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

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
FACULTY OF INFORMATION TECHNOLOGY

**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**
DEPARTMENT OF INFORMATION SYSTEMS


# GENEROVÁNÍ KLIENTSKÝCH APLIKACÍ V TYPESCRIPTU

TYPESCRIPT CLIENT APPLICATION GENERATION

**BAKALÁŘSKÁ PRÁCE**
BACHELOR'S THESIS


**AUTOR PRÁCE** **SAMUEL OBUCH**
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 bakalářské práce

Student: **Obuch Samuel**
Program: Informační technologie
Název: **Generování klientských aplikací v TypeScriptu**
**TypeScript Client Application Generation**
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

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: 28. května 2020
Datum schválení: 16. října 2019

## Abstrakt

Cieľom tejto bakalárskej práce je návrh a vývoj nástroja pre zjednodušenie a zefektívnenie vývoja webových aplikácií. Výsledné riešenie automaticky generuje základnú štruktúru webovej aplikácie z dokumentácie popisujúcej aplikačné rozhranie serveru. Nástroj podporuje dokumentačné štandardy OpenAPI a WADL. Riešenie umožňuje generovať základnú štruktúru pre framework Angular alebo knižnicu Axios.



## Abstract

The aim of this bachelor thesis is to design and develop a tool for simplifying and enhancing the development of web applications. The resulting solution automatically generates the basic web application structure from the documentation of the server's application interface. The tool supports the OpenAPI and WADL documentation standards. It enables the generation of the basic structure for the Angular framework or Axios library.



## Kľúčové slová

TypeScript, REST API, OpenAPI, WADL, Angular, Axios, HTTP klient, generátor, aplikačné rozhranie, webové aplikácie

## Keywords

TypeScript, REST API, OpenAPI, WADL, Angular, Axios, HTTP client, generator, application interface, web applications



## Citácia

OBUCH, Samuel. *Generování klientských aplikací v TypeScriptu.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Burget, Ph.D.

# Generování klientských aplikací v TypeScriptu

## Prehlásenie

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

. . . . . . . . . . . . . . . . . . . . . . .
Samuel Obuch
4. júna 2020

## Poďakovanie

Rád by som sa poďakoval vedúcemu práce, pánovi Ing. Radkovi Burgetovi, Ph.D. za cenné rady, konzultácie a trpezlivosť pri vedení mojej bakalárskej práce.

# Obsah


**1 Úvod** **2**

**2 Vývoj webových aplikácií** **3**
2.1 Architektúra webových aplikácií . . . 3
2.2 Princípy a webové technológie . . . 6
2.3 Existujúce a podobné riešenia . . . 12

**3 Návrh riešenia** **13**
3.1 Analýza požiadaviek . . . 13
3.2 Architektúra nástroja . . . 15
3.3 Návrh štandardizovaného rozhrania . . . 15
3.4 Výber technológií pre vývoj . . . 20

**4 Implementácia riešenia** **22**
4.1 Štruktúra a rozdelenie zdrojových textov riešenia . . . 22
4.2 Spracovanie vstupných argumentov . . . 23
4.3 Systém analýzy vstupného súboru s dokumentáciou API . . . 24
4.4 Generovanie výstupných súborov a použitie šablón . . . 27

**5 Testovanie a nasadenie nástroja** **31**
5.1 Testovanie . . . 31
5.2 Nasadenie nástroja . . . 33

**6 Záver** **34**

**Literatúra** **35**

**A Návod na inštaláciu nástroja** **37**
A.1 Globálna inštalácia . . . 37
A.2 Lokálna inštalácia . . . 37
A.3 Možnosti spustenia nástroja . . . 38
A.4 Vygenerovaná adresárová štruktúra . . . 38

**B Obsah priloženého CD** **39**

# Kapitola 1

# Úvod

Takmer v každej oblasti ľudského života sa snažíme nájsť spôsob, ako by sa dal čas a energia potrebné na dokončenie rôznych úloh čo najlepšie zefektívniť a celý proces čo najviac uľahčiť. Dnešné webové aplikácie sú síce užívateľsky stále prívetivejšie, ale na pozadí sú oveľa viac komplexnejšie. Vývoj takejto aplikácie je pomerne zdĺhavý a náročný proces. Je teda snahou tento proces uľahčiť, aby vytváranie základných prvkov webovej aplikácie bolo možné čo najviac zautomatizovať a programátor sa mohol sústrediť na dôležitejšie časti aplikácie.

V súčasnej dobe prebieha vývoj aplikačného rozhrania tak, že programátor dostane v nejakej forme dokumentáciu aplikačného rozhrania serveru a následne podľa tejto dokumentácie vytvorí zodpovedajúce rozhranie v aplikácii. V prípade, že sa aplikačné rozhranie serveru zmení, čo sa vo fáze vytvárania novej aplikácie deje pomerne často, musí programátor opäť upraviť rozhranie v aplikácii.

Cieľom tejto bakalárskej práce je návrh a implementácia nástroja **client-services-generator**, ktorý automatizuje proces vytvárania aplikačného rozhrania medzi aplikáciou a serverom. Výsledný nástroj generuje z dokumentácie aplikačného rozhrania serveru všetky základné časti rozhrania a dátové štruktúry, ktoré sú v nej popísané. Výsledný nástroj sa nachádza v najväčšom úložisku balíčkov pre vývoj webových aplikácií a je dostupný na adrese `https://www.npmjs.com/package/client-services-generator`. Nástroj je tak dostupný veľkému počtu vývojárov, zároveň je jeho použitie jednoduché a zapadá pohodlne do procesu vývoja webovej aplikácie. Programátor jednoducho nastaví cestu k súboru obsahujúcemu dokumentáciu aplikačného rozhrania serveru, zvolí si typ výstupu a nastaví adresár, kde majú byť výsledné súbory vygenerované.

Táto práca sa v nasledujúcej kapitole 2 zaoberá súčasným stavom vývoja webových aplikácií, technológií používaných pri vývoji a prehľadom existujúcich riešení. Kapitola 3 obsahuje analýzu požiadaviek na výstupné riešenie, návrh riešenia súčasného stavu a výber technológií, pomocou ktorých bude navrhnuté riešenie vytvorené. V kapitole 4 je popísaná implementácia, štruktúra výsledného riešenia a princíp generovania výsledných súborov. Nasledujúca kapitola 5 obsahuje popis testovania nástroja a jeho nasadenie do produkcie. Na záver, kapitola 6 obsahuje zhodnotenie dosiahnutých výsledkov a možnosti ďalšieho rozšírenia nástroja.

# Kapitola 2

# Vývoj webových aplikácií

Výsledkom tejto bakalárskej práce má byť nástroj, ktorý bude generovať základnú kostru webových aplikácií. Preto je potrebné ujasniť si princípy, návrhy a technológie, ktoré sa pri tvorbe webových aplikácií používajú. Pre túto prácu je najdôležitejšia tá časť webovej aplikácie, ktorá popisuje rozhranie medzi webovou aplikáciou a serverom.

Táto kapitola bude najprv popisovať prístupy k architektúre webovej aplikácie. Ďalej budú priblížené najpoužívanejšie technológie a princípy využívané pri tvorbe webových aplikácií. Na záver budú zhodnotené existujúce riešenia, ktoré sa zameriavajú na problematiku automatizácie vytvárania kostry webovej aplikácie.

## 2.1 Architektúra webových aplikácií

Webová aplikácia je komplexný systém skladajúci sa z veľkého počtu prvkov ako užívateľské rozhranie, databáza, server a veľa ďalších. Každý tento prvok spadá do jednej z dvoch základných častí každej webovej aplikácie, a to **front-end** alebo **back-end**. Front-end je klientská časť aplikácie, ktorá sa zobrazuje priamo u užívateľa vo webovom prehliadači. Back-end je časť aplikácie, ktorá sa nachádza na serveri a klient k nej pristupuje len pomocou HTTP[1] požiadaviek.

