude vykonávaná pokiaľ neexistuje žiadny konzument (*Subscriber*), ktorý odoberá jej hodnoty. Konzument, ktorý je prihlásený k odberu, získava oznámenia o ukončení vykonávania odoberanej funkcie až do chvíle, kým svoj odber nezruší. Takáto funkcia môže poskytovať viacero hodnôt rôznych typov v závislosti na kontexte použitia, pričom dáta sa odosielajú ako prúd (*stream*). Bez ohľadu na typ získavaných dát, rozhranie pre odoberanie hodnôt a zrušenie odoberania ostáva nezmenené.

Ukážka použitia *Observable* je zobrazená vo výpise 3.6. Sprostredkovateľ hodnôt vytvára novú inštanciu objektu pomocou `new Observable()`, ktorá prijíma ako parameter funkciu. Táto funkcia definuje činnosti, ktoré budú vykonané, ak konzument (*Subscriber*) zavolá funkciu `subscribe()`. Tým pádom určuje postupnosť krokov, ktorá musí byť vykonaná pre získanie požadovaných hodnôt. Funkcia `subscribe()` prijíma ako parameter objekt, ktorý implementuje rozhranie *Observer*. Toto rozhranie definuje 3 metódy pre 3 typy oznámení, ktoré môže konzument obdržať:

- `next` - Povinná metóda na spracovanie doručených hodnôt.
- `error` - Voliteľná metóda pre spracovanie chybových hlásení.
- `complete` - Voliteľná metóda pre spracovanie notifikácie o ukončení vykonávania.

```
1  import { Observable } from 'rxjs';
2
3  const observable = new Observable(subscriber => {
4    subscriber.next(1);
5    setTimeout(() => {
6      subscriber.next(2);
7      subscriber.complete();
8    }, 1000);
9  });
10
11 observable.subscribe({
12   next(x) { console.log('got value ' + x); },
13   error(err) { console.error('something wrong occurred: ' + err); },
14   complete() { console.log('done'); }
15 });
```

Výpis 3.6: Ukážka vytvorenia *Observable* objektu, implementácie metód rozhrania `Observer` pre spracovanie oznámení a prihlásenie sa k odberu oznámení pomocou metódy `subscribe()`. Inšpirované podľa [29].

### HTTP Client

Framework Angular ponúka pre zabezpečenie komunikácie so serverom zjednodušené HTTP API v rámci služby s názvom *HttpClient*. Táto služba okrem iného implementuje metódy, ktoré reprezentujú základné typy HTTP žiadostí. Vďaka využitiu preťažovania metód[10] existuje pre každý typ HTTP žiadosti viacero deklarácií danej metódy, ktoré sa líšia v použitých parametroch a návratových typoch.

K zabezpečeniu funkčnosti služby *HttpClient* je potrebné importovať modul *HttpClientModule* do modulu, v rámci ktorého budú funkcie tejto služby využívané. Inštanciu služby *HttpClient* je možné v rámci triedy získať pomocou vkladania závislostí (*dependency injection*) a následne použiť jej metódy, ktoré využívajú princípy popisované v predchádzajúcich častiach tejto práce. Každá *HttpClient* transakcia využíva návratový typ *Observable*. Zároveň je možné každej transakcii priradiť typ odpovede, čo umožní pochopiteľnejšie použitie danej transakcie a jednoduchšie spracovanie odpovede. Na to je možné využiť dátové štruktúry definované pomocou rozhraní. Napriek definícii typu odpovede pre danú transakciu je reálny návratový typ odpovede závislý iba na tom, ako na danú žiadosť odpovie server.

Knižnica RxJS taktiež ponúka operátory `catchError` a `retry`, ktoré je možné využiť na spracovanie neúspešných odpovedí od servera. Operátor `retry` prijíma parameter určujúci počet opakovaní odoslania žiadosti, v prípade neúspešnej odpovede. Operátor `catchError` zabezpečuje volanie funkcie, ktorej úlohou je spracovanie neúspešnej odpovede. Príklad použitia HTTP GET metódy s návratovým typom definovaným pomocou rozhrania je uvedený vo výpise 3.7.

```
constructor(private http: HttpClient) { }

configUrl = 'assets/config.json';

export interface Config {
  heroesUrl: string;
  textfile: string;
}

getConfig(): Observable<Config> {
  return this.http.get<Config>(this.configUrl)
    .pipe(
      retry(3), // retry a failed request up to 3 times
      catchError(this.handleError) // then handle the error
    );
}
```

Výpis 3.7: Použitie HTTP GET metódy, ktorú poskytuje služba *HttpClient*. Služba je do triedy vložená ako závislosť pomocou konštruktora. Návratový typ metódy je definovaný pomocou rozhrania `Config`. Transakcia taktiež využíva operátory `retry` a `catchError`. Inšpirované podľa [3].

Framework Angular v rámci svojho HTTP API ponúka funkcionalitu, ktorá umožňuje zachytávanie odchádzajúcich HTTP žiadostí a odpovedí od servera pomocou tzv. *interceptorov*. Všetky odchádzajúce žiadosti a prichádzajúce odpovede je možné transformovať na základe funkcií definovaných v interceptore. Taktiež je možné definovať viacero intercepto-

[10] https://en.wikipedia.org/wiki/Function_overloading

rov, ktorých funkcionalitu je možné zreťaziť. Ich použitie je vhodné pre vytvorenie úloh, ktoré majú byť vykonané pri každej žiadosti alebo odpovedi, namiesto implementácie týchto úloh v rámci každej metódy samostatne.

Interceptory sa v aplikácii definujú ako triedy, ktoré implementujú rozhranie `HttpInterceptor` a jeho metódu `intercept`. V rámci implementácie tejto metódy je možné definovať potrebnú transformáciu HTTP žiadosti, ktorú funkcia získava ako prvý parameter. Metóda `intercept` získava ako druhý parameter objekt implementujúci rozhranie `HttpHandler`. Jeho metódu `handle()` je potom možné využiť na preposielanie (zvyčajne upravenej) žiadosti ďalej. Príklad vytvorenia interceptora je uvedený vo výpise 3.8.

```
@Injectable()
export class NoopInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
    return next.handle(req);
  }
}
```

Výpis 3.8: Príklad vytvorenia interceptora, ktorého úlohou je preposielanie nezmenenej žiadosti ďalej. Prevzaté z [3].

Pre zabezpečenie funkcie interceptorov je ešte potrebná registrácia ich poskytovateľov v rámci definície modulu. Poskytovatelia interceptorov sa registrujú pomocou tokenu[11] `HTTP_INTERCEPTORS`, ktorý je potrebné importovať z knižnice `@angular/common/http`. Tokenu sa potom pomocou kľúča `useClass`[12] priradí trieda implementujúca interceptor. Príklad registrácie poskytovateľa interceptora je uvedený vo výpise 3.9.

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';

@NgModule( {
    ...
    providers: [
        { provide: HTTP_INTERCEPTORS, useClass: MyInterceptor, multi: true }
    ]
})
```

Výpis 3.9: Príklad registrácie poskytovateľa interceptora s názvom `MyInterceptor` v rámci definície *NgModulu* pomocou *injection tokenu* s názvom `HTTP_INTERCEPTORS`. Inšpirované podľa [3].

[11] https://angular.io/api/core/InjectionToken
[12] https://angular.io/guide/dependency-injection-in-action#class-providers-useclass

# Kapitola 4

# Návrh nástroja pre automatické generovanie aplikácií z popisu REST rozhrania

Táto kapitola je venovaná návrhu nástroja pre automatické generovanie kostry klientskych aplikácií na základe popisu REST rozhrania. Na začiatku tejto kapitoly sú popísané niektoré z existujúcich nástrojov. Na základe vlastností existujúcich nástrojov sú v sekcii 4.2 predstavené požiadavky, ktoré budú kladené na nástroj pri jeho návrhu. V sekcii 4.3 je popísaný samotný návrh architektúry nástroja a spôsobu generovania výsledného kódu.

## 4.1 Analýza existujúcich nástrojov

V tejto sekcii sú predstavené existujúce nástroje, ktoré umožňujú generovanie klientskej časti aplikácie na základe popisu REST rozhrania v OpenAPI alebo WADL formáte. Je prediskutovaná ich funkcionalita, ich výhody aj nevýhody. Na konci tejto sekcie je vytvorený súhrn vlastností všetkých nástrojov, ktoré sumarizuje tabuľka 4.1.

### OpenAPI Generator a Swagger Codegen

*OpenAPI Generator*[1] je nástroj, ktorý umožňuje generovanie REST API klientov, serverových častí a dokumentácie na základe OpenAPI špecifikácie. Generátor vznikol z pôvodného nástroja *Swagger Codegen*, spomínaného v sekcii 2.5. Migrácia zo *Swagger Codegen* priniesla niekoľko zmien, ale základná funkcionalita ostala rovnaká. Momentálne sa oba nástroje vyvíjajú ako samostatné projekty. Dôvodom migrácie bol prechod zo špecifikácie Swagger verzie 2.0 na špecifikáciu OpenAPI verzie 3.0. Zakladajúci členovia sa tým chceli vyhnúť problémom pri správe a taktiež urýchliť vývojový cyklus [7], [8].

*OpenAPI Generator* umožňuje generovanie klientskeho kódu v rôznych jazykoch, medzi ktoré patria aj Node.js/JavaScript (ES5, ES6, AngularJS) a Typescript (AngularJS, Angular 2.x - 8.x).

Generátor je napísaný v jazyku Java, ale existuje NPM balík `openapi-generator-cli`[2], ktorý obaľuje jeho funkcionalitu. Výhodou nástroja je, že ponúka obrovské množstvo možností konfigurácie generovania. Na druhej strane sa to môže zdať aj ako nevýhoda, keďže

[1] https://github.com/OpenAPITools/openapi-generator
[2] https://www.npmjs.com/package/@openapitools/openapi-generator-cli

vyžaduje väčšie množstvo času na zoznámenie sa so všetkými možnosťami a spôsobom ich použitia. Ďalšou nevýhodou je, že nepodporuje generovanie podľa WADL špecifikácie.

### Rest client generator

NPM nátroj `rest-client-generator`[3] umožňuje generovanie koncových bodov klientov zo špecifikácie Swagger alebo z popisu v jazyku WADL. Podporuje generovanie kódu v jazyku Typescript pre platformy *Angular5* (za použitia knižnice `@angular/common/http`[4]) a *Angular2* (za použitia knižnice `@angular/http`[5]). Jeho výhodou je jednoduchosť použitia, ale možnosti konfigurácie generovania sú obmedzené iba na pár možností.

### NSwag

Nástroj *NSwag* kombinuje funkcionalitu nástrojov *Swashbuckle*[6] (generovanie OpenAPI alebo Swagger špecifikácie) a *AutoRest*[7] (generovanie knižníc pre klienta). Umožňuje generovanie klienta v jazyku TypeScript pre frameworky *jQuery*, *AngularJS*, *Angular 2+*, *Aurelia*, *KnockoutJS* a iné, zo špecifikácie Swagger alebo OpenAPI. Výhodou nástroja *NSwag* je, že ponúka grafické užívateľské rozhranie v rámci aplikácie *NSwagStudio* (obr. 4.1). Taktiež je možné ho použiť v príkazovom riadku ako NPM nástroj `nswag`[8].

### AutoRest

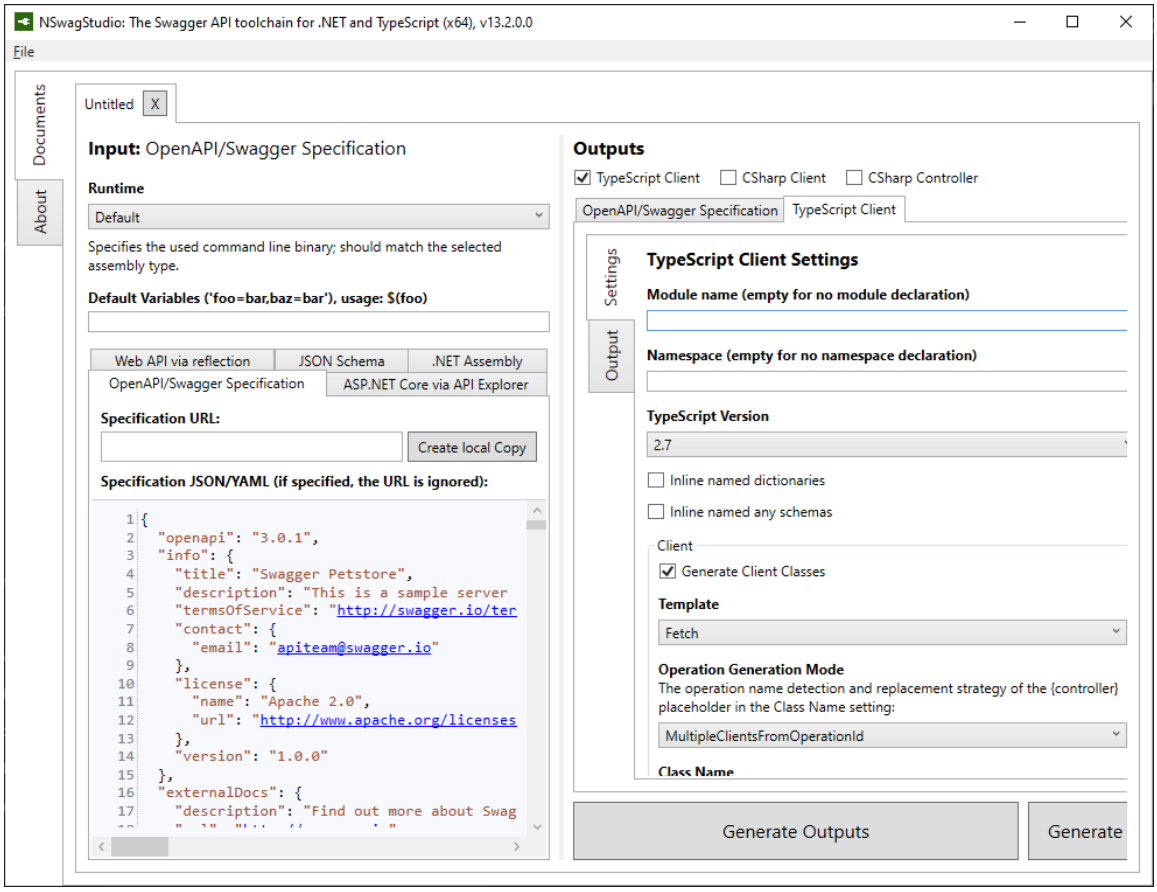
Nástroj *AutoRest* použitý ako súčasť nástroja *NSwag*, je možné využiť aj samostatne ako NPM nástroj `autorest`[9]. Nástroj je schopný generovať kód klienta z popisu OpenAPI špecifikácie. Generovanie je dobre konfigurovateľné a na výber je viacero jazykov, okrem iného aj Node.js a TypeScript. Nevýhodou je, že integráciu vygenerovaného kódu do použitého frameworku (napr. *Angularu*) je nutné vykonať ručne.

| | **Špecifikácia** | | | **Výstup** | | | |
|---|---|---|---|---|---|---|---|
| | **S** | **O** | **W** | **TS** | **Angular** | **NPM balík** | **GUI** |
| Swagger Codegen | ✓ | ✓ | ✗ | ✓ | 1.x - 2.x | `swagger-codegen` | ✗ |
| OpenAPI Generator | ✗ | ✓ | ✗ | ✓ | 1.x - 8.x | `openapi-generator-cli` | ✗ |
| Rest client generator | ✓ | ✗ | ✓ | ✓ | 2.x, 5.x | `rest-client-generator` | ✗ |
| Nswag | ✓ | ✓ | ✗ | ✓ | 1.x - 2.x | `nswag` | ✓ |
| AutoRest | ✓ | ✓ | ✗ | ✓ | ✗ | `autorest` | ✗ |

**S** = Swagger; **O** = OpenAPI; **W** = WADL; **TS** = TypeScript;
**GUI** = Grafické užívateľské rozhranie

Tabuľka 4.1: Porovnanie vlastností existujúcich nástrojov generujúcich kód klienta z popisu REST rozhrania.

[3] https://www.npmjs.com/package/rest-client-generator
[4] https://angular.io/api/common/http
[5] https://www.npmjs.com/package/@angular/http
[6] https://github.com/domaindrivendev/Swashbuckle
[7] https://github.com/Azure/autorest
[8] https://www.npmjs.com/package/nswag
[9] https://www.npmjs.com/package/autorest




Obr. 4.1: Ukážka grafického užívateľského rozhrania aplikácie *NSwagStudio*, ktorú je možné použiť pre generovanie aplikácií na základe popisu REST rozhrania [32].

## 4.2 Požiadavky kladené na navrhovaný nástroj

V súčasnosti existuje viacero nástrojov, ktoré sú schopné vygenerovať klientskú časť aplikácie z popisu REST rozhrania. Tieto nástroje boli bližšie špecifikované v predchádzajúcej sekcii 4.1. Každý z nástrojov má určité výhody a nevýhody, ktoré vo výsledku určujú prečo by bolo vhodné daný nástroj použiť alebo naopak nepoužiť.

Vývojári aplikácií pri svojej práci vo väčšine prípadov nepotrebujú GUI a rovnakú prácu sú schopný spraviť v príkazovom riadku. Z tohto hľadiska je použiteľný každý zo spomínaných nástrojov. Každý z týchto nástrojov je taktiež schopný generovať výstupný kód v jazyku TypeScript. Okrem nástroja *AutoRest* sú všetky ostatné nástroje schopné vygenerovať kód použiteľný vo frameworku Angular. Tu ale veľmi záleží na verzii frameworku, pretože novšie verzie nemusia byť nutne kompatibilné so staršími. Aktuálne najnovšia stabilná verzia frameworku Angular je verzia 9.0.0 a tá je kompatibilná s verziou 8.x. Z tabuľky 4.1 je vidieť, že iba nástroj *OpenAPI Generator* umožňuje vygenerovať kód vo verzii 8.x, kompatibilnej s najnovšou verziou frameworku Angular. Problém však nastáva v prípade, že REST rozhranie bude popísané v jazyku WADL. Z formátu WADL je schopný vygenerovať výstup iba nástroj *Rest client generator*, avšak na výber sú iba dve verzie frameworku Angular (2.x, 5.x), z ktorých ani jedna nie je kompatibilná s jeho najaktuálnejšou verziou.

Aj preto je cieľom tejto práce vytvoriť nástroj, ktorý bude odstraňovať nedostatky existujúcich nástrojov. Nástroj by mal byť schopný generovať časti kódu klientskej aplikácie z popisu REST rozhrania, vytvoreného podľa špecifikácie OpenAPI alebo WADL. Jeho výstupom by mal byť kód v jazyku TypeScript, ktorý bude ľahko integrovateľný s aktuálne najnovšou stabilnou verziou frameworku Angular. Vygenerovaná kostra aplikácie by mala zahrňovať služby pre prístup ku koncovým bodom, ktoré boli vystavené serverom. Spoločne s tým by mal vygenerovaný kód obsahovať definíciu dátových štruktúr používaných pri komunikácii so serverom. Nástroj by mal pri generovaní brať do úvahy konfiguračný súbor, ktorým bude možné modifikovať spôsob generovania a konečný výsledok.

Samotné generovanie kódu by malo prebiehať automaticky. Keďže existuje úzka väzba medzi serverom a klientom, každá zmena koncového bodu servera môže spôsobiť znefunkčnenie klienta. Preto pri každej zmene, ktorú server vykoná na REST rozhraní, bude potrebné opätovne vygenerovať patričné časti klienta. Ešte predtým bude musieť server vystaviť modifikovaný popis REST rozhrania klientovi, napríklad na špecifikovanú adresu alebo fyzicky prenesením súboru na stranu klienta.

Z hľadiska práce s nástrojom navrhujem, aby bol výsledný nástroj vytvorený ako samostatný modul, ktorý bude následne zverejnený v NPM repozitári. To zabezpečí jednoduchosť použitia a vo výsledku aj prístup k nástroju širokej verejnosti. Nástroj nebude mať žiadne grafické užívateľské rozhranie a jediná možnosť ovládania bude cez príkazový riadok. Tabuľka 4.2 sumarizuje požiadavky, ktoré sú kladené na navrhovaný nástroj.

| **Špecifikácia** | | | **Výstup** | | | | |
|---|---|---|---|---|---|---|---|
| **S** | **O** | **W** | **TS** | **Angular** | **NPM balík** | **GUI** | **CLI** |
| ✗ | ✓ | ✓ | ✓ | 8.x - 9.x | ✓ | ✗ | ✓ |

**S** = Swagger; **O** = OpenAPI; **W** = WADL; **TS** = TypeScript;
**GUI** = Grafické užívateľské rozhranie; **CLI** = Príkazový riadok

Tabuľka 4.2: Popis vlastností kladených na navrhovaný nástroj.

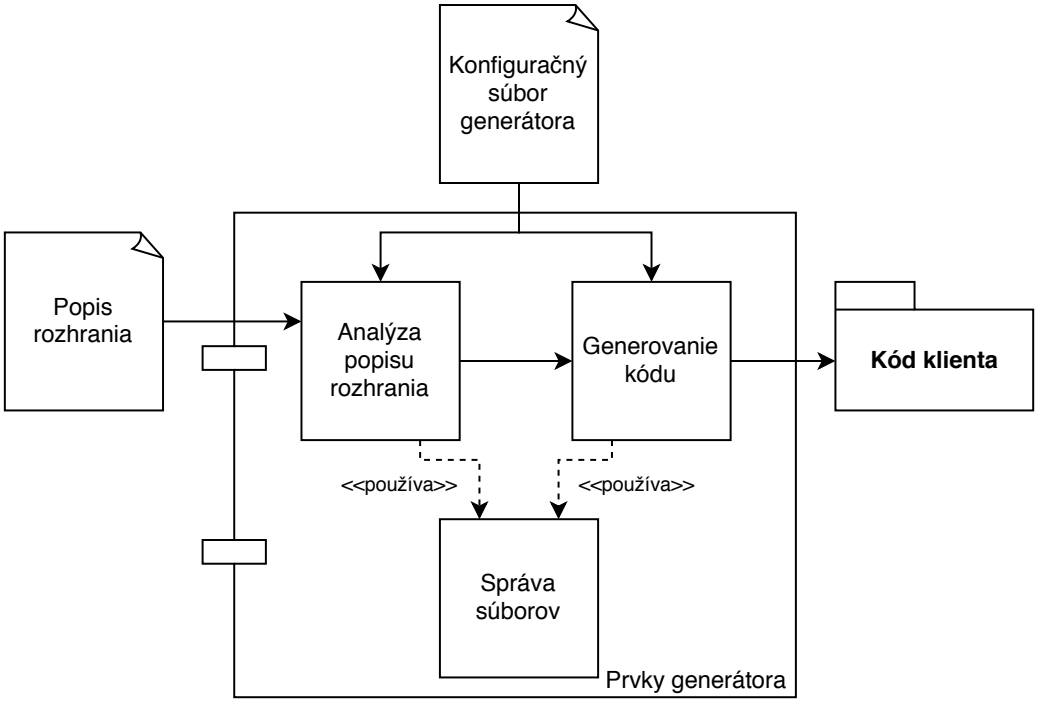
## 4.3 Návrh architektúry nástroja a spôsobu generovania výstupného kódu

V tejto sekcii je popísaný návrh nástroja pre automatické generovanie kostry klientskej aplikácie z popisu REST rozhrania. Návrh nástroja vychádza z požiadaviek popísaných v sekcii 4.2. Na začiatku sekcie bude popísaný návrh základnej štruktúry nástroja, rozdelenie zodpovedností jednotlivých komponent a spôsob ich vzájomnej kooperácie. Neskôr bude špecifickejšie popísaný návrh jednotlivých komponent. V poslednej časti bude popísaný návrh konfigurácie nástroja a spôsobu jeho integrácie do klientskej aplikácie.

### Architektúra nástroja

Pri návrhu nástroja bol kladený dôraz na oddelenie zodpovedností jednotlivých komponent. Preto výslednú architektúru nástroja možno rozdeliť do troch hlavných častí, ktorými sú: správca súborov, analyzátor popisu rozhrania a samotný generátor výstupného kódu (obr. 4.2). Tieto komponenty spolupracujú tak, aby na základe poskytnutých vstupov v po-

dobe štruktúrovaného popisu rozhrania v jazyku WADL alebo podľa špecifikácie OpenAPI a dodatočnej konfigurácie, vytvorili požadovaný výstup v podobe zdrojového kódu. Sprostredkované vstupy pritom nebudú súčasťou nástroja, ale vhodným spôsobom budú nástroju poskytnuté informácie o tom, ako ich získať alebo kde ich hľadať. Nástroj bude pri spustení z príkazového riadka očakávať určenie cesty k súboru s popisom rozhrania alebo ku konfiguračnému súboru, ktorý bude obsahovať informáciu o tom, kde sa súbor s popisom rozhrania nachádza.



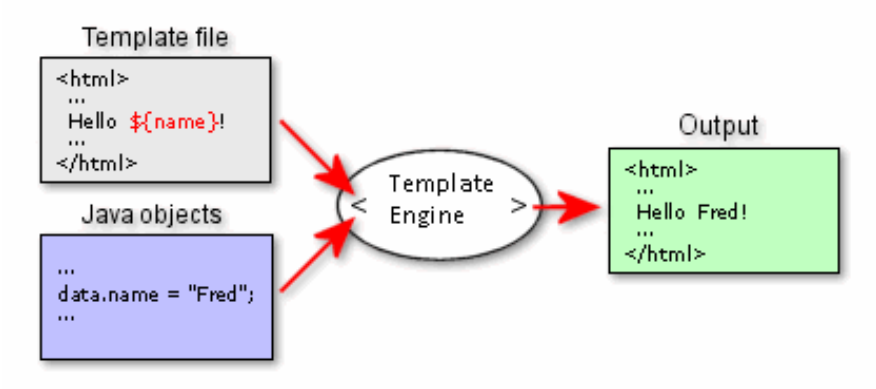

Obr. 4.2: Bloková schéma znázorňujúca navrhované časti nástroja a ich interakciu. Popis rozhrania a konfiguračný súbor nie sú súčasťou nástroja. Nástroj pozostáva z troch hlavných komponent, ktoré spolu interagujú a výstupom nástroja je samostatný modul obsahujúci vygenerovaný kód klienta.

Ako prvé sa súbor s popisom rozhrania analyzuje a všetky zistené informácie sa prevedú do jednotnej reprezentácie, ktorej bude generátor rozumieť. Generátor získa informácie z analýzy a spolu s konfiguračnými informáciami ich použije pre vygenerovanie výsledného kódu. Výsledný kód klienta bude vygenerovaný ako samostatný modul na špecifické miesto v klientskej aplikácii. Všetky operácie so súbormi ako je čítanie, zapisovanie alebo porovnávanie obsahu súborov bude zabezpečovať správca súborov. Operácie so súbormi budú potrebné pri analýze popisu rozhrania, ako aj pri generovaní kódu.

### Návrh generovania kódu klienta

Na vytvorenie výstupného zdrojového kódu bude využitý systém šablónovania. Výhodou využitia šablónovacieho systému je fakt, že umožňuje vytvoriť obsah dokumentu dynamicky na základe staticky definovaných šablón. Šablóny sú v tomto prípade textové súbory, ktoré obsahujú zástupné miesta definované pomocou konkrétneho šablónovacieho jazyka. Šablónovací systém je potom schopný tieto miesta nahradiť konkrétnymi dátami. Na obrázku 4.3 je znázornené, ako šablóna (*Template file*) obsahuje zástupné miesto (v tomto

prípade `${name}`). Systém šablónovania (*Template engine*) zabezpečí vloženie dát z objektu programovacieho jazyka na toto zástupné miesto a tak vznikne výsledný dokument.




Obr. 4.3: Znázornenie tvorby výstupného súboru za pomoci systému šablónovania. Šablóna obsahuje zástupné miesta, na ktoré je možné vložiť dynamicky dáta z programu. Šablónovací systém zabezpečí spojenie šablóny a dát a vytvorí výstupný súbor. Prevzaté z [18].

Existujúce nástroje, ktoré boli spomínané v sekcii 4.1, taktiež na generovanie výsledného kódu používajú systém šablónovania. Napríklad nástroje *OpenAPI Generator* a *Swagger Generator* používajú šablónovací systém *Mustache.js*[10] a *Rest client generator* používa *EJS*[11].

Pri analýze a spracovaní špecifikácie rozhrania sa musí brať do úvahy fakt, že na generovanie výstupných súborov bude využitý systém šablónovania. Ten značne ovplyvňuje to, ako bude vyzerať výsledný formát dát, ktoré generátor získa po spracovaní popisu rozhrania.

Z dôvodu zachovania jednoduchosti bude nástroj využívať šablónovací systém, ktorý nepodporuje žiadne komplexné operácie v rámci šablón. Všetky logické operácie budú vykonané pri analýze popisu rozhrania mimo šablóny a ich výsledky budú zahrnuté do jednotnej reprezentácie rozhrania. Reprezentácia dát bude obsahovať okrem dát popísaných v predchádzajúcej časti aj pravdivostné príznaky, ktoré pomôžu pri vytváraní výstupných zdrojových kódov. Úlohou šablónovacieho systému preto bude iba vkladanie sprostredkovaných dát na správne miesto v šablóne na základe týchto pravdivostných príznakov.

### Analýza a spracovanie popisu rozhrania

Jednou z požiadaviek kladených na nástroj je schopnosť spracovať popis rozhrania podľa špecifikácie OpenAPI alebo WADL (viď tabuľka 4.2). Nástroj bude preto disponovať dvoma analyzátormi, z ktorých jeden bude schopný analyzovať popis podľa štandardu OpenAPI a druhý z jazyka WADL. Pri analýze sa ako prvé zistí, podľa akej špecifikácie je popis rozhrania vytvorený a na základe toho sa použije príslušný analyzátor. Zmysel oboch analyzátorov je vo výsledku rovnaký. Ich cieľom je vo vhodnom tvare sprostredkovať informácie

[10] https://github.com/janl/mustache.js/
[11] https://ejs.co/

ohľadom rozhrania tak, aby bol generátor schopný vygenerovať príslušné časti kódu pre dátové štruktúry a služby pre prístup ku koncovým bodom servera.

Tieto informácie budú generátoru sprostredkované vo forme dátového objektu, ktorého formát bude rovnaký bez ohľadu na použitý štandard. To zabezpečí, že pre generovanie výsledného kódu bude možné použiť rovnakú komponentu generátora. Formát a štruktúra tohto objektu bola definovaná pomocou rozhraní jazyka TypeScript. Jeho typ bude určený rozhraním `SpecificationObject` (kód 4.1), ktorý zahrňuje všetky podstatné informácie nevyhnutné pre generovanie výstupného kódu a ktoré je možné získať z oboch špecifikácií. Rozhranie zároveň berie do úvahy, že sa výsledný kód generuje za pomoci šablónovacieho systému. Definované názvy položiek rozhrania preto zároveň určujú názvy značiek, ktoré sa môžu vyskytovať v rámci šablón.

Rozhranie `SpecificationObject` zahrňuje informácie o použiteľných adresách pre prístup k serveru, zozname dostupných modelov, zozname komplexných typov, zozname výpočtových typov, zozname metód, zozname typov parametrov metód, ako aj zozname použitých bezpečnostných schém. Ďalej obsahuje pomocné funkcie, ktoré používa šablónovací systém na formátovanie výstupu a samotnú konfiguráciu generátora.

```
interface SpecificationObject {
    servers?: string[];
    models?: ModelObject[];
    complexTypes?: ComplexType[];
    enumTypes?: EnumType[];
    parameterTypes?: ParameterType[];
    methods?: MethodObject[];
    configuration?: IGeneratorConfig;
    func?: FunctionObject;
    security?: SecurityObjects;
}
```

Výpis 4.1: Rozhranie definujúce štruktúru objektu, ktorého dáta budú použité šablónovacím systémom pre vygenerovanie výstupného kódu.

Informácie o modeloch sú definované rozhraním `ModelObject` (kód 4.2). Modely reprezentujú dátové štruktúry, ktoré sú použiteľné pre určenie návratových typov HTTP transakcií alebo typov objektov, ktoré budú posielané v rámci týchto transakcií. Toto rozhranie popisuje názov modelu a typ, podľa ktorého bol model vytvorený. Model môže byť vytvorený podľa komplexného alebo výpočtového typu.