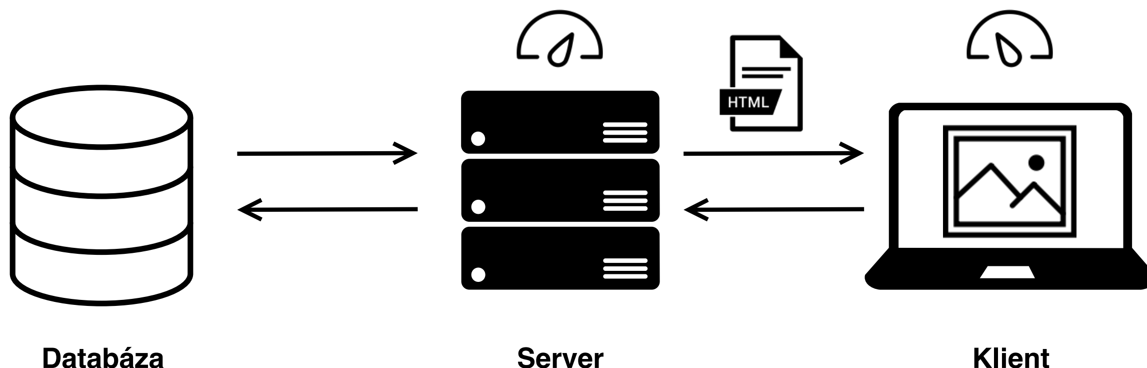
Z pohľadu rozhrania medzi klientskou aplikáciou a serverom existujú dva najznámejšie prístupy: **server side rendering** (skrátene SSR) a **client side rendering** (skrátene CSR) [2]. Oba prístupy majú svoje výhody i nevýhody a každý sa hodí na iné použitie. Najprv bude priblížený princíp SSR, pretože je starší a na základe niektorých jeho nevýhod vznikol CSR.

### 2.1.1 Server side rendering

Princíp fungovania SSR je nasledovný. Užívateľ odošle požiadavku na zobrazenie webovej stránky na server. Server následne vygeneruje HTML dokument s obsahom požadovanej stránky a ten odošle naspäť užívateľovi. Po obdržaní odpovede zo serveru sa užívateľovi zobrazí prijatý HTML dokument[11]. Tento proces je veľmi rýchly a hardvérové požiadavky na koncové užívateľské zariadenie sú minimálne. Problémom však je, že pri každej interakcii s webovou aplikáciou, napríklad otvorenie celého článku, kliknutie na odkaz, a podobne je na server odoslaná požiadavka na novú HTML stránku, ktorú server musí vygenerovať. Takýto prístup je pre server veľmi zaťažujúci, ak je webová aplikácia rozsiahlejšia alebo má

[1] **HTTP** je skratka z anglického **H**yper**t**ext **T**ransfer **P**rotocol.

veľa podstránok, pretože požiadavky sú príliš časté, a tak isto je zaťažovaný aj internetový prenos medzi serverom a klientom [2]. Pre užívateľa je isto nepríjemné, že pri interakcii s webovou aplikáciou sa znova načítava celé stránka len namiesto malej časti, ktorá sa reálne zmenila.




Obr. 2.1: Princíp fungovania SSR - Pri požiadavke o webovú stránku server musí vygenerovať finálny HTML súbor, čo vo väčšom počte spôsobí zvýšenie záťaže na server a tým aj spomalenie generovania webovej stránky pre klientov. Nároky na klienta sú však minimálne.

**Výhody použitia SSR:**

- Stránka je ľahko indexovateľná pre vyhľadávače (SEO[2])
- Prvotná stránka sa načíta rýchlejšie oproti CSR
- Vhodné pre stránky so statickým obsahom (prezentačné / reklamné)

**Nevýhody použitia SSR:**

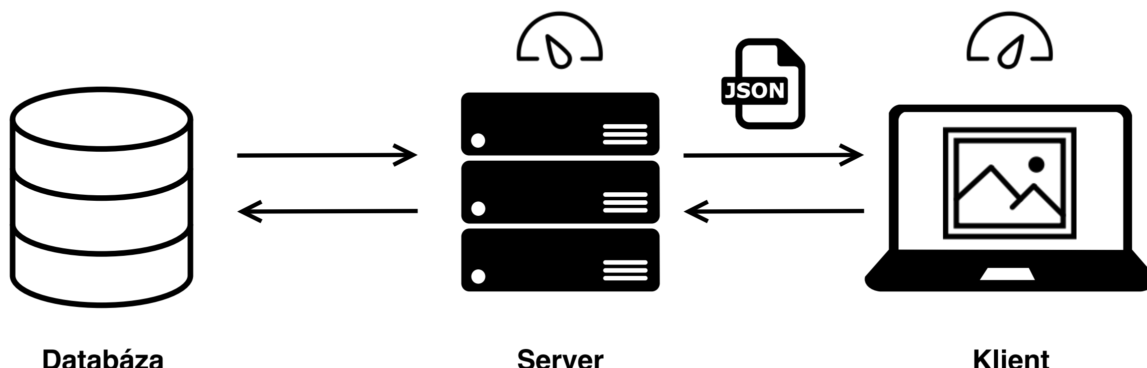
- Väčšia záťaž na server, kvôli častým dotazom
- Pre zmenu na stránke je potrebné ju celú znova načítať
- Časté načítavanie spôsobuje aj väčšiu záťaž pre sieť a celkovo je stránka pomalšia
- Kvôli obmedzeniam vyššie musí mať stránka menej interaktívnych prvkov

### 2.1.2 Client side rendering

Princípom fungovania CSR je presun generovania HTML stránok zo serveru na klienta. Pri požiadavke na zobrazenie webovej stránky pošle server klientovi len takmer prázdny HTML dokument s odkazmi na JavaScriptové knižnice. Webový prehliadač u užívateľa musí všetky tieto knižnice stiahnuť a po ich následnom spustení sa užívateľovi vygeneruje obsah webovej aplikácie [11]. Pri interakcii s webovou aplikáciou sa nenačítava celá webová stránka odznovu ako pri SSR, ale pregeneruje sa len časť, ktorá sa zmenila. Tento prístup sa rozšíril aj vďaka rastúcemu výkonu osobných počítačov a s tým súvisiacej väčšej popularite

[2]**SEO** je skratka z anglického **S**earch **E**ngine **O**ptimization - stránka ľahko analyzovateľná pre automaty internetových vyhľadávačov.

JavaScriptových knižníc a rámcov, ako napríklad Vue.js[3], React.js[4] alebo Angular[5] [19]. Pri CSR sa medzi klientom a serverom už ďalej neposielajú HTML stránky, ale len dáta, väčšinou vo formáte JSON[6], o ktoré užívateľ pri interakcii s aplikáciou požiada. Vo väčšine prípadov sa pri tomto prístupe využíva REST API 2.2.1, ktoré je špecifické práve tým, že sa medzi klientom a serverom posielajú len dáta a nie celé HTML stránky.




Obr. 2.2: Princíp fungovania CSR - Pri požiadavke o webovú stránku server odošle len malý HTML súbor a klient musí tento súbor následne spracovať a stiahnuť v ňom požadované knižnice, aby bolo možné zobraziť webovú stránku. Následne sú však medzi serverom a klientom posielané len dáta. Nároky na server sú v tomto prípade minimálne, ale ak sa jedná o komplexnejšie webové aplikácie, nároky na klienta sú vyššie z dôvodu generovania celého rozhrania.

**Výhody použitia CSR:**

- Rýchlejšie načítavanie vďaka odbúraniu nutnosti načítavať celú stránku odznova
- Väčšie možnosti interakcie s užívateľom
- Vhodné pre webové aplikácie s použitím veľkého množstva dostupných JavaScriptových knižníc

**Nevýhody použitia CSR:**

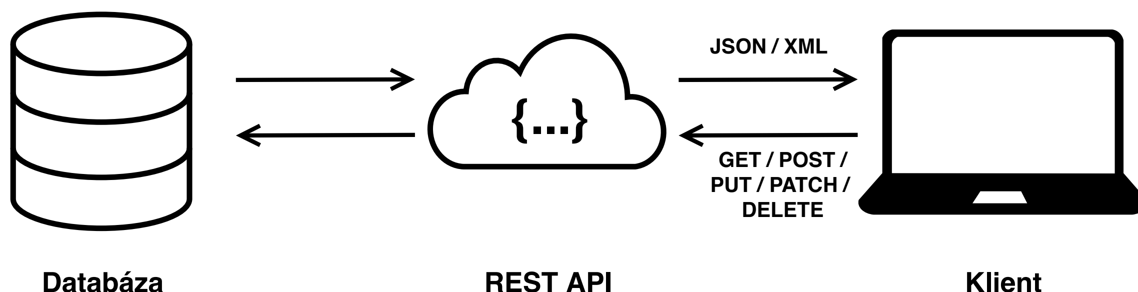
- Prvotné načítanie webovej stránky môže trvať dlhšie kvôli nutnosti stiahnuť všetky použité JavaScriptové knižnice
- Zložitá dostupnosť pre indexovacie automaty vyhľadávačov (SEO)
- Nutnosť väčšinou používať JavaScriptové knižnice, ktoré sa starajú o vykresľovanie obsahu

[3]`https://vuejs.org/`
[4]`https://reactjs.org/`
[5]`https://angular.io/`
[6]**JSON** je skratka z anglického **J**ava**S**cript **O**bject **N**otation (JavaScriptový objektový zápis) - najčastejšie sa tento formát využíva najmä v oblasti webových služieb a aplikácií.

## 2.2 Princípy a webové technológie

V nasledujúcej podkapitole budú popísané princípy a technológie, ktoré sú relevantné pri vývoji webových aplikácií. Tieto technológie a princípy sú dôležité najmä z pohľadu rozhrania a komunikácie medzi klientom a serverom.

### 2.2.1 REST[7] API




Obr. 2.3: Schéma princípu REST API - Klient komunikuje so serverom len pomocou HTTP metód a ako odpoveď získava požadované dáta vo formáte JSON alebo XML, nikdy nie celé HTML dokumenty. (Inšpirované z [https://www.seobility.net/en/wiki/REST_API]).

Jedná sa o jednu z najznámejších a najpoužívanejších architektúr rozhrania súčastnosti čo sa týka API webových aplikácií [8]. Túto architektúru vytvoril pán Roy Fielding v rámci jeho dizertačnej práce v roku 2000. **REST** rozhraniu sa konkrétne venuje 5. kapitola, v ktorej je popísaných 6 základných princípov, ktoré sú bližšie špecifikované nižšie [17]. Základným prvkom každého REST API sú zdroje (angl. resources). Každý zdroj je identifikovaný pomocou **URI**[8]. Zdroje sú objekty, ktoré majú svoj typ, obsahujú dáta a metódy, podobne ako objekty v OOP[9]. Zdroje však majú len pár definovaných metód, ktoré zodpovedajú metódam definovaným v HTTP, ale nie je podmienkou, že v REST musia plniť rovnakú funkciu ako v HTTP[7]. Napríklad na úpravu existujúcich dát sa môže používať POST metóda namiesto v HTTP odporúčanej PUT metódy. Dôležité však je, aby tento prístup bol jednotný v celom REST API. Najznámejšie a najpoužívanejšie metódy v REST sú: **GET**, **POST**, **PUT**, **PATCH** a **DELETE** [8]. Ich odporúčané použitie je nasledovné:

- **GET** - používa sa pre získanie / čítanie informácií zo zdroja
- **POST** - používa sa na vytvorenie nového zdroja
- **PUT** - používa sa pre úpravu existujúceho zdroja
- **PATCH** - používa sa pre úpravu len určitej časti existujúceho zdroja
- **DELETE** - používa sa pre odstránenie existujúceho zdroja

[7]**REST** je skratka z anglického **Re**presentational **S**tate **T**ransfer.
[8]**URI** je skratka z anglického **U**niform **R**esource **I**dentifier (jednotný identifikátor zdroja).
[9]**OOP** je skratka pre **O**bjektovo **o**rientované **p**rogramovanie.

Nasledovné popisy jednotlivých princípov som čerpal z dizertačnej práce [5] pána Roya Fieldinga, v ktorej definoval 6 základných princípov, na ktorých je postavená REST architektúra.

### Princíp č. 1: Klient-Server

Prvým princípom je použitie klient-server architektúry. V praxi to znamená oddelenie klienta a serveru, aby boli na sebe nezávislí. Klient by mal poznať iba URI zdrojov a konkrétna implementácia na strane serveru mu je neznáma a pre neho nepodstatná. Toto oddelenie umožňuje použitie serveru pre viacero aplikácií naraz bez nutnosti tvoriť pre každé použitie špecifický server. Jeden obecný server môže byť použitý rovnako pre webovú aplikáciu ako aj mobilnú aplikáciu alebo pre hocijaké iné koncové zariadenie.

### Princíp č. 2: Bezstavovosť (Stateless)

Druhým princípom je bezstavovosť, čo znamená, že požiadavka odoslaná na server musí obsahovať všetky potrebné údaje k získaniu zdroja, pretože na strane serveru nie je udržiavaný žiadny stav klienta. Celý stav udržiava len klient. Ak je teda potrebné, aby sa klient autorizoval pri požiadavke na server, musí každá požiadavka obsahovať autorizačný údaj, aby ju server akceptoval. To umožňuje serveru, aby bol čo najviacej obecný a odľahčuje jeho vyťaženosť, ktorú by spôsobilo manažovanie stavov jednotlivých klientov.

### Princíp č. 3: Cache[10]

Tretím princípom je využívanie cache pamäte na strane klienta alebo serveru ak je to možné. Dáta, ktoré majú byť uložené v cache pamäti však musia byť označené ako „*cacheable*“[11]. Tieto dáta sú väčšinou statické, a preto je vhodné ukladať ich do cache pamäte. Dáta uložené v cache pamäti na strane klienta je možné znovu použiť, ak klient žiada o rovnaké dáta a tým pádom nie je potrebné posielať požiadavku na server, čo šetrí dátový prenos medzi klientom a serverom. Dáta uložené v cache pamäti na strane serveru je možné použiť, ak niektorý z klientov žiada o dáta, ktoré už boli raz odoslané. To môže napríklad zmenšiť počet prístupov do databázy a ušetriť výpočtový výkon ak sa dáta dopočítavajú.

### Princíp č. 4: Jednotné rozhranie (Uniform Interface)

Štvrtým princípom, a zároveň asi jedným z najdôležitejších je jednotné rozhranie. Práve dôraz na jednotné rozhranie medzi jednotlivými časťami REST API odlišuje túto architektúru od iných používaných architektúr rozhrania. V praxi to môže znamenať, že ak sa použije metóda POST na vytváranie aj editovanie zdrojov, je potrebné tento prístup zachovať naprieč všetkými zdrojmi, aby interakcia so zdrojmi bola jednotná. Tento princíp umožňuje zjednodušenie architektúry systému, ale zároveň spôsobuje, že dáta a štruktúra dát posielaných klientom sú menej špecifické pre konkrétne použitie u jednotlivých klientov. Aby bolo rozhranie jednotné, sú potrebné pravidlá pre usmernenie.

[10] **Cache** je anglický výraz pre **rýchlu vyrovnávaciu pamäť**. Tento výraz je však v oblasti informatiky veľmi známy a často sa používa aj v našich textoch.