```
interface ModelObject {
    name: string;
    type?: string;
}
```

Výpis 4.2: Rozhranie reprezentujúce informácie potrebné pre vygenerovanie dátovej štruktúry - modelu.

Potrebné informácie o komplexnom type sú definované rozhraním `ComplexType` a informácie o výpočtovom type definuje rozhranie `EnumType`. Komplexný typ je určený jeho názvom a zoznamom jeho vlastností, pričom definovanými vlastnosťami môže rozširovať iný komplexný typ. Rovnako aj výpočtový typ je určený jeho názvom a obsahuje zoznam hod-

nôt, ktoré môže dátová položka tohto typu nadobúdať. Kód 4.3 zobrazuje definíciu oboch rozhraní.

```
interface ComplexType {
    name: string;
    properties?: PropertyObject[];
    extending?: string;
}
```

```
interface EnumType {
    name: string;
    values?: EnumObject[];
}
```

Výpis 4.3: Rozhrania reprezentujúce informácie potrebné pre vygenerovanie komplexného typu (vľavo) a výpočtového typu (vpravo).

Informácie o vlastnostiach (*properties*) komplexného typu sú určené rozhraním `PropertyObject` (kód 4.4). Každá vlastnosť pozostáva z jej názvu a typu. Keďže jazyk TypeScript definuje dátové štruktúry pomocou rozhraní, je potrebné k vlastnostiam zahrnúť aj informáciu o tom, o aký typ vlastnosti sa jedná. Rozhranie `PropertyObject` preto obsahuje pravdivostné príznaky `required`, `isSimple`, `isArray` a `isAdditional`, podľa ktorých je možné za pomoci šablónovacieho systému vygenerovať správny typ vlastnosti.

```
interface PropertyObject {
    name: string;
    type: string;
    required: boolean;
    isSimple: boolean;
    isArray?: boolean;
    isAdditional?: boolean;
}
```

```
interface ParameterObject
    extends PropertyObject {
    required: boolean;
    style: string;
    in: string;
}
```

Výpis 4.4: Rozhrania reprezentujúce vlastnosť komplexného typu (vľavo) a vlastnosť typu objektu, obsahujúceho parametre metódy (vpravo).

Každej metóde, ktorá podľa špecifikácie API prijíma parametre, je potrebné tieto parametre určitým spôsobom poskytnúť. Namiesto vymenovávania jednotlivých parametrov metódy v rámci jej deklarácie je možné metóde sprostredkovať všetky parametre naraz pomocou objektu. Jednotlivé položky objektu potom reprezentujú parametre metódy. K zabezpečeniu správneho použitia takýchto objektov musia byť určené ich typy, ktorých štruktúra bude v programe určená taktiež pomocou rozhraní. Položka `parameterTypes` rozhrania `SpecificationObject`, ktorej typ je určený rozhraním `ParameterType[]` (kód 4.5) preto obsahuje zoznam dátových štruktúr reprezentujúcich objekty s parametrami metód. Tieto dátové štruktúry oproti komplexným typom, pozostávajú z vlastností definovaných rozhraním `ParameterObject` (kód 4.4). Toto rozhranie dedí vlastnosti od rozhrania `PropertyObject` (kód 4.4) a pridáva ďalšie vlastnosti dôležité pre parametre.

```
interface ParameterType {
    name: string;
    properties?: ParameterObject[];
}
```

Výpis 4.5: Rozhranie definujúce štruktúru objektu s parametrami metód.

Informácie potrebné pre vygenerovanie metódy sú definované rozhraním `MethodObject` (kód 4.6). Obsahuje informácie o názve metódy, ceste ku koncovému bodu, použitom type HTTP metódy, zozname parametrov, tele metódy, návratovom type pri úspešnej odpovedi servera, zozname kódov a popisov chýb pri neúspešnej odpovedi od servera a o použitých bezpečnostných schémach. Taktiež definuje pravdivostné príznaky, ktoré sú v tomto prípade definované pomocou funkcií a ich vyhodnotenie zabezpečuje šablónovací systém.

```
interface MethodObject {
    name: string;
    httpMethod: HttpMethod;
    pathParameters: ParameterObject[];
    queryParameters: ParameterObject[];
    headerParameters: ParameterObject[];
    cookieParameters: ParameterObject[];
    parametersType: string;
    requestBodies: MediaTypeObject[];
    successResponse: ResponseObject;
    failResponses: ResponseObject[];
    path: string;
    security: SecurityObjects;
    hasSuccessResponse: boolean;
    consumeFormData: () => boolean;
    hasRequestBody: () => boolean;
    hasMediaTypeSetting: () => boolean;
    hasNullBody: () => boolean;
    hasHeaders: () => boolean;
    hasParameters: () => boolean;
    hasFailResponses: () => boolean;
    hasCookieParams: () => boolean;
}
```

Výpis 4.6: Rozhranie určujúce štruktúru informácií potrebných pre vygenerovanie metódy.

Informácie o úspešných aj neúspešných odpovediach od servera sú definované rozhraním `ResponseObject` (kód 4.7). Toto rozhranie zahrňuje kód odpovede, jej popis a prípadne informácie o dátach, ktoré sú v odpovedi zahrnuté. Dáta zahrnuté v odpovedi sú popísané rozhraním `MediaTypeObject` (kód 4.7), ktoré okrem iného zahrňuje informácie o ich MIME type[12] a prípadne o príslušnej dátovej štruktúre, ktorá tieto dáta reprezentuje.

Nástroj bude podporovať zavedenie bezpečnostných schém pre autentifikáciu na úrovni celého API alebo pre konkrétne metódy. Pri použití autentifikačnej schémy na úrovni API bude táto schéma použitá pre všetky metódy. Informácie o použitých bezpečnostných schémach sú definované rozhraním `SecurityObjects` (kód 4.8), pričom podporované sú 4 typy bezpečnostných schém: Basic HTTP[13], Bearer Token[14], Header a Query API keys[15]. Bezpečnostné schémy je možné kombinovať a informácie o samostatnej schéme definuje rozhranie `SecurityObject` (kód 4.8).

[12] https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types
[13] https://tools.ietf.org/html/rfc7617
[14] https://tools.ietf.org/html/rfc6750
[15] https://swagger.io/docs/specification/authentication/api-keys/

```
interface ResponseObject {
    content: MediaTypeObject[];
    code: string;
    description?: string;
}
```

```
interface MediaTypeObject {
    mediaType?: string;
    schema?: string;
    required?: boolean;
    isFormDataMime?: boolean;
    isLast?: boolean;
}
```

Výpis 4.7: Rozhrania určujúce štruktúru informácií, potrebných pre definovanie návratových typov HTTP metód, kódov a popisov odpovedí.

```
interface SecurityObjects {
    basicHttpSchemas: SecurityObject[];
    bearerHttpSchemas: SecurityObject[];
    headerApiKeySchemas: SecurityObject[];
    queryApiKeySchemas: SecurityObject[];
    usedApiKeys?: string[];
}
```

```
interface SecurityObject {
    scope: string[];
    schemeName: string;
    name: string;
}
```

Výpis 4.8: Rozhrania určujúce štruktúru informácií, potrebných pre špecifikáciu použitých bezpečnostných schém na úrovni celého API alebo na úrovni jednotlivých metód.

### Automatická detekcia zmien rozhrania

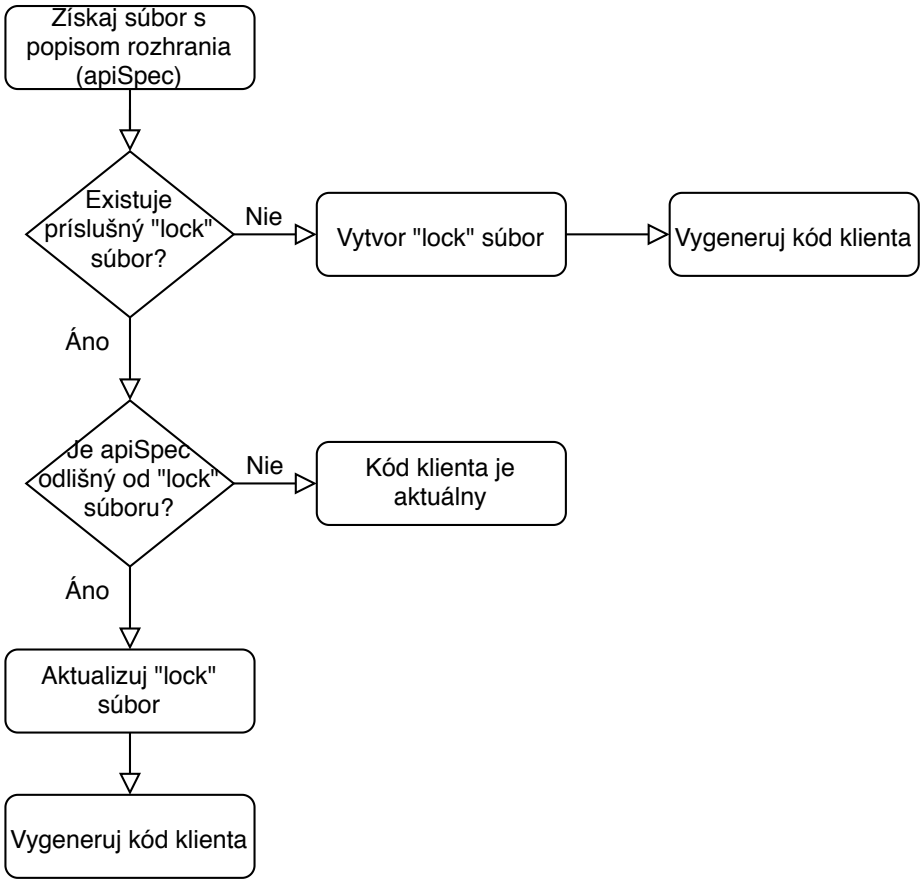
Návrh procesu automatickej detekcie zmien v popise rozhrania je zobrazený na vývojovom diagrame 4.4. Pred samotným generovaním kódu klienta sa vykoná popísaný proces, pomocou ktorého sa bude zisťovať aktuálnosť vygenerovaného kódu. Pri prvotnom vygenerovaní kódu sa vytvorí kópia popisu rozhrania, ktorá sa uloží ako nový súbor. Táto kópia bude vo svojom názve obsahovať príponu `-lock`, nasledovanú príponou súboru (ďalej ako *lock* súbor). Súbor *lock* bude vždy obsahovať verziu špecifikácie rozhrania, podľa ktorej bol výsledný kód vygenerovaný. Pri každom spustení generátora sa zistí, či súbor popisujúci rozhranie odpovedá *lock* súboru. V prípade, že nastala zmena v popise rozhrania, kód klienta bude vygenerovaný nanovo a súbor *lock* bude aktualizovaný.

### Návrh konfigurácie a integrácie do klientskej aplikácie

Konfigurácia nástroja má používateľovi do určitej miery umožniť prispôsobenie vygenerovaného kódu. Ku konfigurácii nástroja bolo možné pristúpiť dvoma spôsobmi:

1. Konfigurovať nástroj pomocou parametrov pri spúšťaní programu.
2. Vytvoriť samostatný konfiguračný súbor a nástroju sprostredkovať cestu ku tomuto súboru pomocou parametra pri jeho spúšťaní.

Pri návrhu som sa rozhodol využiť druhú možnosť a konfiguráciu nástroja oddeliť do samostatného súboru. Tento prístup umožní používateľovi nástroja zmeniť jeho konfiguráciu, bez nutnosti zmeny príkazu pre jeho spúšťanie. Používateľovi nástroja s tým otvára možnosť definovať skript pre spúšťanie nástroja a pri akejkoľvek zmene konfigurácie tento skript nemusí zmeniť. Príkladom takéhoto skriptu môže byť definovanie konzolového príkazu v rámci súboru `package.json` v časti `scripts`. Keďže framework Angular, Angular




Obr. 4.4: Vývojový diagram znázorňujúci proces automatického vyhodnocovania potreby generovania kódu. Popis rozhrania (*apiSpec*) sa porovnáva s *lock* súborom, ktorý uchováva kópiu popisu rozhrania, podľa ktorého bol kód klienta naposledy vygenerovaný. V prípade zistenia rozdielu medzi obsahom nového popisu rozhrania a súboru *lock* prebehne opätovné vygenerovanie kódu a súbor *lock* sa aktualizuje.

CLI, ale aj komponenty ktoré využíva, sú zabalené a distribuované ako NPM balíky, súbor `package.json` je súčasťou každej aplikácie využívajúcej framework Angular [2].

Pomocou konfiguračného súboru bude používateľ schopný modifikovať vygenerovaný kód. Príkladom konfigurácie môže byť určenie výstupného adresára, do ktorého majú byť vygenerované súbory so zdrojovými kódmi umiestnené alebo pomenovania súborov. Jediným povinným atribútom konfiguračného súboru je určenie cesty k súboru s popisom REST rozhrania, ak tá nie je definovaná parametrom programu. Súbor s popisom rozhrania nemusí byť uložený lokálne a cesta k nemu môže byť definovaná pomocou webovej adresy. Príklad konfiguračného súboru nástroja v jazyku JSON je uvedený vo výpise 4.9. Zoznam všetkých použiteľných možností konfigurácie je uvedený v tabuľke 4.3.