[11] **Cacheable** je anglický výraz pre príznak u dát, ktoré je možné uložiť do rýchlej vyrovnávacej pamäte (cache).

REST má tieto 4 pravidlá:

1. Identifikácia zdrojov - URI
2. Manipulácia so zdrojmi len pomocou ich reprezentácií - zdroj reprezentovaný napr. ako JSON alebo XML
3. Samopopisujúce správy - obsahujú všetky potrebné údaje pre získanie zdroja
4. HATEOAS[12] - server môže vrátiť v rámci odpovede odkazy na ďalšie podzdroje, ktoré rozšíria možnosti klienta na získanie konkrétnejšieho zdroja vzhľadom na aktuálny stav dát [17].

#### Princíp č. 5: Vrstvený systém (Layered System)

Piatym princípom je rozdelenie systému do vrstiev, ktoré medzi sebou komunikujú, ale žiadna nevidí za vrstvu, s ktorou priamo komunikuje. Tento prístup nám umožňuje rozdeliť napríklad jednotlivé časti serveru na rozdielne fyzické servery, a tak zmierniť záťaž serveru, s ktorým komunikuje klient. Klient však vidí len jeden server a to práve ten, s ktorým komunikuje. Vrstvený systém síce zvyšuje zložitosť celého systému, ale umožňuje väčšiu flexibilitu, ako napríklad zapuzdrenie starších služieb pre podporu starších klientov. Najväčšou nevýhodou tohto systému je, že sa z pohľadu klienta zvyšuje čas potrebný na spracovanie požiadaviek, pretože jednotlivé časti systému potrebujú čas pre finálne sprístupnenie zdroja klientovi.

#### Princíp č. 6: Kód na požiadanie (Code-On-Demand)

Posledným princípom je kód na požiadanie, ktorý umožňuje rozširovanie klienta počas behu aplikácie. Tento princíp je voliteľný, pretože nie všetci klienti môžu mať oprávnenie takýto kód spúšťať. Zavedenie tohto princípu však umožňuje na požiadanie klienta pridať novú funkcionalitu čo zjednodušuje implementáciu na strane klienta a zároveň aj schováva časť funkcionality aplikácie. Väčšinou sa tento princíp nevyužíva, ale zároveň je v prípade potreby možné ho využiť a neodporuje princípom REST architektúry.

### 2.2.2 Dokumentácia API

Dôležitou časťou pri tvorbe webových aplikácií je zdokumentovanie vytvoreného API. Najjednoduchším spôsobom je voľný textový popis, ktorý je jednoducho čitateľný a pochopiteľný pre programátora webovej aplikácie, ale je dlhodobo neudržateľný, najmä vo väčších tímoch a strojovo nespracovateľný. Preto vznikla potreba vytvoriť dokumentačné jazyky a štandardy pre popis týchto rozhraní, ako jazyk WADL 2.2.3 alebo OpenAPI 2.2.4 [1].

[12] **HATEOAS** je skratka z anglického **H**ypermedia **as** **t**he **E**ngine **of** **A**pplication **S**tate (Hypermedia ako aplikačný stav).

### 2.2.3 WADL[13]

Jedným z najstarších dokumentačných jazykov pre popis REST rozhraní je práve WADL. Taktiež ako OpenAPI 2.2.4 je určený pre dokumentovanie API založených na HTTP. WADL dokumentácia sa zapisuje vo formáte XML a je strojovo spracovateľná [6]. WADL vznikol v roku 2006 a vytvorila ho firma Sun Microsystems. Vychádza z jazyka WSDL[14], ktorý je určený pre popis SOAP[15] rozhraní, ale nie je úplne vhodný pre popis REST rozhraní, preto vznikol WADL [1]. V roku 2009 bol uvoľnený a odovzdaný pod správu W3[16] konzorcia, ale odvtedy nebolo naplánované žiadne rozširovanie alebo štandardizovanie tohto jazyka [10].

```
<?xml version="1.0"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:yn="urn:yahoo:yn"
xmlns="http://wadl.dev.java.net/2009/02">
 <resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
  <resource path="newsSearch">
   <method name="GET" id="search">
    <request>
     <param name="query" type="xsd:string" style="query" required="true"/>
     <param name="results" style="query" type="xsd:int" default="10"/>
    </request>
    <response status="200">
     <representation mediaType="application/xml" element="yn:ResultSet"/>
    </response>
   </method>
  </resource>
 </resources>
</application>
```

Výpis 2.1: Ukážka časti jednoduchej WADL dokumentácie v XML formáte. (Prevzaté z [https://www.w3.org/Submission/wadl/]).

Koreňovým elementom WADL dokumentácie je element **application**. Ten obsahuje element **resources**, ktorý obsahuje všetky dostupné zdroje a definuje základnú url, kde sa dokumentované API nachádza. V ňom sa nachádza element **resource**, ktorý definuje cestu ku konkrétnemu zdroju. Obsahuje jeden alebo viaceré elementy **method**, ktoré definujú dostupné metódy nad konkrétnym zdrojom.
Tento element obsahuje práve jeden element **request**, v ktorom sú definované požadované vstupné parametre. Metóda môže obsahovať ešte jeden alebo viacero elementov **response**, tie však nie sú povinné. V tomto elemente je definovaná štruktúra a typ správy, ktorú server vracia ako odpoveď na požiadavku. WADL poskytuje samozrejme viacero možností okrem uvedených príkladov v tejto krátkej ukážke[17].

[13] **WADL** je skratka z anglického **W**eb **A**pplication **D**escription **L**anguage.
[14] **WSDL** je skratka z anglického **W**eb **S**ervices **D**escription **L**anguage.
[15] **SOAP** je skratka z anglického **S**imple **O**bject **A**ccess **P**rotocol.
[16] **W3** je skratka pre **W**orld **W**ide **W**eb.
[17] Kompletná špecifikácia je dostupná na stránkach W3 konzorcia https://www.w3.org/Submission/wadl/.