```
{
  "apiSpec": "https://raw.githubusercontent.com/OAI/OpenAPI-Specification/
              master/examples/v3.0/petstore.yaml",
  "outputDir": "./src/app/gateway",
  "parameterTypeSuffix": "Parameters",
  "retryFailedRequest": 2,
  "forceGeneration": true,
  "parametersAsObject": true
}
```

Výpis 4.9: Príklad konfiguračného súboru nástroja v jazyku JSON.

| **Názov** | **Dátový typ** | **Povinný** | **Implicitná hodnota** | **Popis** |
|---|---|---|---|---|
| apiSpec | string | ✓[1] | ✗ | Cesta/web URI k súboru s popisom rozhrania. |
| outputDir | string | ✗ | “.” | Cesta k výstupnému adresáru. |
| interfacePrefix | string | ✗ | “I” | Predpona názvu rozhrania. |
| interfaceSuffix | string | ✗ | ✗ | Prípona názvu rozhrania. |
| capitalizeTypeName | boolean | ✗ | true | Veľké písmená názvov typov. |
| typePrefix | string | ✗ | ✗ | Predpona názvu typu. |
| typeSuffix | string | ✗ | ✗ | Prípona názvu typu. |
| parameterTypePrefix | string | ✗ | ✗ | Predpona názvu rozhrania parametrov metód. |
| parameterTypeSuffix | string | ✗ | “Params” | Prípona názvu rozhrania parametrov metód |
| moduleName | string | ✗ | “gateway” | Predpona názov modulu. |
| serviceName | string | ✗ | “gateway” | Predpona názvu služby. |
| interceptorName | string | ✗ | “gateway” | Predpona názvu interceptora. |
| retryFailedRequest | number | ✗ | ✗ | Počet opakovaní HTTP transakcie pri neúspešnej odpovedi. |
| forceGeneration | boolean | ✗ | false | Vynútenie generovania bez zmeny rozhrania. |
| parametersAsObject | boolean | ✗ | false | Parametre metód zoskupené do objektov. Ich typy sú pridané k dátovým štruktúram. |

[1] Povinný parameter iba v prípade, ak cesta k popisu rozhrania nie je určená parametrom programu.

Tabuľka 4.3: Zoznam použiteľných konfiguračných možností nástroja pre prispôsobenie vygenerovaného zdrojového kódu. Názov určuje názov položky v konfiguračnom súbore v jazyku JSON. Tabuľka ďalej zobrazuje dátový typ položky, povinnosť jej použitia, implicitnú hodnotu, ktorá bude použitá, ak daná položka nebude v konfigurácii prítomná a jej popis.

# Kapitola 5

# Implementácia nástroja

Táto kapitola sa zaoberá implementáciu nástroja podľa vytvoreného návrhu popísaného v kapitole 4. V úvode tejto kapitoly je uvedený popis technológií, ktoré boli využívané pri implementácii nástroja. Kapitola sa taktiež zaoberá štruktúrou a konfiguráciou projektu a ďalej sa sústredí na popis implementácie jednotlivých častí. V závere tejto kapitoly je popísaná štruktúra generovaného zdrojového kódu.

## 5.1 Technológie použité pri implementácii nástroja

Pri implementácii nástroja boli používané viaceré technológie a nástroje tretích strán. Ako implementačný jazyk bol zvolený jazyk *TypeScript* z dôvodu, že podporuje princípy objektovo orientovaného programovania a umožňuje silné typovanie. Behové prostredie kódu zabezpečuje *Node.js* a na tvorbu obsahu výstupných súborov bol použitý šablónovací systém *Mustache.js*. Samotné programovanie prebiehalo vo vývojovom prostredí *Visual Studio Code*. V nasledujúcej časti bude uvedený krátky popis týchto technológií.

### Node.js

Podstata nástroja určovala podmienky pri výbere použitých technológií. Nástroj má byť schopný vygenerovať zdrojový kód, ktorý bude potrebné zapisovať do súborov. Na to je však potrebná technológia, ktorá podporuje prácu so súborovým systémom a umožňuje vstupno-výstupné operácie. Taktiež som chcel zúžitkovať nadobudnuté skúsenosti s programovacím jazykom JavaScript a preto rozhodnutie padlo na použitie technológie Node.js.

Node.js[1] je *open source* behové prostredie pre jazyk JavaScript, ktoré je nezávislé na použitej platforme. Node.js spája výhody programovacieho jazyka JavaScript a princípy, ktoré boli predtým používané iba v rámci operačných systémov. Node.js používa *JavaScriptový engine V8*[2], tvoriaci jadro prehliadača *Google Chrome*, mimo webový prehliadač. *Engine V8* kód v jazyku JavaScript neinterpretuje, ale kompiluje. Z tohto dôvodu umožňuje využitie vlastností vysokoúrovňového programovacieho jazyka ako správa pamäte alebo dynamické typovanie a zároveň umožňuje využiť výkon kompilovaného jazyka. Jazyk JavaScript v rámci Node.js ponúka funkcie systémového jazyka a to napríklad správu vyrovnávacích pamätí, prúdové spracovanie vstupov a výstupov, prácu so sieťovými soketmi a hlavne manipuláciu so súbormi [23].

[1] https://nodejs.dev/introduction-to-nodejs
[2] https://v8.dev/

### Jazyk Typescript

TypeScript je jazyk a zároveň sada nástrojov umožňujúca generovanie kódu v jazyku JavaScript. Je to objektovo orientovaný jazyk, ktorý je nadmnožinou jazyka JavaScript a na generovanie čistého JavaScript kódu používa kompilátor. Vygenerovaný kód môže využiť všetky dostupné nástroje, frameworky a knižnice jazyka JavaScript. Jeho objektová orientácia umožňuje využiť pri tvorbe aplikácií osvedčené objektovo orientované techniky a návrhové vzory. Jazyk TypeScript je plne podporovaný viacerými populárnymi JavaScript frameworkami, medzi ktoré patria aj *Angular*, *React*, *Vue*, *Ember*, *Meteor*, *Backbone*, *Aurelia* či *Polymer*. Jazyk TypeScript prináša niekoľko výhod: [28]

- **Kompilácia** - Obrázok 5.1 ilustruje kompiláciu kódu v jazyku TypeScript (TS) do JavaScript (JS) kódu. Kompilátor pri kompilácii kódu kontroluje syntax a zobrazuje prípadné chyby.
- **Silné typovanie** - Nepripúšťa zmenu typu premennej za behu programu.
- **Integrácia s populárnymi JavaScript knižnicami** - TypeScript využíva definičné súbory s príponou `.d.ts`. Tie obsahujú informácie popisujúce každú dostupnú funkciu alebo premennú knižnice spolu s príslušnými anotáciami typov.
- **Zapúzdrenie** - TypeScript je schopný definovať dáta a operácie nad dátami v rámci jednej komponenty, pričom ako väčšina iných programovacích jazykov na to využíva triedy.
- **Modifikátory prístupu** - Koncept skrývania dát umožňuje využívať privátne a verejné premenné vďaka modifikátorom prístupu *private* a *public*.


Obr. 5.1: Kód v jazyku TypeScript (TS) je skompilovaný pomocou kompilátora do kódu v jazyku JavaScript (JS).

### Mustache.js

Spomedzi veľkého množstva existujúcich šablónovacích systémov podporujúcich jazyk JavaScript bol vybraný *Mustache.js*[3]. Tento šablónovací systém umožňuje vytváranie zdrojových kódov pomocou nahradzovania značiek umiestnených v šablónach za poskytnuté dáta. *Mustache.js* nepodporuje využívanie logických operácií v šablónach. Syntax systému *Mustache.js* neobsahuje žiadne *if* podmienky, *else* klauzuly alebo *for* cykly. Namiesto toho umožňuje iba vkladanie značiek, pričom niektoré sú nahradené dátami, iné naopak nie alebo sú nahradené sériami hodnôt.

Šablóny sú v tomto prípade textové súbory, ktoré obsahujú neobmedzený počet značiek. Značky sú identifikované podľa dvojice zložených zátvoriek, ktoré obklopujú kľúč dátovej položky. Systém *Mustache.js* z dôvodu bezpečnosti pri jeho používaní v rámci HTML automaticky kóduje znaky. Na zachovanie pôvodných nezakódovaných hodnôt sa namiesto dvoch zátvoriek používajú tri alebo sa na začiatok kľúča vkladá znak &.

[3] https://github.com/janl/mustache.js

### NPM

NPM (*Node Package Manager*) je online repozitár (dátové úložisko) pre publikovanie *open source* Node.js projektov a zároveň konzolový nástroj určený na interakciu s týmto úložiskom. Pomocou neho je možné inštalovať balíky nachádzajúce sa v repozitári, spravovať ich verzie alebo spravovať balíky, na ktorých je projekt závislý. Na uľahčenie spravovania závislostí Node.js projektu sa využíva súbor `package.json` obsahujúci zoznam balíkov, na ktorých je projekt závislý. Tie je možné nainštalovať jednoducho využitím príkazu `npm install` [21].

### Použité nástroje tretích strán

Pri implementácii som využíval existujúce nástroje, ktoré do značnej miery uľahčili celý vývojový proces. Ich použitie umožnilo zamerať sa na implementáciu dôležitých častí nástroja a pri tom využiť ich funkcionalitu. Všetky použité nástroje pochádzajú z NPM repozitára a ich použitie je uvedené v súbore `package.json` v časti `dependencies` a `devDependencies`. Zoznam použitých nástrojov priamo korešponduje so zoznamom uvedeným v tabuľke 5.1. V časti `devDependencies` sú uvedené balíky, ktoré boli potrebné iba pri vývoji nástroja. Tieto balíky nie sú priamo súčasťou nástroja a k zabezpečeniu ich funkcie je potrebná ich inštalácia pomocou príkazu `npm install` v koreňovom adresári nástroja. Všetky ostatné balíky (v tabuľke 5.1 umiestené v rámci produkčného prostredia) sú automaticky nainštalované pri inštalácii nástroja z NPM úložiska.

### Visual Studio Code

Visual Studio Code[4] je populárne vývojové prostredie medzi vývojármi aplikácií. Je vytvorené spoločnosťou *Microsoft*, ale okrem systému *Windows* je jeho použitie možné aj na operačných systémoch *macOS* a *Linux*. Tento editor zdrojového kódu ponúka zabudovanú podporu pre JavaScript, TypeScript a Node.js, ale jeho použitie je možné aj pre ďalšie jazyky. Medzi jeho hlavné výhody patria:

- *IntelliSense* - Automatické zvýrazňovanie kódu podľa syntaxe jazyka a doplňovanie dostupných možností na základe definovaných typov, funkcií a importovaných modulov.
- Podpora ladenia zdrojového kódu (*debugging*).
- Podpora práce so systémom pre správu verzií zdrojového kódu (napr. *Git*[5] alebo podobné).
- Ponúka možnosť inštalácie veľkého množstva rozšírení.

[4] https://code.visualstudio.com/
[5] https://git-scm.com/

| Prostredie | Názov | Verzia | Popis použitia |
|---|---|---|---|
| Produkčné | lodash | ^4.17.15 | Iterácia nad prvkami objektov a polí. |
| | mustache | ^4.0.1 | Použitý šablónovací systém. |
| | openapi-types | ^1.3.5 | Využitie definovaných typov štandardu OpenAPI. |
| | xml2js | ^0.4.23 | Prevod WADL špecifikácie definovanej v jazyku XML na objekt v jazyku JavaScript. |
| | xml2js-xpath | ^0.11.0 | XPath dotazovanie nad objektami v jazyku JavaScript vytvorenými pomocou nástroja xml2js. |
| | yaml | ^1.9.2 | Spracovanie obsahu textového súboru v jazyku YAML a jeho prevod na objekt jazyka JavaScript. |
| | yargs | ^15.3.1 | Spracovanie parametrov Node.js programu. |
| | valid-url | ^1.0.9 | Zisťovanie, či je definovaná cesta k zdroju (súboru) webová adresa alebo sa jedná o lokálny súbor. |
| | mkdirp | ^1.0.4 | Vytvorenie adresárovej štruktúry pri zapisovaní súborov. |
| | typescript | ^3.8.3 | Jazyk TypeScript. |
| Vývojové | @types/lodash | ^4.14.150 | Definície typov pre balík lodash. |
| | @types/mustache | ^4.0.1 | Definície typov pre balík mustache. |
| | @types/valid-url | ^1.0.3 | Definície typov pre balík valid-url. |
| | @types/node | ^12.12.37 | Definície typov pre Node.js. |
| | @types/yargs | ^15.0.4 | Definície typov pre balík yargs. |
| | ts-node-dev | ^1.0.0-pre.44 | Využívanie automatického reštartu nástroja pri zmene zdrojového kódu. |
| | tslint | 5.18.0 | Formátovanie zdrojového kódu. |

Tabuľka 5.1: Zoznam použitých externých balíkov získaných z NPM úložiska, ktoré nástroj využíva k svojej činnosti. Balíky uvedené v tabuľke v časti vývojového prostredia boli využívané iba pri implementácii nástroja. Označenie verzie symbolom ^ znamená použitie akejkoľvek verzie tohto nástroja, ktorá je kompatibilná s uvedenou verziou.

## 5.2 Štruktúra a konfigurácia projektu

Pred samotným vývojom nástroja bolo nutné vytvoriť vhodnú štruktúru projektu a doplniť potrebné konfiguračné súbory. Ako prvé bolo potrebné inicializovať nový Node.js projekt pomocou príkazu `npm init` v koreňovom adresári projektu. Vďaka tomuto príkazu je možné vytvoriť súbor `package.json` využitím sprievodcu, ktorý žiada o doplnenie základných informácii o projekte ako názov, verziu, popis, vstupný bod projektu, testovací príkaz, adresu git repozitára, kľúčové slová, informácie o autorovi a výber licencie. Po inicializácii projektu som následne vytvoril jeho adresárovú štruktúru. Hlavným rozdielom Node.js projektu pri využití jazyka TypeScript je práve jeho adresárová štruktúra. Keďže zdrojové kódy napísané v jazyku TypeScript je nutné kompilovať do čistého JavaScriptu (viď sekcia 5.1), je potrebné ich rozumne oddeliť. Práve preto sú všetky zdrojové kódy v jazyku TypeScript

umiestnené do adresára `src` a zdrojové kódy, ktoré vznikli po kompilácii sú umiestnené do adresára `dist`. Samotný adresár `src` obsahuje okrem zdrojových kódov nástroja ďalšie dva podadresáre. Podadresár s názvom `types` obsahuje definície typov a podadresár `templates` obsahuje definované *Mustache.js* šablóny. K projektu bolo taktiež potrebné pridať pomocné súbory `.gitignore` a `README.md`. Po nainštalovaní jazyka TypeScript bol v rámci projektu vytvorený konfiguračný súbor `tsconfig.json` pomocou príkazu `tsc --init`.

### Konfigurácia jazyka TypeScript

Jazyk TypeScript používa konfiguračný súbor `tsconfig.json`[6] pre správne nastavenie kompilátora, ako aj vstupných súborov a ďalších nastavení potrebných pre kompiláciu projektu. Popis použitých nastavení kompilátora je uvedený v tabuľke 5.2. Tieto nastavenia sú v súbore `tsconfig.json` súčasťou objektu s kľúčom `compilerOptions`. Kompilácia zdrojového kódu potom prebieha pomocou príkazu `tsc` z koreňového adresára projektu alebo využitím definovaného skriptu v súbore `package.json`, ktorý je možné spustiť príkazom `npm run build`.

| **Použitá vlastnosť** | **Hodnota** | **Popis** |
|---|---|---|
| outDir | ./dist | Názov adresára, do ktorého budú umiestnené zdrojové kódy po kompilácii v jazyku JavaScript. |
| sourceMap | true | Generuje príslušné `.map` súbory potrebné pri ladení kódu. |
| declaration | true | Generuje príslušné `.d.ts` súbory, ktoré sprostredkúvajú typové informácie pre kód napísaný v jazyku JavaScript. |
| module | CommonJS | Definovanie štandardu pre moduly výstupného kódu. Node.js používa štandard *CommonJS*[7]. |
| moduleResolution | node | TypeScript sa pokúša napodobniť spôsob aký používa Node.js pre rozlišovanie modulov[8]. |
| target | es2015 | Špecifikácia jazyka JavaScript, podľa ktorej bude výstupný kód vygenerovaný a ktorú zároveň Node.js podporuje[9]. |

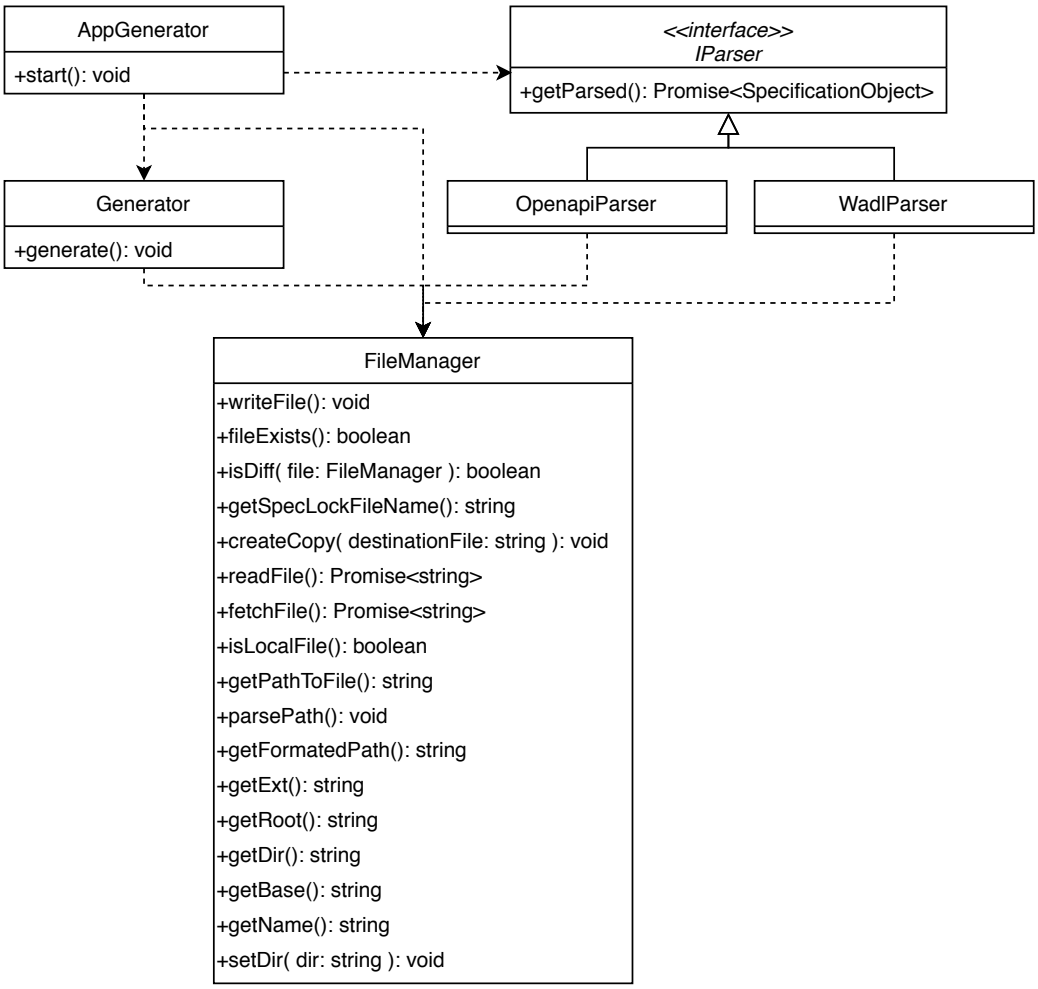
Tabuľka 5.2: Zoznam použitých nastavení TypeScript kompilátora definovaných v konfiguračnom súbore s názvom `tsconfig.json`. Všetky popísané nastavenia sú súčasťou objektu `compilerOptions`.

### Diagram tried

Pre jednotlivé komponenty nástroja popisované pri návrhu architektúry (viď sekcia 4.3) boli vytvorené samostatné triedy. Obrázok 5.2 zobrazuje diagram tried, ktorý znázorňuje vzťahy medzi triedami a zobrazuje ich použiteľné verejné metódy. Základom je trieda `AppGenerator`, ktorej metóda `start()` zabezpečí zahájenie činnosti nástroja. Táto trieda riadi činnosť nástroja a určuje tok programu. Pre analýzu a spracovanie API špecifikácie boli

[6] https://www.typescriptlang.org/docs/handbook/tsconfig-json.html
[7] https://nodejs.org/docs/latest/api/modules.html
[8] https://nodejs.org/api/modules.html#modules_all_together
[9] https://nodejs.org/en/docs/es6/

vytvorené dve triedy, ktoré implementujú rozhranie `IParser`. Trieda `OpenapiParser` zabezpečuje spracovanie OpenAPI špecifikácie, zatiaľ čo trieda `WadlParser` spracováva špecifikáciu API definovanú pomocou WADL. Obidve implementujú verejnú metódu `getParsed()`, ktorá vracia *Promisu*[10] s dátovým objektom typu `SpecificationObject` (kód 4.1). Tento objekt obsahuje dáta potrebné pre vygenerovanie výstupného kódu a je predaný triede `Generator` pri vytváraní jej inštancie. Tá zabezpečuje samotné generovanie kódu a jej metóda `generate()`, doplňuje dáta do definovaných šablón za pomoci šablónovacieho systému. Výsledný kód je zapísaný do súborov pomocou triedy `FileManager`. Táto trieda poskytuje operácie pre uľahčenie práce so súborom a je využívaná aj v rámci ostatných tried.




Obr. 5.2: Diagram tried nástroja pre generovanie kódu klientskej aplikácie z popisu REST rozhrania. Diagram zobrazuje všetky implementované triedy, rozhrania a ich verejné metódy a modeluje ich vzájomné vzťahy.

[10] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

## 5.3 Popis implementačných detailov

Vstupným bodom programu je súbor `src/main.ts` resp. `dist/main.js` po jeho kompilácii, ktorý je určený v rámci konfiguračného súboru `package.json` pod položkou `main`[11]. Po spustení programu sa vytvorí inštancia triedy `AppGenerator` a zavolá sa jej metóda `start()`.

### AppGenerator

Trieda `AppGenerator` reprezentuje samotný nástroj a jej metóda `start()` určuje ďalší tok programu. Trieda má na starosti aj spracovanie parametrov programu, k čomu využíva knižnicu `yargs`[12]. Konfiguračný súbor určený parametrom programu je spracovaný pomocou metódy `parseConfigurationFile()`. Konfiguračný súbor je definovaný vo formáte JSON a následne prevedený na objekt, ktorého typ je určený rozhraním `IGeneratorConfig`. Toto rozhranie definuje všetky prípustné hodnoty, ktoré môžu byť použité pre konfiguráciu nástroja (viď sekcia 4.3). Trieda `AppGenerator` očakáva prítomnosť informácie, ktorá určuje cestu k súboru s popisom rozhrania, pričom ten môže byť uložený lokálne alebo na vzdialenom serveri. Cesta môže byť určená dvoma spôsobmi: parametrom programu alebo položkou v konfiguračnom súbore. V prípade, že táto informácia existuje, trieda pokračuje v metóde `processOutputRegeneration()`. Táto metóda zabezpečuje kontrolu nad tým, či v špecifikácii API došlo k nejakej zmene a či je potrebné výsledný kód vygenerovať nanovo. Metóda implementuje postup popísaný vývojovým diagramom 4.4. Ak je generovanie kódu potrebné, potom pokračuje v metóde `parseAndGenerate()`. V nej, pomocou metódy `getParser()` skontroluje formát súboru so špecifikáciou rozhrania a na základe jeho typu vytvorí potrebnú inštanciu triedy `OpenapiParser` alebo `WadlParser`. Obidve triedy implementujú rozhranie `IParser` s metódou `getParsed()`, pomocou ktorej získa spracovaný popis rozhrania vo forme objektu typu `SpecificationObject`. Následne sa vytvorí inštancia triedy `Generator`, ktorej sa predáva tento objekt a volá sa jej metóda `generate()` pre vygenerovanie výstupu.

### OpenapiParser

Úlohou triedy `OpenapiParser` je analýza a spracovanie popisu rozhrania vytvoreného podľa štandardu OpenAPI a jeho prevod na objekt typu `SpecificationObject` (kód 4.1). Trieda implementuje rozhranie `IParsed` a preto poskytuje verejnú metódu `getParsed()` na získanie tohto objektu. V rámci metódy `getParsed()` sa ako prvé popis rozhrania transformuje z textového súboru na objekt jazyka JavaScript. Špecifikácia OpenAPI môže byť definovaná vo formátoch JSON alebo YAML, avšak jazyk TypeScript/JavaScript natívne podporuje prevod na objekt iba z textu vo formáte JSON. K transformácii textu v jazyku YAML na objekt jazyka JavaScript bola použitá knižnica `yaml`[13]. Na uľahčenie práce s objektom reprezentujúcim OpenAPI špecifikáciu bol tomuto objektu priradený typ `OpenAPIV3.Document`. Tento typ nebol implementovaný priamo v nástroji, ale bol prevzatý z knižnice `openapi-types`[14].

Po inicializácii objektu typu `SpecificationObject` prebieha jeho postupné napĺňanie hodnotami získanými z popisu rozhrania. Ako prvé sa v rámci metódy `parseServers()`

[11] https://docs.npmjs.com/files/package.json#main
[12] https://www.npmjs.com/package/yargs
[13] https://www.npmjs.com/package/yaml
[14] https://www.npmjs.com/package/openapi-types

spracujú URL adresy všetkých dostupných serverov. Metóda `parsePaths()` následne z objektu *Paths Object*[15] vyfiltruje všetky HTTP metódy a ich ďalšie spracovanie zabezpečuje funkcia `parseMethods()`. Táto funkcia pre každú HTTP metódu vytvorí objekt typu `MethodObject` (kód 4.6) a nakoniec všetky objekty spojí do zoznamu. Tento zoznam potom reprezentuje položku `methods` v rámci objektu `SpecificationObject`.

Pri spracovávaní jednotlivých metód sa taktiež doplňujú informácie o modeloch, komplexných typoch, výpočtových typoch a typoch parametrov metód. Spracovávanie schém[16], podľa ktorých vznikajú typy a dátové štruktúry primárne zabezpečuje rekurzívna metóda `parseSchemaObject()`. Ak schéma reprezentuje objekt, jej vlastnosti sa postupne spracovávajú rovnakou metódou `parseSchemaObject()`. Táto rekurzia končí v prípade, že aktuálne spracovávaná schéma nereprezentuje objekt, ktorý by obsahoval ďalšie vlastnosti, ale jej typ je jednoduchým typom[17] (*integer*, *number*, *boolean*, *string*).

Spracovanie bezpečnostných schém zabezpečuje metóda `getParsedSecurityObject()`, ktorej výstupom je objekt typu `SecurityObjects` (kód 4.8). Metóda sa používa na spracovanie bezpečnostných schém na úrovni celého API, ako aj na úrovni jednotlivých metód. Metóda `appendSecurityToMethods()` zabezpečí, že bezpečnostné schémy použité na úrovni celého API sa použijú pre každú metódu.

Špecifikácia OpenAPI umožňuje definovanie znovu-použiteľných komponent[18], na ktoré sa možno odkazovať pomocou objektu referencie[19]. Získavanie objektov, ktoré môžu byť definované referenciou zabezpečuje generická metóda `getComponentObject<T>()`, ktorá vracia objekt reprezentujúci komponentu daného typu `T`.

### WadlParser

Úlohou triedy `WadlParser` je analýza a spracovanie popisu rozhrania v jazyku WADL a jeho prevod na objekt typu `SpecificationObject`. Aj táto trieda implementuje rozhranie `IParsed` a poskytuje verejnú metódu `getParsed()` na získanie tohto objektu.

Keďže je jazyk WADL založený na XML, dôležitou otázkou pri spracovávaní WADL špecifikácie bolo určenie spôsobu jej reprezentácie v rámci programu. Možným riešením bolo jej transformovanie na objekt jazyka JavaScript. K tomuto účelu bol využitý nástroj `node-xml2js`[20], ktorý umožňuje konvertovať XML na objekt s využitím určitých možností konfigurácie[21]. Okrem implicitných konfiguračných možností som použil aj:

- `explicitArray: false` - Synovské uzly (XML elementy) sú vkladané do polí iba v prípade, ak je ich počet väčší ako 1. Inak sú uzly vytvorené ako samostatné objekty.
- `xmlns: true` - Každému objektu reprezentujúcemu XML element priradí položku `'$ns'`, ktorá obsahuje informácie o jeho lokálnom názve a URI menného priestoru (*namespace*).
- `tagNameProcessors: [xml2jsFunc.stripPrefix]` - Určenie funkcie na úpravu názvov XML značiek. Všetky prefixy názvov značiek (okrem prefixu `xmlns`) budú odstránené (napr. názov značky `<xs:sequence/>` bude prevedený na `'sequence'`).

[15] https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.3.md#pathsObject
[16] https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.3.md#schemaObject
[17] https://json-schema.org/understanding-json-schema/reference/type.html
[18] https://swagger.io/specification/#componentsObject
[19] https://swagger.io/specification/#referenceObject
[20] https://www.npmjs.com/package/xml2js
[21] https://www.npmjs.com/package/xml2js#options

Pre uľahčenie práce s objektom vytvoreným pomocou nástroja `node-xml2js` som využil pomocný nástroj `node-xml2js-xpath`[22]. Tento nástroj umožňuje dotazovanie nad vlastnosťami objektu za pomoci syntaxu jazyka XPath[23], avšak jeho nevýhodou je, že možnosti dotazovania sú obmedzené. Tento nástroj taktiež nepodporuje prácu s mennými priestormi. Pre zaistenie podpory dotazovania nad mennými priestormi bola implementovaná metóda `findInNamespace()`. Táto metóda využíva funkcionalitu nástroja `node-xml2js-xpath`, pomocou ktorého získava všetky vlastnosti objektu podľa daného XPath výrazu. Funkcia ešte prijíma parameter, ktorý určuje menný priestor. Výsledné vlastnosti potom redukuje na základe zhody informácií o mennom priestore, ktoré sú obsiahnuté v položke `'$ns'` s menným priestorom definovaným parametrom.

WadlParser získava informácie o dátových štruktúrach z XML Schema (XSD) súboru, na ktorý sa WADL špecifikácia odkazuje pomocou elementu `grammars`. Jeho spracovanie zabezpečuje metóda `parseXsdFile()`, ktorá vracia množinu modelov, komplexných a výpočtových typov. Všetky elementy zodpovedajúce komplexným typom[24] sú spracované pomocou metódy `parseComplexTypes()`. Spracovanie elementov[25], z ktorých sú vytvorené modely, zabezpečuje metóda `parseElements()` a jednoduché typy[26] sú spracované pomocou metódy `parseSimpleTypes()`.

### FileManager

Pre prácu so súbormi nástroj využíva triedu `FileManager`. Táto trieda reprezentuje samotný súbor a ponúka operácie, ktoré je možné nad súborom vykonať. Pri vytváraní inštancie triedy `FileManger` konštruktor očakáva definovanie cesty k súboru, ktorý sám reprezentuje a prípadne aj jeho dátový obsah. Definovanie dátového obsahuje je nepovinné, ale je používané v prípade vytvárania nového súboru pomocou metódy `writeFile()`. Táto metóda zapisuje definovaný dátový obsah do súboru určeného jeho cestou. Pri tom využíva nástroj `mkdirp`[27] k vytvoreniu potrebnej adresárovej štruktúry, ak dané adresáre ešte neexistujú.

Čítanie obsahu súboru zabezpečuje metóda `readFile()`. Ak je cesta k súboru určená pomocou web URI, jeho obsah je najprv získaný pomocou metódy `fetchFile()`. Na získavanie vzdialených súborov využíva metódu `get`[28] implementovanú v *Node.js HTTPS* module. Trieda ďalej implementuje metódy na kontrolu existencie súboru, vytvorenia kópie súboru a porovnávania obsahu s iným súborom, pričom k tomu využíva metódy z *Node.js File System* modulu[29]. Trieda ponúka aj ďalšie metódy pre získanie podrobných informácii o súbore, ktoré sprostredkováva za pomoci využitia metód z *Node.js Path* modulu[30].

### Generator

Trieda `Generator` zabezpečuje vygenerovanie súborov, ktoré obsahujú zdrojový kód klientskej aplikácie. Samotný výstupný kód je tvorený za pomoci šablónovacieho systému *Musta-*

[22] https://www.npmjs.com/package/xml2js-xpath
[23] https://www.w3.org/TR/1999/REC-xpath-19991116/
[24] https://www.w3.org/TR/xmlschema11-1/#Complex_Type_Definitions
[25] https://www.w3.org/TR/xmlschema11-1/#cElement_Declarations
[26] https://www.w3.org/TR/xmlschema11-1/#Simple_Type_Definitions
[27] https://www.npmjs.com/package/mkdirp
[28] https://nodejs.org/api/https.html#https_https_get_url_options_callback
[29] https://nodejs.org/api/fs.html
[30] https://nodejs.org/api/path.html

*che.js*. Objekt typu `SpecificationObject`, ktorého dáta budú vkladané namiesto značiek v šablónach táto trieda získava ako parameter konštruktora pri vytváraní jej inštancie.

Ešte pred samotným generovaním výstupných kódov, trieda zabezpečuje formátovanie názvov typov a dátových štruktúr, podľa definovaných možností v konfiguračnom súbore nástroja. Toto formátovanie zabezpečuje metóda `formatSpecification()`. Takto sformátovaný dátový objekt je následne pripravený na použitie šablónovacím systémom. Šablónovací systém je používaný v rámci metódy `renderFile()` (kód 5.1), ktorá očakáva dva parametre. Prvý parameter určuje cestu k šablóne a druhý parameter určuje cestu k výstupnému súboru. Všetky šablóny sú v rámci nástroja definované ako samostatné textové súbory. Metóda zabezpečí prečítanie šablóny a poskytne jej obsah spolu s dátovým objektom typu `SpecificationObject` funkcii `render()`[31] šablónovacieho systému *Mustache.js*. Táto funkcia zabezpečí, že definované značky v šablóne budú nahradené poskytnutými dátami, pričom značka bude nahradená dátovou položkou s rovnakým názvom jej kľúča. Spojené dáta reprezentujúce výsledný zdrojový kód klientskej aplikácie sú následne zapísané do súboru. Metódu `renderFile()` využíva trieda opakovane pre každú šablónu, ktorá reprezentuje jeden výstupný súbor.