### 2.2.4 OpenAPI

Pre dokumentovanie REST rozhraní existuje viacero štandardov. Jedným z najznámejších a najrozšírenejších je práve OpenAPI. Je to dokumentačný štandard, ktorý určuje štýl akým sa majú dokumentovať API založené na HTTP. Väčšinou sa jedná práve o REST API. OpenAPI dokumentácia sa zapisuje vo formáte YAML alebo JSON. Táto dokumentácia obsahuje popisy vstupov a výstupov API, ako aj server, na ktorom sa dané API nachádza [8]. OpenAPI špecifikácia vznikla v roku 2017 zo Swagger špecifikácie ako reakcia na potrebu zjednotiť dokumentačné štandardy pre API. OpenAPI je pod správou OpenAPI Initiative[18], ktorú podporili veľké technologické firmy ako Google, Microsoft, IBM a mnoho ďalších [3].

```
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description.
  version: 1.0.0
servers:
  - url: http://api.example.com/v1
    description: Optional server description.
paths:
  /users:
    get:
      summary: Returns a~list of users.
      description: Optional extended description.
      operationId: getListOfUsers,
      responses:
        '200':
          description: A~JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

Výpis 2.2: Ukážka jednoduchej OpenAPI dokumentácie v YAML formáte. (Prevzaté z [https://swagger.io/docs/specification/basic-structure/]).

Každá dokumentácia musí obsahovať kľúč **openapi**, ktorý definuje verziu použitého štandardu. Nasleduje kľúč **info**, ktorý definuje názov dokumentácie, verziu a prípadne detailnejší popis dokumentácie. Kľúč **servers** obsahuje pole URL adries k serverom, ktoré podporujú zdokumentované API.
Posledný kľúč **paths** obsahuje adresy na existujúce zdroje, ktoré API podporuje. Tieto adresy potom obsahujú definície metód, ktoré je nad konkrétnym zdrojom možné volať. Nakoniec sú v metódach definované vstupné parametre, ak metóda nejaké vyžaduje a štruktúra odpovede, ktorú vracia server pri zavolaní tejto metódy. OpenAPI štandard je samozrejme rozsiahlejší a poskytuje oveľa viac možností, ako je uvedené v tomto príklade[19].

[18] https://www.openapis.org/
[19] Kompletná špecifikácia je dostupná na http://spec.openapis.org/oas/v3.0.3.

Výhodou OpenAPI štandardu oproti WADL 2.2.3 je napríklad to, že je nie len strojovo čitateľný, ale aj jednoduchšie čitateľný pre používateľa, ktorý chce pochopiť a používať zdokumentované API [4].

### 2.2.5 TypeScript[20]

Väčšina webových aplikácií je napísaná v programovacom jazyku **JavaScript**. Tento jazyk je podporovaný vo všetkých moderných webových prehliadačoch. Dokonca je možné napísať aj serverovú časť aplikácie v JavaScripte. Avšak pri rozsiahlejších aplikáciách, kde spolupracuje viacero vývojárov, môže nastať veľa zbytočných chýb kvôli tomu, že JavaScript nemá takmer žiadnu typovú kontrolu a nepodporuje triedy ani rozhrania, existujú tu iba objekty. Preto je vhodnejšie použitie jazyka TypeScript, ktorý prináša so sebou veľa výhod.

**TypeScript** je programovací jazyk, ktorý je transpilovaný do JavaScriptu, takže nie je potrebné upravovať webové prehliadače a iné technológie pre spustenie webovej aplikácie. Bol navrhnutý pánom Andersom Hejlsbergom a je pod správou Microsoftu. Jedná sa však o open-source[21] projekt, aby bol dostupný čo najväčšej komunite a každý môže prispieť svojím nápadom pre zlepšenie tohto jazyka. Prináša do JavaScriptu typy, statickú typovú kontrolu, možnosť definovať triedy a rozhrania [18]. Tieto prvky pomáhajú písať čistejší a prehľadnejší kód s menším počtom chýb spôsobených nepozornosťou alebo prehliadnutím prepísania typu premennej.

### 2.2.6 Angular[22]

Pri tvorbe moderných webových aplikácií sa často využívajú veľké JavaScriptové knižnice alebo frameworky[23], ktoré uľahčia a vytvoria základ aplikácie ako vykresľovanie samotnej webovej aplikácie, správu a ukladanie stavu aplikácie alebo prepojenie medzi klientom a serverom. Medzi jeden z najpopulárnejších frameworkov patrí práve Angular.

**Angular** je JavaScriptový framework vytvorený spoločnosťou Google v roku 2016 pre tvorbu webových aplikácií. Je to open-source projekt, ale jeho hlavným prispievateľom je práve spoločnosť Google. Angular vyžaduje používanie jazyka TypeScript v rámci celej aplikácie, čo je pre niektorých vývojárov odradzujúce. Keďže sa jedná o framework a nie len knižnicu, obsahuje už v základe veľa potrebných nástrojov, ktoré v prípade využitia iných technológií pri tvorbe webovej aplikácie musia byť pridané ako externé knižnice, no niekedy sú práve tieto nástroje nadbytočné a navyšujú veľkosť jednoduchej aplikácie. Aj keď Angular patrí medzi najpopulárnejšie JavaScriptové frameworky, jeho popularita stále klesá na úkor novších a výkonnejších JavaScriptových knižníc ako Vue.js alebo React, ktoré ponúkajú vyšší výkon a nepôsobia ťažkopádne a mohutne ako Angular [13].

[20] https://www.typescriptlang.org/

[21] **Open-source** (otvorený kód) je typ projektu, ktorého zdrojový kód je dostupný verejnosti a ktokoľvek za určitých podmienok môže prispieť k jeho ďalšiemu vývoju.

[22] https://angular.io/

[23] **Framework** je anglický výraz pre **aplikačný rámec**. Tento anglický výraz je však v oblasti informatiky veľmi známy práve v jeho anglickej podobe.