```
1 private renderFile( templatePath: string, outputFile: string ): void {
2     new FileManager( templatePath ).readFile()
3         .then( template => {
4             const output = Mustache.render( template, this.specification );
5             new FileManager( this.generatorConfig.outputDir + '/' + outputFile,
6                              output ).writeFile();
7         } );
8 }
```

Výpis 5.1: Metóda využívajúca šablónovací systém *Mustache.js* a jeho metódu `render()`, ktorá zabezpečí nahradenie značiek v šablóne za poskytnuté dáta. Výsledok tejto metódy je zapísaný do súboru definovaného pomocou parametra metódy a ten je umiestnený do adresára určeného konfiguráciou nástroja.

### Definícia šablón

Šablóny nachádzajúce sa v adresári `src/templates/` sú definované ako textové súbory s príponou `.mustache`. Každá šablóna, ktorú nástroj využíva, reprezentuje jeden výstupný súbor. Jej obsah definuje štruktúru výstupného zdrojového kódu, ktorý bude používaný v rámci klientskej aplikácie.

Obsah šablón je tvorený staticky definovaným kódom a značkami, ktoré slúžia ako zástupné miesta pre dáta, pričom názov značky reprezentuje kľúč dátovej položky. Staticky definovaný kód je nemenný a vo výstupnom súbore sa objaví vždy. Výnimkou je statický kód alebo akýkoľvek iný obsah obklopený značkami, ktoré určujú začiatok a koniec sekcie[32] (značky so znakmi `#` a `/` pred názvom kľúča). Obsah definovaný v rámci sekcie môže byť zahrnutý do výstupu jeden alebo viackrát na základe aktuálnej hodnoty kľúča značky. V prípade, že sa kľúč sekcie v rámci dát nenachádza alebo je jeho hodnota `null`, `undefined`, `false`, `0` alebo `NaN`, potom obsah tejto sekcie nie je zahrnutý do výstupu. Presným opakom sú invertované sekcie, ktorých názvy kľúčov začínajú znakom `^` namiesto `#` a ich obsah je

[31] https://github.com/janl/mustache.js#usage
[32] https://github.com/janl/mustache.js#sections

do výstupu zahrnutý, ak sa kľúč sekcie v rámci dát nenachádza alebo je jeho hodnota `null`, `undefined`, `false`, `0` alebo `NaN`.

Kód 5.2 ukazuje časť šablóny, ktorá definuje konštantu s konfiguračnými nastaveniami HTTP transakcií. Využíva sekcie a invertované sekcie pre kľúče `hasRequestBody` a `hasSuccessResponse`, ktoré v tomto prípade nahradzujú logiku podmienky *if* a klauzuly *else*. Blok dát v sekcii ohraničenej značkami `{{#methods}}` a `{{\methods}}` sa vo výstupnom kóde objaví pre každú položku zoznamu `methods`.

```
export const methodConfig: MethodConfig = {
{{#methods}}    {{name}}: {
{{#hasRequestBody}}         consume: {{name}}.consume[0],
{{/hasRequestBody}}
{{^hasRequestBody}}         consume: null,
{{/hasRequestBody}}
{{#hasSuccessResponse}}         accept: {{name}}.accept[0],
{{/hasSuccessResponse}}
{{^hasSuccessResponse}}         accept: null,
{{/hasSuccessResponse}}
        reportProgress: false,
        withCredentials: false
    },

{{/methods}}
}
```

Výpis 5.2: Časť definície *Mustache.js* šablóny využívajúcej sekcie a invertované sekcie, pomocou ktorých definuje štruktúru objektu používaného pre konfiguráciu HTTP transakcií.

## 5.4 Štruktúra a popis výstupného kódu nástroja

Výstupom nástroja sú súbory obsahujúce zdrojový kód, ktorý je použiteľný v klientskej aplikácii využívajúcej framework Angular. Vygenerované súbory sú umiestnené do adresára určeného konfiguráciou nástroja. V tejto sekcii bude popísaná štruktúra vygenerovaného kódu z hľadiska frameworku Angular. Uvedené názvy súborov budú obsahovať variabilné časti, ktoré sú označené pomocou znakov `< >`. Aj tieto variabilné časti názvov sú vo výsledku nahradené hodnotami určenými konfiguráciou nástroja.

### Dátové štruktúry

Všetky vygenerované dátové štruktúry, ktoré budú používané v rámci služby sú umiestnené do súboru `models.ts`. Aj keď sú dátové štruktúry definované pomocou rozhraní, pre ich použitie v rámci služieb boli vytvorené aliasy. Typový alias[33] nevytvára nový typ, iba definovanému rozhraniu priraďuje nový názov. Využitie tohto prístupu bolo užitočné hlavne vzhľadom na spôsob definície XML Schema (ďalej ako XSD) elementov[34] a ich následné využitie vo WADL súbore. Kód 5.3 ukazuje príklad definície XSD elementu s názvom `userResponse` typu `user`, pričom typ `user` je komplexným typom. V rámci WADL súboru je potom možné na element `userResponse` odkazovať, čo je ukázané vo výpise 5.4.

[33] https://www.typescriptlang.org/docs/handbook/advanced-types.html#type-aliases
[34] https://www.w3.org/TR/xmlschema11-1/#td

Vygenerovaný kód dátových štruktúr reprezentujúci XSD element z výpisu 5.3 je ukázaný vo výpise 5.5. Komplexný typ `user` je reprezentovaný rozhraním `IUser`. Pre toto rozhranie je potom vytvorený typový alias `UserResponse` reprezentujúci XSD element `userResponse`.

```
<xs:element name="userResponse" type="user"/>
<xs:complexType name="user">
    <xs:sequence>
        <xs:element name="roles" type="role" nillable="true"/>
        <xs:element name="permissions" type="xs:string" nillable="true"/>
    </xs:sequence>
</xs:complexType>
<xs:simpleType name="role">
    <xs:restriction base="xs:string">
        <xs:enumeration value="USER"/>
        <xs:enumeration value="ADMIN"/>
    </xs:restriction>
</xs:simpleType>
```

Výpis 5.3: Príklad definície XSD elementu, ktorého typ je určený komplexným typom. Komplexný typ obsahuje vlastnosť, ktorá využíva definíciu jednoduchého typu.

```
<representation element="userResponse" mediaType="application/json"/>
```

Výpis 5.4: Definícia WADL reprezentácie pomocou odkazu na XSD element definovaný v príklade 5.3.

```
export type UserResponse = IUser;
export type Role = "USER" | "ADMIN" ;

export interface IUser {
    roles?: Role;
    permissions?: string;
}
```

Výpis 5.5: Vygenerované dátové štruktúry reprezentujúce XSD element z príkladu 5.3.

### Služba

Všetky metódy zabezpečujúce komunikáciu so serverom sú umiestnené v rámci služby `<serviceName>Service`, ktorá sa nachádza v súbore `<serviceName>.service.ts`. Táto služba na vykonávanie HTTP transakcií využíva metódy poskytované službou `HttpClient`, ktorej inštanciu získava pomocou systému vkladania závislostí (*dependency injection*, ďalej ako DI). Konfiguračné nastavenia metód, autentifikačných kľúčov a adresy pre prístup k serveru sú službe sprostredkované pomocou tokenov, ktoré taktiež získava pomocou DI. Každý z týchto tokenov nesie hodnotu typu `BehaviorSubject`[35], ktorý využíva princípy *Observable* popísané v sekcii 3.4. Služba sa prihlasuje k odberu všetkých tokenov, aby mohla získavať hodnoty, ktoré v sebe nesú a aby bola informovaná o každej zmene tejto hodnoty.

[35] https://www.learnrxjs.io/learn-rxjs/subjects/behaviorsubject

Trieda reprezentujúca službu `<serviceName>Service` dedí funkcionalitu od triedy s názvom `BaseService`, ktorá poskytuje pomocné metódy napríklad pre serializáciu parametrov alebo tela metódy. Samotná služba `<serviceName>Service` implementuje metódy pre prístup k serveru a spracovanie neúspešných odpovedí. Všetky metódy definované pre prístup k serveru využívajú princíp preťažovania metód[36]. Každá takáto metóda má vytvorené 4 definície. Tie sa od seba líšia sprostredkovaným parametrom `observe`, podľa ktorého je určený návratový typ metódy. Príklad definície hlavičiek metódy je uvedený vo výpise 5.6.

Všetky neúspešné odpovede od servera sú spracované pomocou metódy `handleError()`. Každá metóda využívajúca HTTP transakcie používa operátor `catchError` na zachytávanie neúspešných odpovedí. Tento operátor zabezpečí volanie metódy, ktorá obsahuje zoznam kódov neúspešných odpovedí a príslušných chybový hlásení pre konkrétnu metódu. Tento zoznam spolu so zachytenou chybou od servera ďalej posiela ako parameter metóde `handleError()`, ktorá na základe kódu chyby vypisuje príslušné chybové hlásenie.