## 2.3 Existujúce a podobné riešenia

V oblasti generovania kostry aplikácie z popisu API rozhrania už existujú samozrejme riešenia, ktoré sa zaoberajú touto problematikou. Pri návrhu aplikácie som zopár takýchto riešení našiel a využil ich práve pre zhodnotenie ich výhod i nevýhod a pokúsil sa navrhnúť nástroj, ktorý odstráni tieto ich nedostatky.

### 2.3.1 api-client-generator[24]

Nástroj **api-client-generator** je uložený v repozitáry brnenskej firmy FlowUp, ktorá má v oblasti vývoja webových aplikácií bohaté skúsenosti. Tento nástroj generuje základnú štruktúru webovej aplikácie potrebnú pre komunikáciu medzi klientom a serverom. Nástroj túto štruktúru generuje z dokumentácie REST API v OpenAPI alebo Swagger štandarde pre framework Angular. Pri testovaní nástroja sa mi páčilo jednoduché nastavovanie vstupných parametrov a medzi výhody nepochybne patrí, že nástroj dokáže pracovať aj s predchodcom OpenAPI štandardu a to Swagger dokumentačným štandardom. Výsledné vygenerované súbory sa mi zdali niekedy až príliš komplikované a rozdelené do veľa čiastkových súborov. Čo však považujem za najväčší nedostatok nástroja je, že často sa mu nepodarilo nájsť definície objektov v OpenAPI dokumentácii, ktoré vracia API a teda nevygeneroval modely, ktoré by zodpovedali týmto definíciám. Odpovede zo serveru tak nemali žiadny definovaný typ a prichádzali by sme tak o výhody použitia jazyka TypeScript.

### 2.3.2 openapi-typescript-client-api-generator[25]

Ďalší nástroj, ktorý som našiel bol **openapi-typescript-client-api-generator**. Tento nástroj generuje kostru pre aplikácie, ktoré používajú pre komunikáciu so serverom knižnicu **Axios**. Tento nástroj ma zaujal jednoduchou a prehľadnou výslednou adresárovou štruktúrou a prehľadným štýlom vygenerovaných zdrojových kódov pre jednotlivé požiadavky na server. Tak isto ako nástroj vyššie podporuje okrem OpenAPI aj Swagger dokumentačný štandard. Medzi jeho nevýhody radím to, že má pomerne zbytočne komplikované nastavenie vstupných parametrov, vďaka čomu bol zo začiatku problém len so samotným spustením nástroja. Ďalšou a oveľa zásadnejšou nevýhodou je to, že pri väčších a komplikovanejších OpenAPI dokumentáciách nástroj často skončil veľkou a nič nehovoriacou chybou, ktorá nastala pri analýze vstupného súboru. S jednoduchšími dokumentáciami si však nástroj poradil celkom spoľahlivo.

### 2.3.3 Swagger Codegen[26]

S týmto nástrojom som sa stretol pri práci s OpenAPI špecifikáciou a jedná sa o pomerne komplexný nástroj. Nachádza sa v rámci *SwaggerHub* aplikácie, ktorá obsahuje viaceré nástroje pre dokumentovanie i generovanie výsledných štruktúr. Nástroj síce nezapadá úplne priamo do procesu vývoja ako predchádzajúce dve riešenia, ale vo väčšine prípadov je dokumentácia napísaná v OpenAPI uložená práve v *SwagerHub* aplikácii. Nástroj obsahuje generovanie klientskej kostry pre veľa jazykov ako Java, PHP, C#, Python či TypeScript. Okrem toho je možné z popisu dokumentácie vygenerovať aj serverovú časť aplikácie. Klientská kostra aplikácie v jazyku TypeScript je však určená len pre framework Angular.

[24] https://www.npmjs.com/package/api-client-generator
[25] https://www.npmjs.com/package/@progresso/openapi-typescript-client-api-generator
[26] https://swagger.io/tools/swagger-codegen/

# Kapitola 3

# Návrh riešenia

Dôležitou časťou pri vývoji je analýza požiadaviek a návrh riešenia, ktoré bude spĺňať tieto požiadavky. Preto je potrebné tejto časti venovať pozornosť, pretože zlý návrh môže spôsobiť, že finálny program bude nepoužiteľný a bude ho treba celý prerobiť, čo je v praxi veľmi časovo i finančne náročné a samozrejme neprípustné.

V tejto kapitole budú najprv špecifikované práve požiadavky, ktoré by mal finálny nástroj spĺňať. Následne bude popísaný návrh, ktorý by mal vyhovovať požadovaným vlastnostiam z pohľadu nástroja ako celku a potom bude priblížený návrh jednotného štandardizovaného rozhrania, do ktorého budú spracované všetky vstupné súbory. Na záver bude popísaný výber technológií potrebných pre vytvorenie finálneho nástroja.

## 3.1 Analýza požiadaviek

Najdôležitejšou požiadavkou je schopnosť generovať funkčný základ webovej aplikácie pre komunikáciu medzi klientom a serverom. Generovanie tohto základu by malo zapadať do procesu vývoja webovej aplikácie, aby programátor nebol nútený inštalovať si aplikácie alebo programy naviac, ktoré by neboli súčasťou projektu prípadne prostredia, v ktorom pracuje. Tento základ musí byť v jazyku TypeScript pre väčšiu čistotu kódu a možnosti používať dátové typy, ktoré pomáhajú vyvarovať sa chybám, ktoré môžu vzniknúť v jazyku JavaScript práve kvôli nechcenému pretypovaniu nejakej premennej. Ideálny cieľový framework je práve Angular, ktorý vyžaduje použitie TypeScriptu a zároveň už v základe obsahuje nástroje potrebné pre komunikáciu medzi klientom a serverom. Dobrým prístupom by však bolo podporovať aj nejakú knižnicu, ktorá sprostredkúva komunikáciu medzi klientom a serverom v prípade, ak sa programátor rozhodne nepoužiť pre svoju aplikáciu framework Angular.

Ďalšou dôležitou požiadavkou je podporovať také dokumentačné jazyky API, ktoré patria medzi najčastejšie používané medzi programátormi. Jedným z týchto jazykov je práve OpenAPI, ktoré je v súčasnosti najpoužívanejším štandardom a veľa verejne dostupných API má dokumentáciu práve v tomto jazyku. Jazyk WADL síce nepatrí dnes medzi najpopulárnejšie, no jeho dôležitosť spočíva v tom, že z webových API napísaných v jazyku Java je možné vygenerovať dokumentáciu práve v jazyku WADL.

Poslednou požiadavkou na výsledný nástroj je možnosť distribúcie tohto nástroja medzi vývojárov tak, aby bol dostupný širokej verejnosti a mohol ho pri vývoji webovej aplikácie použiť hocikorý programátor, ktorému použitie tohto nástroja uľahčí vývoj.

Požiadavky na výsledný nástroj by sa teda dali rozdeliť do dvoch kategórií: **formálne požiadavky**, ktoré vyplývajú priamo zo zadania a **neformálne požiadavky**, ktoré boli upresnené počas konzultácií s vedúcim práce.

**Formálne požiadavky:**

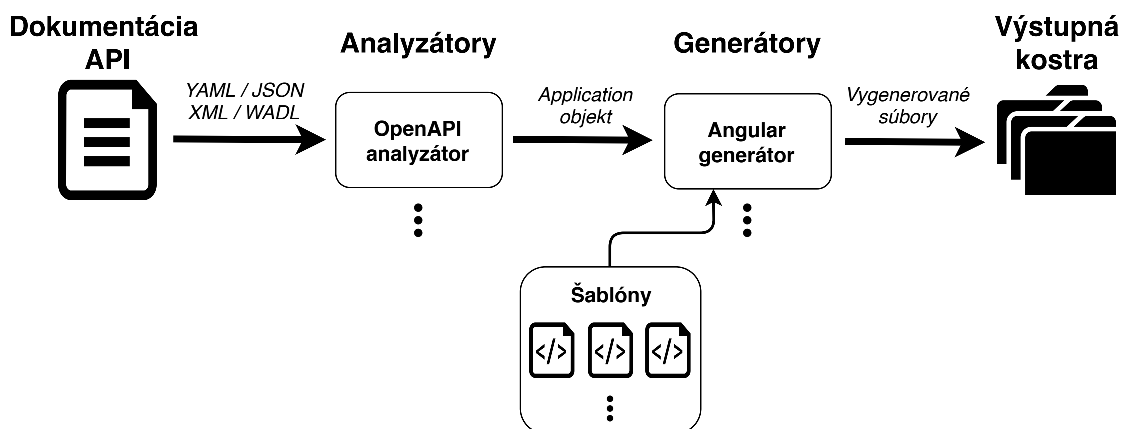
- Nástroj musí generovať funkčnú kostru pre komunikáciu medzi klientom a serverom
- Generovaná kostra musí byť v jazyku TypeScript
- Nástroj musí podporovať generovanie kostry pre framework Angular
- Musia byť podporované dokumentačné jazyky OpenAPI a WADL

**Neformálne požiadavky:**

- Nástroj by mal zapadať do procesu vývoja webovej aplikácie bez nutnosti inštalácie ďalších aplikácií
- Nástroj by mal podporovať generovanie kostry aj pre nejakú knižnicu slúžiacu na komunikáciu medzi klientom a serverom
- Nástroj by mal byť ľahko distribuovateľný medzi programátorov webových aplikácií

## 3.2 Architektúra nástroja

Výsledný nástroj by sa dal rozdeliť na dve najzákladnejšie časti, a to analyzátor (parser) vstupného súboru a generátor kostry aplikácie. Analyzátor spracúva vstupný súbor obsahujúci dokumentáciu REST API a následne vytvára objekt typu **Application** 3.3, ktorý obsahuje všetky relevantné informácie pre generátor získané zo vstupného súboru. Generátor tento objekt analyzuje a za pomoci šablón jednotlivých častí aplikácie vygeneruje kostru aplikácie pre komunikáciu medzi klientom a serverom.

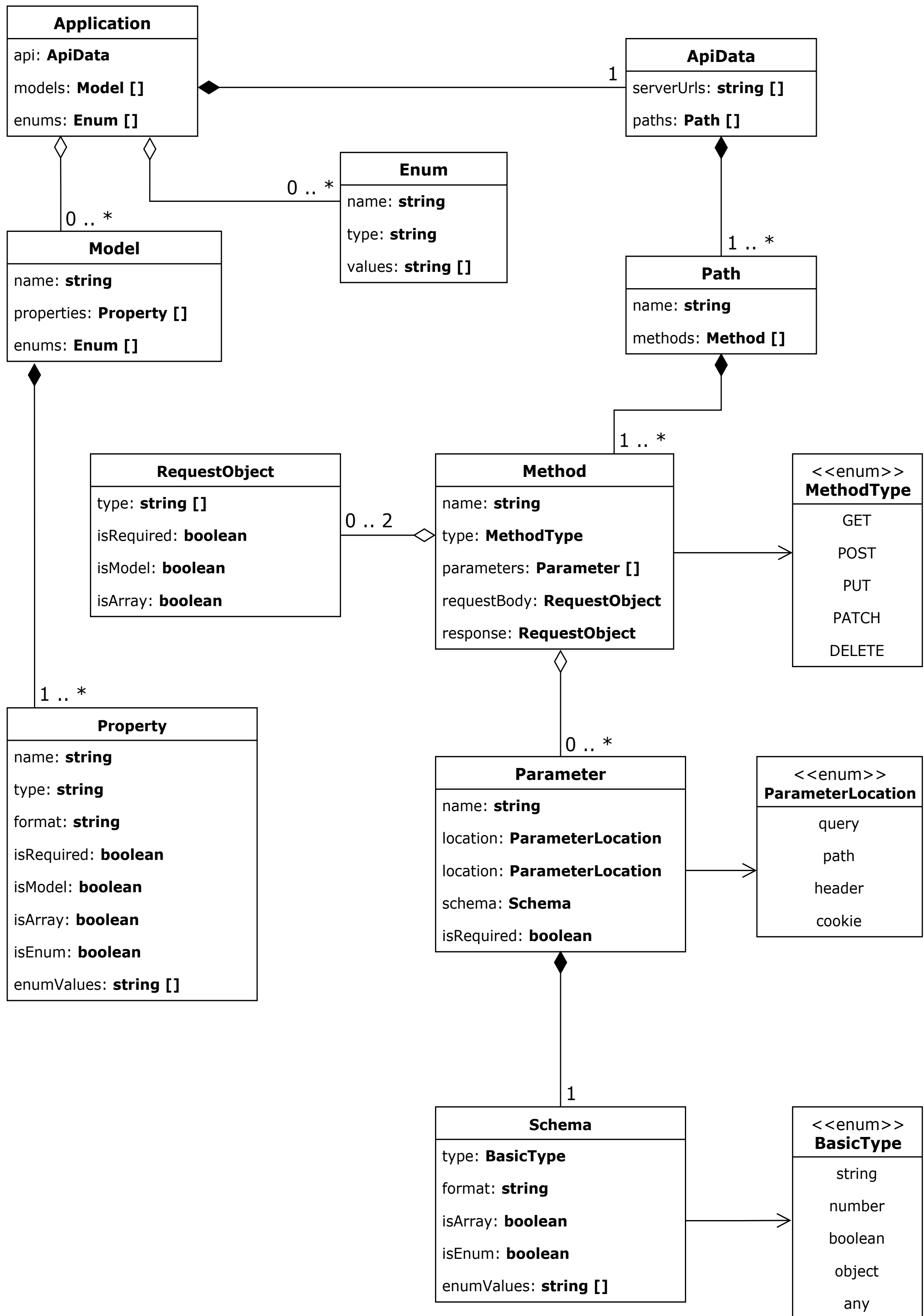



Obr. 3.1: Návrh architektúry nástroja - Na vstupe sa nachádza súbor obsahujúci dokumentáciu, tento súbor je načítaný a analyzovaný konkrétnym analyzátorom a informácie z neho získané sú predané generátoru, ktorý na základe týchto informácií vygeneruje pomocou šablón výslednú kostru webovej aplikácie.

## 3.3 Návrh štandardizovaného rozhrania

Pre udržateľnosť a možnosť ďalšieho rozšírenia nástroja je potrebné si definovať jednotné a štandardizované rozhranie, cez ktoré budú medzi sebou komunikovať analyzátor a generátor. Vďaka štandardizovanému rozhraniu bude možné pridávať do nástroja analyzátory ďalších dokumentačných jazykov, ako aj pridávať nové generátory, napríklad pre iné JavaScriptové frameworky alebo knižnice. Preto bolo rozhodnuté, že každý analyzátor musí po analýze vstupného súboru vracať objekt typu Application a každý generátor ako svoj vstupný parameter vyžaduje práve objekt typu Application.

Trieda **Application** 3.3.1 sa skladá z viacero ďalších tried, a preto bol pre lepšiu vizualizáciu vytvorený diagram tried, ktorý špecifikuje štruktúru a vzťahy medzi ďalšími triedami. Niektoré prvky tejto triedy boli inšpirované informáciami obsiahnutými v OpenAPI dokumentácii, pretože špecifikuje viacej informácií ako WADL dokumentácia. Takže pre WADL je tento objekt rovnako vhodný, len nebude obsahovať niektoré detaily, ktoré je možné získať len z OpenAPI dokumentácie. Kompletný diagram tried je možné vidieť nižšie. Následne budú popísané jednotlivé časti tohto diagramu.




Obr. 3.2: Návrh štandardizovaného rozhrania znázornený pomocou diagramu tried.

### 3.3.1 Trieda Application

| **Application** |
| --- |
| api : **ApiData**<br>models : **Model** [ ]<br>enums : **Enum** [ ] |
| |

Táto trieda je hlavným komunikačným objektom medzi analyzátorom a generátorom. Atribút **api** je typu **ApiData** 3.3.3 a obsahuje informácie o jednotlivých zdrojoch REST API a ďalšie potrebné informácie. Ďalší atribút **models** je pole typu **Model** 3.3.9 a obsahuje informácie o štruktúre a typoch definovaných objektov, ktoré budú posielané medzi klientom a serverom. Posledným atribútom je **enums**, ktorý je pole typu **Enum** 3.3.2 a obsahuje definície množín pomenovaných hodnôt, ktoré budú použité pri komunikácii v rámci tohto konkrétneho API.

### 3.3.2 Trieda Enum

| **Enum** |
| --- |
| name : **string**<br>type : **string**<br>values: **string** [ ] |
| |

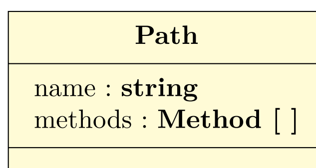
Trieda Enum obsahuje informácie o jednotlivých konečných množinách definovaných hodnôt, ktoré sú definované v dokumentácii API. Atribút **name** obsahuje názov konkrétnej množiny hodnôt. Ďalší atribút **type** definuje typ hodnôt, ktoré množina nadobúda. Posledný atribút **values** obsahuje jednotlivé hodnoty, ktoré sú definované v tejto množine hodnôt.

### 3.3.3 Trieda ApiData

| **ApiData** |
| --- |
| serverUrls : **string** [ ]<br>paths : **Path** [ ] |
| |

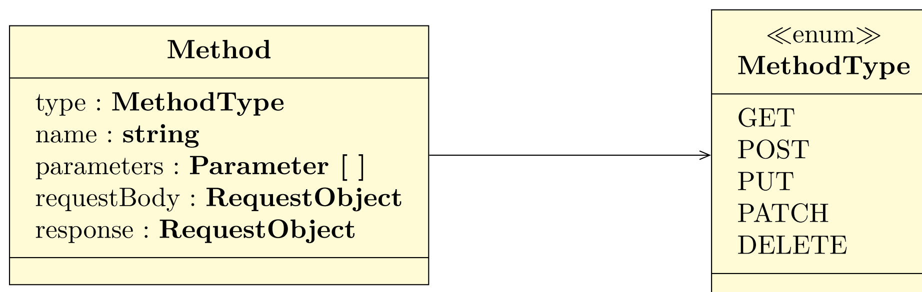
Trieda ApiData združuje všetky potrebné informácie týkajúce sa adries pre prístup k serveru a samotným zdrojom v REST API. Atribút **serverUrls** obsahuje pole URL adries k serverom, na ktorých sa nachádza zdokumentované REST API. Atribút **paths**, ktorý je pole typu **Path** 3.3.4, obsahuje jednotlivé adresy k zdrojom, ktoré REST API obsahuje.

### 3.3.4 Trieda Path



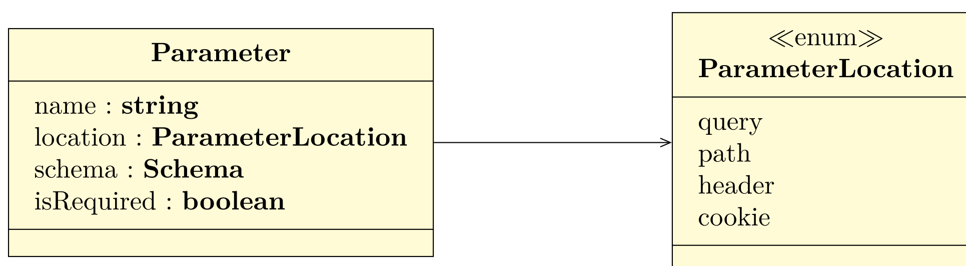