```
1 getInventory( observe?: 'body' ): Observable<GetInventory>;
2 getInventory( observe?: 'response' ): Observable<HttpResponse<GetInventory>>;
3 getInventory( observe?: 'events' ): Observable<HttpEvent<GetInventory>>;
4 getInventory( observe: any = 'body' ): Observable<any>;
```

Výpis 5.6: Využívanie preťažovania metód pri implementácii metód pre prístup ku koncovým bodom servera. Hodnota parametera `observe` určuje návratový typ metódy.

### Interceptor

V rámci súboru `<interceptorName>.interceptor.ts` je implementovaný interceptor s názvom `<interceptorName>Interceptor`. Jeho úlohou je transformácia všetkých odchádzajúcich HTTP žiadostí, ktoré sú odoslané v rámci modulu `<moduleName>Modul`. Transformácia spočíva v serializácii tela HTTP žiadosti, na základe nastaveného MIME typu položky `Content-Type`[37] v HTTP hlavičke. Samotná serializácia je vykonávaná za pomoci funkcie `serializeBody()`[38], ktorá je taktiež súčasťou HTTP API modulu frameworku *Angular*.

### NgModul

Súčasťou vygenerovaného kódu je *NgModul* s názvom `<moduleName>Modul`, ktorého implementácia sa nachádza v súbore `<moduleName>.module.ts`. V rámci tohto súboru sú taktiež definované všetky *injection tokeny*[39] používané na konfiguráciu výstupného kódu. *NgModul* registruje poskytovateľov týchto tokenov a každému z nich priraďuje počiatočnú hodnotu. Pre zabezpečenie funkcionality interceptora `<interceptorName>Interceptor` sa v rámci modulu registruje poskytovateľ tokenu `HTTP_INTERCEPTORS`[40]. Ďalej je v rámci modulu registrovaný poskytovateľ vygenerovanej služby `<serviceName>Service`, ktorá zabezpečuje prístup k serveru. Táto služba využíva funkcionalitu služby `HttpClient` a preto je pre zabezpečenie jej funkcie nevyhnutné v rámci modulu importovanie modulu `HttpClientModule`.

[36] https://en.wikipedia.org/wiki/Function_overloading
[37] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type
[38] https://angular.io/api/common/http/HttpRequest#serializebody
[39] https://angular.io/api/core/InjectionToken
[40] https://angular.io/api/common/http/HTTP_INTERCEPTORS

### Konfiguračné možnosti

Posledným vygenerovaným súborom je súbor s názvom `configuration.ts`. V rámci neho sú definované objekty a konštanty, pomocou ktorých je možné konfigurovať metódy služby `<serviceName>Service`. Možnosti konfigurácie sú následovné:

1. **Konfigurácia HTTP žiadostí** - Konfiguračné možnosti HTTP transakcií sú definované v rámci objektu `methodConfig` typu `MethodConfig`. Tento objekt obsahuje jednu položku pre každú metódu, ktorá sa vyskytuje v službe `<serviceName>Service` a ktorá zároveň používa HTTP transakcie. Každá vlastnosť objektu je pomenovaná podľa názvu metódy a reprezentuje objekt s konfiguračnými možnosťami pre danú metódu, ktorými sú:

   - `consume` - Určenie MIME typu pre položku `Content-Type` v hlavičke HTTP žiadosti.
   - `acccept` - Určenie MIME typu pre položku `Accept`[41] v hlavičke HTTP žiadosti.
   - `reportProgress` - Získavanie podrobných informácií o stave HTTP žiadosti.
   - `withCredentials` - Určenie, či má byť HTTP žiadosť odoslaná spolu s identifikáciou odosielateľa.

   Popis rozhrania môže obsahovať definíciu viacerých reprezentácií tela žiadostí a odpovedí. Všetky použiteľné možnosti MIME typov reprezentácií pre danú metódu sú definované v samostatných objektoch v konfiguračnom súbore. Tieto objekty majú rovnaký názov ako metóda a obsahujú zoznam použiteľných možností pre položky `consume` a `accept`.

2. **Určenie API kľúčov** - Všetky API kľúče definované rozhraním sú uvedené ako samostatné položky objektu `apiKeys`. Na začiatku má každý kľúč priradenú hodnotu `null` a pre ich správnu funkciu je potrebné hodnoty API kľúčov nahradiť.

3. **Určenie tokenu pre autentifikáciu typu Bearer** - Konštanta `bearerToken` reprezentuje token, ktorý bude používaný pri HTTP autentifikácii typu *Bearer*.

4. **Určenie tokenu pre HTTP Basic autentifikáciu** - Konštanta `basicToken` reprezentuje token používaný pri *HTTP Basic* autentifikácii. Hodnota tokenu má byť vytvorená z identifikácie užívateľa a jeho hesla pomocou kódovania **Base64**[42].

5. **Určenie URL adresy servera** - Konštanta `apiEndpoint` určuje adresu servera, na ktorú budú odosielané HTTP žiadosti. Jej hodnota je určená položkou zo zoznamu `apiEndpoints`, ktorý obsahuje všetky použiteľné URL adresy. Počiatočná hodnota tejto konštanty je nastavená na prvú z definovaných URL adries.

Objekty a konštanty určené pre konfiguráciu sú zo súboru `configuration.ts` exportované a použité v rámci *NgModulu*. Ich hodnoty sú použité na inicializáciu `BehaviorSubject` objektov, ktoré uchovávajú aktuálne konfiguračné nastavenia. Nakoniec sa hodnoty týchto objektov používajú pri registrácii konfiguračných tokenov. Aj keď je počiatočná konfigurácie určená hodnotami zo súboru `configuration.ts`, používateľ má možnosť konfiguráciu programovo zmeniť. Stačí, že v rámci komponenty získa konfiguračný token a priradí mu novú hodnotu pomocou jeho metódy `next()`. Keďže služba je prihlásená k odberu všetkých tokenov, ich nové hodnoty budú v rámci služby automaticky používané.

[41] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept
[42] https://tools.ietf.org/html/rfc3548#section-3

## 5.5 Integrácia do repozitára NPM

Ako už bolo spomínané, jedným z hlavných cieľov tejto práce bolo vytvoriť nástroj ako samostatný balík, ktorý môže byť následne zverejnený. Výsledný nástroj bol preto publikovaný do verejného online NPM repozitára pod názvom *angular-rest-generator*[43]. Jeho publikovanie v NPM repozitári zabezpečuje dostupnosť všetkým užívateľom a uľahčuje jeho použitie v rámci klientskych aplikácií. Publikovanie do verejného repozitára bolo bezplatné a vyžadovalo iba registráciu nového účtu.

[43] https://www.npmjs.com/package/angular-rest-generator

# Kapitola 6

# Vyhodnotenie a testovanie nástroja

Táto kapitola sa zaoberá testovaním navrhnutého a implementovaného nástroja. V prvej časti je uvedený popis serverov a API špecifikácií, ktoré boli použité pri testovaní nástroja. Sekcia 6.2 sa zaoberá použitým spôsobom testovania vygenerovaného kódu a popisuje ukážkovú aplikáciu, ktorá bola vytvorená a použitá na testovanie. V sekcii 6.3 sú popísané vlastnosti API špecifikácií, ktoré nástroj podporuje. Záverečná sekcia obsahuje námety na možné vylepšenie nástroja.

## 6.1 Použité API špecifikácie a servery

Pre otestovanie nástroja bolo nevyhnutné použitie servera, ktorý ponúkal služby a zároveň ich popis podľa štandardu OpenAPI. Z tohto dôvodu som použil server *Petstore server*[1] spoločnosti Swagger, pričom jeho API špecifikáciu vytvorenú podľa štandardu OpenAPI verzie 3 som získal vo formátoch JSON a YAML z nástroja *Swagger Editor*[2].

Pre otestovanie generovania dátových štruktúr podľa WADL špecifikácie som využil popis rozhrania, ktorý som získal z testovacích súborov nástroja *rest-client-generator*[3]. Tento popis využíval reprezentácie definované pomocou XSD súboru, ktorý bol taktiež súčasťou testovacích súborov spomínaného nástroja[4].

## 6.2 Testovacia webová aplikácia

Pre otestovanie použiteľnosti vygenerovaného kódu bola vytvorená jednoduchá webová aplikácia. Cieľom vytvorenia tejto aplikácie bola demonštrácia funkčnosti vygenerovaného kódu v rámci klientskej aplikácie využívajúcej framework *Angular*. Táto aplikácia bola postavená na frameworku *Angular* verzie 9 a využíva CSS štýly definované pomocou frameworku *Bootstrap* verzie 4.4[5]. Pomocou tejto aplikácie je možné demonštrovať komunikáciu so serverom a to bez potreby implementácie volaní jednotlivých metód.

Jedinou úlohou aplikácie je volanie metód pre prístup k serveru a zobrazovanie odpovedí od servera. Súčasťou aplikácie je modul s názvom `GatewayModule`, ktorý bol vygenerovaný

[1] https://petstore.swagger.io/

[2] https://editor.swagger.io/?_ga=2.49032678.15493450.1588773244-181771021.1584950976#

[3] https://github.com/dzurikmiroslav/rest-client-generator/blob/master/test/application.wadl

[4] https://raw.githubusercontent.com/dzurikmiroslav/rest-client-generator/master/test/application.xsd

[5] https://getbootstrap.com/docs/4.4/getting-started/introduction/

pomocou nástroja *angular-rest-generator*. Všetky vygenerované súčasti tohto modulu vrátane služby `GatewayService`, ktorá poskytuje metódy pre prístup k serveru sú umiestnené v adresári `src\app\gateway`. Aplikácia importuje `GatewayModule` v rámci `AppModule` modulu, čím sprístupňuje jeho funkcionalitu celej aplikácii.

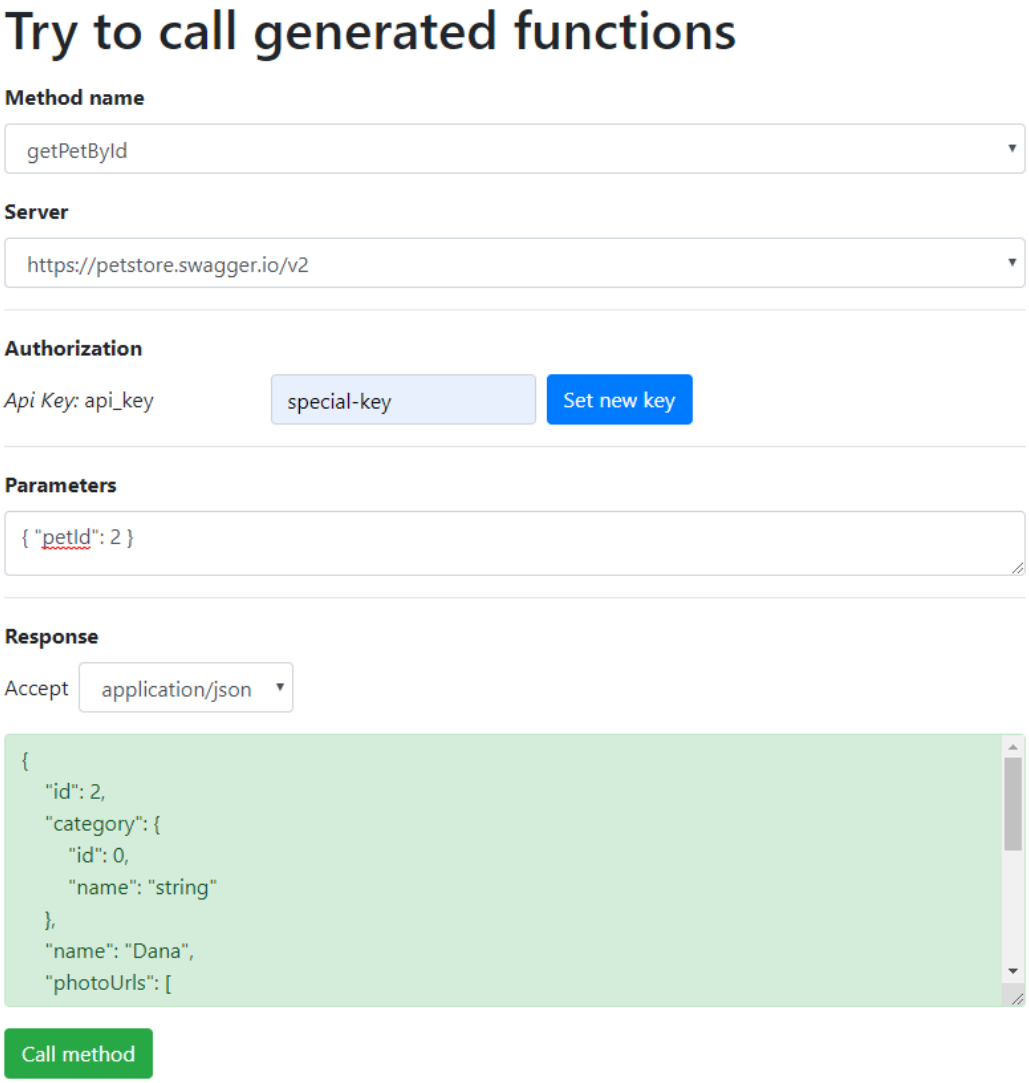
Základom aplikácie je formulár, pomocou ktorého je možné vykonávať volania metód. Tento formulár ponúka na výber všetky dostupné metódy služby `GatewayService` v rozbaľovacom zozname **Method**. Ďalej je možné vybrať zo zoznamu **Server** jednu z adries, na ktorých je server dostupný. Po výbere metódy sa zobrazia jej konfiguračné možnosti. Ak metóda používa niektorý z podporovaných spôsobov autentifikácie, potom sa v časti **Authorization** zobrazia používané tokeny a kľúče a ich aktuálne hodnoty. Tie je možné zmeniť prepísaním aktuálnej hodnoty a stlačením tlačidla **Set new token/key**. Ak metóda očakáva parametre, potom zobrazí textové pole **Parameters**, do ktorého je možné vložiť jej parametere vo formáte JSON. To isté platí pre telo metódy, pričom v časti **Body** je možné zvoliť jeho MIME typ. Po vyplnení všetkých potrebných polí je možné zavolať metódu tlačidlom **Call method**. Získané odpovede od servera sú zobrazované v časti **Response**, pričom úspešné odpovede majú zelený podklad a naopak neúspešné odpovede majú červený podklad. Na obrázku 6.1 je možné vidieť použitie testovacej aplikácie pri volaní metódy služby `GatewayService` a získanú úspešnú odpoveď od servera.

Pomocou testovacej aplikácie bolo možné overiť, že vygenerovaný kód je v rámci klientskej aplikácie funkčný a pripravený na použitie. Aplikácia je však schopná pomocou formulára overiť iba kód, ktorý bol vygenerovaný nástrojom pri použití konfiguračnej možnosti `"parametersAsObject": true`. V opačnom prípade bolo možné overiť fungovanie vygenerovaného kódu iba pomocou ručnej implementácie v aplikácii. Dôvodom bolo to, že pomocou programovacieho jazyka TypeScript nebolo možné zistiť typ parametrov, ktoré metóda očakáva. Pri použití objektu reprezentujúceho všetky parametre tento problém odpadá, pretože je poskytnutý v formáte JSON a spracovaný metódou `JSON.parse()`.

## 6.3 Vlastnosti výsledného nástroja a podpora štandardov

Požiadavky kladené na nástroj pri jeho návrhu (tab. 4.2) boli čiastočne splnené. Implementovaný nástroj umožňuje generovanie dátových štruktúr na základe špecifikácie rozhrania vytvorenej podľa štandardu OpenAPI alebo WADL. Pri použití WADL zároveň musí platiť, že reprezentácie schém sú vytvorené pomocou XML Schema definície a špecifikácia rozhrania sa na nich odkazuje. Nástroj taktiež podporuje generovanie služieb zabezpečujúcich komunikáciu so serverom, avšak tie je schopný generovať iba podľa OpenAPI špecifikácie. Bez ohľadu na použitý štandard, výsledný zdrojový kód je vytvorený podľa syntaxe jazyka TypeScript. Zároveň platí, že všetky vygenerované súčasti sú zamerané na použitie v rámci klientskej aplikácie využívajúcej framework *Angular* kompatibilný s jeho verziou 9.x. Implementovaný nástroj je ovládateľný iba pomocou príkazového riadku a bol publikovaný v NPM repozitári pod názvom *angular-rest-generator*. Zoznam výsledných vlastností nástroja sumarizuje tabuľka 6.1.

Keďže štandardy OpenAPI a WADL sú rozsiahle, z časového hľadiska som nebol schopný implementovať všetky ich súčasti. Zoznam podporovaných vlastností štandardov, ktoré je nástroj schopný spracovať sú uvedené v prílohe A.




Obr. 6.1: Testovanie použiteľnosti vygenerovaného kódu pomocou testovacej klientskej aplikácie využívajúcej framework *Angular*. Aplikácia zobrazuje formulár, pomocou ktorého je možné vykonať volania metód a zobrazovať získané odpovede od servera po vykonaní HTTP transakcie. Formulár taktiež zobrazuje dostupné konfiguračné možnosti každej metódy.

| | **Výstup** | | | | | | |
|---|---|---|---|---|---|---|---|
| | **DŠ** | **S** | **TS** | **Angular** | **NPM balík** | **GUI** | **CLI** |
| **OpenAPI** | ✓ | ✓ | ✓ | 8.x - 9.x | `angular-rest-generator` | ✗ | ✓ |
| **WADL** | ✓ | ✗ | ✓ | 8.x - 9.x | `angular-rest-generator` | ✗ | ✓ |

**DŠ** = Dátové štruktúry; **S** = Služby; **TS** = TypeScript;
**GUI** = Grafické užívateľské rozhranie; **CLI** = Príkazový riadok;

Tabuľka 6.1: Popis vlastností implementovaného nástroja.

## 6.4 Možné vylepšenia

Medzi možné vylepšenia nástroja do budúcnosti patrí v prvom rade odstránenie jeho aktuálnych nedostatkov. Ide hlavne o doplnenie podpory generovania služieb podľa popisu rozhrania v jazyku WADL a kompletizácia podpory všetkých súčastí štandardu OpenAPI. Ďalšie vylepšenia nástroja by mohli zahrňovať doplnenie konfiguračných možností, s ktorými by mal používateľ nástroja väčšiu kontrolu nad výslednou štruktúrou vygenerovaného kódu. Tieto možnosti by mohli zahrňovať podporu umiestnenia jednotlivých metód do samostatných služieb, prípadne iných celkov, napríklad menných priestorov. Taktiež by bolo vhodné doplniť možnosti špecifikácie verzie frameworku *Angular* a prípadne aj samotnú špecifikáciu verzie jazyka TypeScript. V neposlednom rade by mohol nástroj podporovať možnosti generovania výstupu bez využitia závislostí na knižniciach frameworku *Angular*. Príkladom môže byť využitie rozhrania `XMLHttpRequest`[6] namiesto služby `HttpClient`.

[6] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

# Kapitola 7

# Záver

Cieľom tejto práce bolo navrhnúť a implementovať nástroj na automatické generovanie kostry aplikácií v jazyku TypeScript na základe popisu REST rozhrania. Riešenie malo zahrňovať generovanie dátových štruktúr a služieb, pomocou ktorých by bolo možné pristupovať ku koncovým bodom servera. Riešenie sa malo zameriavať na popis REST rozhraní vytvorených podľa štandardu OpenAPI alebo v jazyku WADL a taktiež na aplikácie podporujúce jazyk TypeScript a využívajúce framework *Angular*.

V teoretickej časti tejto práce boli vysvetlené princípy webových služieb a aplikačných programových rozhraní. Podrobnejšie boli popísané rozhrania vytvorené podľa architektonického štýlu REST. Taktiež boli uvedené jazyky a technológie používané pre ich popis, so zameraním na štandard OpenAPI a jazyk WADL. Následne boli prezentované princípy tvorby moderných webových aplikácií a technológie využívané pri ich tvorbe. Táto časť sa zameriavala na aplikácie typu *single-page*, jazyk TypeScript a framework *Angular*.

Pred samotným vývojom nástroja boli prezentované existujúce riešenia, ktoré je možné v súčasnosti využiť pre generovanie kódu podľa popisu REST rozhrania. Boli analyzované vlastnosti týchto nástrojov, ich výhody a nevýhody a na záver boli nástroje navzájom porovnané. Dôležitou časťou tejto práce bolo stanovenie vlastností, ktoré by malo výsledné riešenie spĺňať. Na základe určených požiadaviek bol stanovený spôsob generovania výstupného kódu a vytvorený návrh nástroja. Pre implementáciu nástroja bol zvolený jazyk TypeScript a jeho využitie v rámci prostredia *Node.js*. Pre tvorbu obsahu výstupných súborov bol zvolený šablónovací systém *Mustache.js*.

Výsledkom tejto práce je nástroj, ktorý umožňuje generovanie dátových štruktúr na základe popisu rozhrania v OpenAPI a WADL. Ďalej umožňuje generovanie služieb zabezpečujúcich komunikáciu so serverom na základe popisu rozhrania v OpenAPI. Nástroj bol vytvorený ako samostatný modul a bol publikovaný vo verejnom NPM repozitári pod názvom *angular-rest-generator*. Nástroj ponúka niekoľko možností konfigurácie a je použiteľný v rámci klientskych aplikácií využívajúcich framework *Angular*.

Pre otestovanie nástroja bola vytvorená samostatná testovacia aplikácia, pomocou ktorej bolo overené, že zdrojový kód vygenerovaný nástrojom *angular-rest-generator* je funkčný a jednoducho použiteľný.

Hlavým prínosom tohto nástroja je uľahčenie a urýchlenie procesu programovania klientskych aplikácií, ktoré na komunikáciu so serverom využívajú REST rozhranie. Do budúcna sa plánuje rozšírenie podpory štandardu OpenAPI a taktiež zavedenie podpory generovania služieb na základe popisu v jazyku WADL.

# Literatúra


[1] Angular. *Architecture overview* [online]. Version 8.2.15-local+sha.72c3ebe682. Google, 2019 [cit. 2019-12-31]. Dostupné z: https://angular.io/guide/architecture.

[2] Angular. *Workspace npm dependencies* [online]. Version 8.2.15-local+sha.72c3ebe682. Google, 2019 [cit. 2020-04-25]. Dostupné z: https://angular.io/guide/npm-packages.

[3] Angular. *Communicating with backend services using HTTP* [online]. Version 9.1.4-local+sha.c440165384. Google, 2020 [cit. 2020-04-29]. Dostupné z: https://angular.io/guide/http#communicating-with-backend-services-using-http.

[4] Angular. *Introduction to modules* [online]. Version 9.1.4-local+sha.c440165384. Google, 2020 [cit. 2020-04-29]. Dostupné z: https://angular.io/guide/architecture-modules.

[5] Angular. *Introduction to services and dependency injection* [online]. Version 9.1.4-local+sha.c440165384. Google, 2020 [cit. 2020-04-29]. Dostupné z: https://angular.io/guide/architecture-services.

[6] Berlind, D. APIs Are Like User Interfaces–Just With Different Users in Mind. *ProgrammableWeb.com* [online]. December 2015 [cit. 2019-12-21]. Dostupné z: https://www.programmableweb.com/news/apis-are-user-interfaces-just-different-users-mind/analysis/2015/12/03.

[7] Cheng, W., Schubert, J., Bornet, C. et al. *Swagger Codegen Fork: Q&A* [online]. SmartBear, január 2019 [cit. 2020-1-9]. Dostupné z: https://github.com/OpenAPITools/openapi-generator/blob/master/docs/qna.md.

[8] Cheng, W., Schubert, J., Bresson, J. et al. *Migrating from Swagger Codegen* [online]. SmartBear, august 2019 [cit. 2020-1-9]. Dostupné z: https://github.com/OpenAPITools/openapi-generator/blob/master/docs/migration-from-swagger-codegen.md.

[9] Emmit, A. a Scott, J. *SPA Design and Architecture: Understanding single-page web applications*. 1. vyd. Manning, 2015. ISBN 978-1-6172-9243-9.

[10] Fielding, R. T. REST APIs must be hypertext-driven. *Roy.Gbiv.com* [online]. Október 2008 [cit. 2020-05-18]. Dostupné z: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[11] Hadley, M. *Web Application Description Language* [online]. W3C, august 2009 [cit. 2019-12-25]. Dostupné z: https://www.w3.org/Submission/wadl/.

[12] Hombergs, T. REST with Hypermedia - Hot or Not? *Reflectoring.io* [online]. 2019 [cit. 2019-12-22]. Dostupné z: https://reflectoring.io/rest-hypermedia/.

[13] Jin, B., Sahni, S. a Shevat, A. *Designing Web APIs*. First edition. Sebastopol: O'Reilly, 2018. ISBN 978-1-492-02692-1.

[14] Linux Foundation. New Collaborative Project to Extend Swagger Specification for Building Connected Applications and Services. *Web.Archive.org* [online]. November 2015 [cit. 2019-12-28]. Dostupné z: https://web.archive.org/web/20160427104213/http://www.linuxfoundation.org/news-media/announcements/2015/11/new-collaborative-project-extend-swagger-specification-building.

[15] Massé, M. *REST API Design Rulebook*. First edition. Sebastopol: O'Reilly, 2012. ISBN 978-1-449-31050-9.

[16] Mikowski, M. S. a Powell, J. C. *Single Page Web Applications: JavaScript end-to-end*. 1. vyd. Manning, 2013. ISBN 978-1-6172-9075-6.

[17] Miller, D., Whitlock, J., Gardiner, M. et al. *OpenAPI Specification* [online]. Version 3.0.2. OpenAPI Initiative, október 2018 [cit. 2019-12-27]. Dostupné z: http://spec.openapis.org/oas/v3.0.2.

[18] Nderground. Java Template Engines. *Hackernoon.com* [online]. December 2018 [cit. 2019-01-13]. Dostupné z: https://hackernoon.com/java-template-engines-ef84cb1025a4.

[19] Neoteric. Single-page application vs. multiple-page application. *Medium.com* [online]. December 2016 [cit. 2019-12-29]. Dostupné z: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58.

[20] Newcomer, E. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. 1. vyd. Indianapolis: AddisonWesley Professional, máj 2002 [cit. 2020-05-13]. ISBN 978-0201750812.

[21] Node.js. *What is npm?* [online]. OpenJS Foundation, august 2011 [cit. 2020-04-25]. Dostupné z: https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/.

[22] Osmani, A. *Learning JavaScript Design Patterns*. First edition. Sebastopol: O'Reilly Media, august 2012 [cit. 2020-04-29]. ISBN 978-1-449-33181-8.

[23] Pasquali, S. a Faaborg, K. *Mastering Node.js*. Second Edition. Birmingram: Packt, december 2017 [cit. 2020-04-27]. ISBN 978-1-78588-896-0.

[24] Ponelat, J. *Designing APIs with Swagger and OpenAPI* [online]. MEAP VERSION 3. Manning, 2019 [cit. 2019-01-13]. Dostupné z: https://livebook.manning.com/book/designing-apis-with-swagger-and-openapi/meap-version-3/v-3/.

[25] Richardson, L. *RESTful web services*. First edition. Sebastopol: O'Reilly, 2007. ISBN 978-0-596-52926-0.

[26] Richardson, L. Act Three: The Maturity Heuristic. *Crummy.com* [online]. Január 2009 [cit. 2019-12-22]. Dostupné z: https://www.crummy.com/writing/speaking/2008-QCon/act3.html.

[27] Richardson, L. *RESTful Web APIs*. 1. vyd. Sebastopol: O'Reilly, 2013. ISBN 978-1-4493-5806-8.

[28] Rozentals, N. *Mastering TypeScript*. Second edition. Birmingram: Packt, február 2017 [cit. 2020-04-27]. ISBN 978-1-78646-871-0.

[29] RxJS. *Observable* [online]. Version 6.5.5-local+sha.7e4589a1. 2019 [cit. 2020-05-25]. Dostupné z: https://rxjs-dev.firebaseapp.com/guide/observable.

[30] Smartbear. *Swagger Editor* [online]. 2020 [cit. 2020-05-20]. Dostupné z: https://editor.swagger.io/.

[31] Smith, S. *Architect Modern Web Applications with ASP.NET Core and Azure* [online]. v2.2. Washington: DevDiv, .NET and Visual Studio product teams, 2019 [cit. 2019-12-28]. Dostupné z: https://dotnet.microsoft.com/download/e-book/aspnet/pdf.

[32] Suter, R. *NSwagStudio* [online]. September 2015, revidované 23.9.2017 [cit. 2020-05-25]. Dostupné z: https://github.com/RicoSuter/NSwag/wiki/NSwagStudio.

[33] Swagger. *What Is Swagger?* [online]. SmartBear, 2019 [cit. 2019-12-27]. Dostupné z: https://swagger.io/docs/specification/about/.

[34] TypeScript. *Interfaces* [online]. Microsoft, 2020 [cit. 2020-04-29]. Dostupné z: https://www.typescriptlang.org/docs/handbook/interfaces.html.

# Príloha A

# Podporované vlastnosti štandardov

## A.1 OpenAPI

Všetky uvedené vlastnosti vychádzajú zo štandardu OpenAPI verzie 3[1].

### Dátové typy

| Dátový typ | Formát | Výsledný dátový typ |
|---|---|---|
| integer | int32 | number |
| integer | int64 | number |
| number | float | number |
| number | double | number |
| string | | string |
| string | byte | string |
| string | binary | Blob |
| boolean | | boolean |
| string | date | Date |
| string | date-time | Date |
| string | password | string |

Tabuľka A.1: Dátové typy a ich formáty používané v rámci štandardu OpenAPI a ich odpovedajúce dátové typy v jazyku TypeScript.

### Podporované vlastnosti objektov

Objekt OpenAPI (tab. A.2) tvorí základ OpenAPI špecifikácie.

| Názov vlastnosti | Typ |
|---|---|
| servers | [Server Object] (tab. A.3) |
| paths | Paths Object (tab. A.5) |
| components | Components Object (tab. A.4) |
| security | [Security Requirement Object] (tab. A.16) |

Tabuľka A.2: Podporované vlastnosti objektu *OpenAPI Object*

[1] Http://spec.openapis.org/oas/v3.0.3

| Názov vlastnosti | Typ |
|---|---|
| url | string |

Tabuľka A.3: Podporované vlastnosti objektu *Server Object*.

| Názov vlastnosti | Typ |
|---|---|
| schemas | Map[string, Schema Object (tab. A.14) \| Reference Object (tab. A.13)] |
| responses | Map[string, Response Object (tab. A.12) \| Reference Object (tab. A.13)] |
| parameters | Map[string, Parameter Object (tab. A.8) \| Reference Object (tab. A.13)] |
| requestBodies | Map[string, Request Body Object (tab. A.9) \| Reference Object (tab. A.13)] |
| securitySchemes | Map[string, Security Scheme Object (tab. A.15) \| Reference Object (tab. A.13)] |

Tabuľka A.4: Podporované vlastnosti objektu *Components Object*.

| Názov vlastnosti | Typ |
|---|---|
| /path | Path Item Object (tab. A.6) |

Tabuľka A.5: Podporované vlastnosti objektu *Paths Object*.

| Názov vlastnosti | Typ |
|---|---|
| $ref | string |
| get | Operation Object (tab. A.7) |
| put | Operation Object (tab. A.7) |
| post | Operation Object (tab. A.7) |
| delete | Operation Object (tab. A.7) |
| options | Operation Object (tab. A.7) |
| head | Operation Object (tab. A.7) |
| patch | Operation Object (tab. A.7) |

Tabuľka A.6: Podporované vlastnosti objektu *Path Item Object*.

| Názov vlastnosti | Typ |
|---|---|
| operationId | string |
| parameters | [Parameter Object (tab. A.8)<br>\| Reference Object (tab. A.13)] |
| requestBody | Request Body Object (tab. A.9)<br>\| Reference Object (tab. A.13) |
| responses | Responses Object (tab. A.11) |
| security | [Security Requirement Object] (tab. A.16) |

Tabuľka A.7: Podporované vlastnosti objektu *Operation Object*.

| Názov vlastnosti | Typ |
|---|---|
| name | string |
| in | string |
| required | boolean |
| schema | Schema Object (tab. A.14)<br>\| Reference Object (tab. A.13) |

Tabuľka A.8: Podporované vlastnosti objektu *Parameter Object*.

| Názov vlastnosti | Typ |
|---|---|
| content | Map[string, Media Type Object (tab. A.10)] |
| required | boolean |

Tabuľka A.9: Podporované vlastnosti objektu *Request Body Object*.

| Názov vlastnosti | Typ |
|---|---|
| schema | Schema Object (tab. A.14)<br>\| Reference Object (tab. A.13) |

Tabuľka A.10: Podporované vlastnosti objektu *Media Type Object*.

| Názov vlastnosti | Typ |
|---|---|
| default | Response Object (tab. A.12)<br>\| Reference Object (tab. A.13) |
| HTP Status Code | Response Object (tab. A.12)<br>\| Reference Object (tab. A.13) |

Tabuľka A.11: Podporované vlastnosti objektu *Responses Object*.

| Názov vlastnosti | Typ |
|---|---|
| description | string |
| content | Map[string, Media Type Object (tab. A.10)] |

Tabuľka A.12: Podporované vlastnosti objektu *Response Object*.

| Názov vlastnosti | Typ |
|---|---|
| $ref | string |

Tabuľka A.13: Podporované vlastnosti objektu *Reference Object*.

| Názov vlastnosti | Typ |
|---|---|
| required | boolean |
| enum | [string] |
| type | string |
| items | Schema Object (tab. A.14) |
| properties | Schema Object (tab. A.14) |
| additionalProperties | boolean \| Schema Object (tab. A.14) |
| format | string |
| nullable | boolean |

Tabuľka A.14: Podporované vlastnosti objektu *Schema Object*.

| Názov vlastnosti | Typ |
|---|---|
| type | string |
| name | string |
| in | string |
| scheme | string |
| bearerFormat | string |

Tabuľka A.15: Podporované vlastnosti objektu *Security Scheme Object*.

| Názov vlastnosti | Typ |
|---|---|
| name | [string] |

Tabuľka A.16: Podporované vlastnosti objektu *Security Requirement Object*.

## A.2 WADL

Element *application* (tab. A.17) tvorí základ WADL popisu[2].

| Názov elementu |
|---|
| grammars (tab. A.18) |

Tabuľka A.17: Podporované elementy v rámci elementu *application*.

| Názov elementu |
|---|
| include (tab. A.19) |

Tabuľka A.18: Podporované elementy v rámci elementu *grammars*.

| Názov atribútu | Typ |
|---|---|
| href | xsd:anyURI |

Tabuľka A.19: Podporované atribúty v rámci elementu *include*.

[2] https://www.w3.org/Submission/wadl/

## A.3 XML Schema

Všetky uvedené dátové typy a štruktúry vychádzajú zo štandardu XML Schema 1.1[3] (ďalej ako XSD).

### Dátové typy

| Dátový typ XSD | Výsledný dátový typ |
|---|---|
| language | string |
| normalizedString | string |
| string | string |
| token | string |
| date | Date |
| dateTime | Date |
| duration | Date |
| gDay | Date |
| gMonth | Date |
| gMonthDay | Date |
| gYear | Date |
| gYearMonth | Date |
| time | Date |
| byte | number |
| decimal | number |
| int | number |
| integer | number |
| long | number |
| negativeInteger | number |
| nonNegativeInteger | number |
| nonPositiveInteger | number |
| positiveInteger | number |
| short | number |
| unsignedLong | number |
| unsignedInt | number |
| unsignedShort | number |
| unsignedByte | number |
| boolean | boolean |
| iné | Object |

Tabuľka A.20: Podporované dátové typy XML Schema definície a ich odpovedajúce dátové typy v jazyku TypeScript.

### Podporované XML Schema elementy

Element *schema* (tab. A.21) je koreňovým elementom každej XML Schema definície.

[3] https://www.w3.org/XML/Schema

| Názov elementu |
|---|
| element (tab. A.22) |
| complexType (tab. A.23) |
| simpleType (tab. A.24) |

| Názov atribútu | Typ |
|---|---|
| xmlns | xsd:anyURI |

Tabuľka A.21: Podporované elementy a atribúty v rámci elementu *schema*.

| Názov elementu |
|---|
| complexType (tab. A.23) |

| Názov atribútu | Typ |
|---|---|
| name | string |
| type | XSD typ (tab. A.20)<br>\| ComplexType name<br>\| SimpleType name |
| nillable | boolean |

Tabuľka A.22: Podporované elementy a atribúty v rámci elementu *element*.

| Názov elementu |
|---|
| sequence (tab. A.25) |
| complexContent (tab. A.26) |

| Názov atribútu | Typ |
|---|---|
| name | string |

Tabuľka A.23: Podporované elementy a atribúty v rámci elementu *complexType*.

| Názov elementu |
|---|
| restriction (tab. A.28) |

| Názov atribútu | Typ |
|---|---|
| name | string |

Tabuľka A.24: Podporované elementy a atribúty v rámci elementu *simpleType*.

| Názov elementu |
|---|
| element (tab. A.22) |

| Názov atribútu | Typ |
|---|---|
| - | |

Tabuľka A.25: Podporované elementy a atribúty v rámci elementu *sequence*.

| Názov elementu |
|---|
| extension (tab. A.27) |
| restriction (tab. A.28) |

| Názov atribútu | Typ |
|---|---|
| - | |

Tabuľka A.26: Podporované elementy a atribúty v rámci elementu *complexContent*.

| Názov elementu |
|---|
| sequence (tab. A.25) |

| Názov atribútu | Typ |
|---|---|
| base | XSD typ (tab. A.20)<br>\| ComplexType name<br>\| SimpleType name |

Tabuľka A.27: Podporované elementy a atribúty v rámci elementu *extension*.

| Názov elementu |
|---|
| sequence (tab. A.25) |
| enumeration (tab. A.29) |

| Názov atribútu | Typ |
|---|---|
| base | XSD typ (tab. A.20) |

Tabuľka A.28: Podporované elementy a atribúty v rámci elementu *restriction*.

| Názov elementu |
|---|
| - |

| Názov atribútu | Typ |
|---|---|
| value | string |

Tabuľka A.29: Podporované elementy a atribúty v rámci elementu *enumeration*.

# Príloha B

# Návod na použitie nástroja angular-rest-generator

Nástroj `angular-rest-generator` je možné využiť v rámci klientskej aplikácie využívajúcej framework Angular následovne:

1. Inštalácia nástroja v koreňovom adresári klientskej aplikácie pomocou NPM CLI, zadaním príkazu:
   - `npm install angular-rest-generator`
2. Vytvorenie konfiguračného súboru vo formáte JSON s definovanými vlastnosťami podľa tabuľky 4.3 (nepovinné, ak je nástroj spustený s parametrom `-apispec`).
3. Vygenerovanie cieľového kódu podľa špecifikácie rozhrania spustením nástroja z príkazového riadku:
   (a) Špecifikácia rozhrania definovaná parametrom (nepovinný konfiguračný súbor):
      - `angular-rest-generator -apispec ./apispecification.json`
      - `angular-rest-generator -apispec ./apispecification.json -config ./generatorconfig.json`
      - `angular-rest-generator -apispec ./apispecification.json -config ./generatorconfig.json -f`

   (b) Špecifikácia rozhrania definovaná parametrom `apiSpec` v konfiguračnom súbore:
      - `angular-rest-generator -config ./generatorconfig.json`
      - `angular-rest-generator -config ./generatorconfig.json -f`
4. Importovanie vygenerovaného modulu `<moduleName>Module`:
   (a) Do modulu, v rámci ktorého bude využívaná služba `<serviceName>Service`. Služba bude prístupná iba v tomto module.
   (b) Do koreňového modulu (zvyčajne `AppModule`). Služba bude prístupná v rámci celej aplikácie.
5. Vloženie služby `<serviceName>Service` do komponenty pomocou vkladania závislostí (*dependency injection*).
6. Volanie metód ktoré služba `<serviceName>Service` implementuje.

# Príloha C

# Návod na použitie testovacej aplikácie

Testovaciu aplikáciu, ktorá využíva služby vygenerované pomocou nástroja `angular-rest-generator` je možné otestovať následovne:

1. Stiahnutie testovacej aplikácie (napríklad z git repozitára[1]).
2. Inštalácia NPM modulov zadaním príkazu v koreňovom adresári aplikácie:
   - `npm install`
3. Spustenie testovacej aplikácie zadaním jedného z možných príkazov v koreňovom adresári aplikácie:
   - `ng serve -open`
   - `npm run start-open`
4. Volanie metód pomocou formulára vo webovom prehliadači.

[1] https://github.com/SilvesterLipjanec/example-app-angular-rest-generator.git

# Príloha D

# Obsah priloženého pamäťového média

Priložené pamäťové médium obsahuje nasledujúce adresáre:

- **tool** - Zdrojové kódy implementovaného nástroja.
- **example-app** - Zdrojové kódy testovacej webovej aplikácie.
- **latex** - Zdrojové kódy technickej správy.
- **thesis** - Obsahuje technickú správu vo formáte PDF.