Trieda Path obsahuje informácie o jednotlivých zdrojoch REST API. Atribút **name** obsahuje cestu (URI) ku konkrétnemu zdroju. Druhý atribút **methods** je pole typu **Method** 3.3.5 a obsahuje všetky metódy, ktoré je možné volať nad konkrétnym zdrojom (napr. GET, POST, PUT a iné).

### 3.3.5 Trieda Method



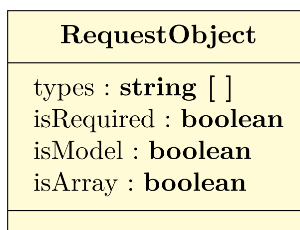

Trieda Method obsahuje informácie o konkrétnej metóde, ktorú je možné volať nad konkrétnym zdrojom definovaným v triede Path 3.3.4. Atribút **type** je konečná množina definovaných hodnôt typu **MethodType** a určuje typ metódy zodpovedajúci HTTP metóde. Atribút **name** obsahuje názov metódy, ktorý je definovaný v dokumentácii a bude použitý pre pomenovanie finálnej metódy vo výslednom kóde. Atribút **parameters** je pole typu **Parameter** 3.3.6 a obsahuje zoznam všetkých parametrov, ktoré je možné použiť pri volaní konkrétnej metódy nad zdrojom. Atribút **requestBody** je objekt typu **RequestObject** 3.3.7 a obsahuje definíciu štruktúry a typu dát požiadavky, ktorá je typu POST, PUT alebo PATCH. Posledný atribút **response** je rovnako typu **RequestObject** a obsahuje definíciu štruktúry a typu dát, ktoré budú vrátené ako odpoveď zo serveru.

### 3.3.6 Trieda Parameter



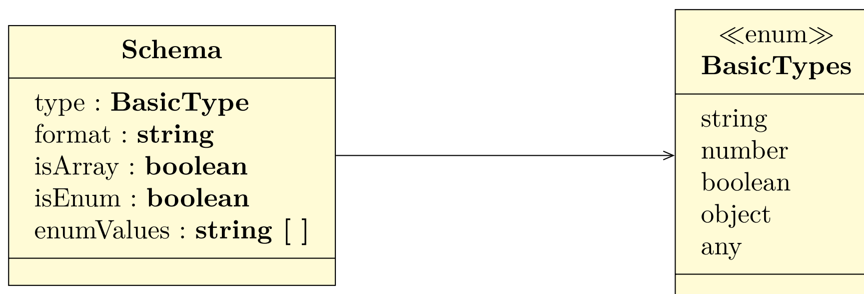

Trieda Parameter obsahuje informácie o konkrétnom parametre, ktorý je možné použiť pri volaní metódy nad konkrétnym zdrojom. Atribút **name** obsahuje názov parametru. Atribút **location** je konečná množina definovaných hodnôt typu **ParameterLocation**, ktorý určuje umiestnenie parametru pri volaní metódy nad zdrojom. Atribút **schema** obsahuje objekt typu **Schema** 3.3.8, ktorý špecifikuje typ a prípustné hodnoty konkrétneho parametru. Posledný atribút **isRequired**, určuje či je parameter pri volaní metódy povinný.

### 3.3.7 Trieda RequestObject



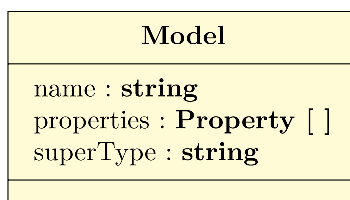

Trieda RequestObject obsahuje informácie o objekte, ktorý je posielaný na server v rámci metód POST, PUT a PATCH. Tak isto táto trieda slúži na popis objektu, ktorý server vráti ako odpoveď na požiadavku odoslanú z klienta. Použitie tejto triedy pre oba tieto typy je vhodné pre veľkú podobnosť ich štruktúry. Atribút **types** je pole prípustných typov pre objekt, ktorý bude poslaný na server alebo z neho príde ako odpoveď. Atribút **isRequired** definuje či daná metóda musí obsahovať v rámci požiadavky definovaný objekt. Atribút **isModel** určuje či aspoň jeden z typov objektu je užívateľom definovaná trieda. Posledný atribút **isArray** definuje či sa jedná o pole objektov alebo len o jeden samostatný objekt.

### 3.3.8 Trieda Schema




Trieda Schema špecifikuje typ, štruktúru a prípustné hodnoty parametru, ktorý sa používa pri volaní konkrétnej metódy nad zdrojom. Atribút **type** je konečná množina definovaných hodnôt typu **BasicTypes**, ktorá špecifikuje typ parametru jedným zo základných typov používaných v jazyku TypeScript. Atribút **format** obsahuje dodatočnú informáciu o type, napríklad špecifikuje, že sa jedná o integer[1]. Atribút **isArray** určuje či parameter obsahuje len jednu hodnotu špecifikovaného typu alebo pole hodnôt tohto typu. Atribút **isEnum** určuje či parameter môže nadobúdať hodnotu z konečnej množiny definovaných hodnôt, ktoré sa nachádzajú v atribúte **enumValues**.

### 3.3.9 Trieda Model




Trieda Model obsahuje informácie o štruktúre triedy, resp. objektu, ktorý je definovaný v dokumentácii API a bude používaný pri komunikácii medzi klientom a serverom.

[1] TypeScript nerozlišuje medzi celými a desatinnými číslami a označuje ich spoločným typom number.

Atribút **name** obsahuje názov definovanej triedy. Atribút **properties** je pole typu **Property** 3.3.10 a obsahuje definície jednotlivých atribútov triedy.

### 3.3.10 Trieda Property

| **Property** |
| --- |
| name : **string**<br>type : **string**<br>format : **string**<br>isRequired : **boolean**<br>isModel : **boolean**<br>isArray : **boolean**<br>isEnum : **boolean**<br>enumValues : **string** [ ] |
| |

Trieda Property obsahuje špecifikáciu typu a štruktúry atribútu, ktorý je obsiahnutý v triede definovanej v rámci dokumentácie API. Atribút **name** špecifikuje meno atribútu triedy, **type** definuje jeho typ a **format** špecifikuje tento typ rovnako ako v prípade triedy **Schema** 3.3.8. Atribút **isRequired** určuje či je atribút povinný a musí byť teda definovaný v každej inštancii triedy. Atribút **isModel** určuje či je typ atribútu iná definovná trieda. Atribút **isArray** určuje či je atribút pole definovaného typu. Atribút **isEnum** určuje či atribút môže nadobúdať hodnotu z konečnej množiny definovaných hodnôt, ktoré sú definované v atribúte **enumValues**.

## 3.4 Výber technológií pre vývoj

Ďalšou dôležitou časťou pri tvorbe návrhu je vybrať si správne a vhodné technológie, pomocou ktorých bude navrhnuté riešenie implementované. Keďže jednou z požiadaviek na výsledné riešenie bolo, aby nástroj zapadal do procesu vývoja webovej aplikácie a nebolo nutné inštalovať ďalšie programy, bolo nutné napísať aplikáciu v JavaScripte pre prostredie **Node.js**.

### 3.4.1 Node.js[2]

Node.js je prostredie, ktoré umožňuje spúšťanie programov napísaných v jazyku JavaScript mimo internetového prehliadača. Primárne je určené najmä pre vytváranie jednoduchých serverov napísaných práve v JavaScripte. Node.js je postavené na V8 engine[3]. Toto prostredie sa používa pri vývoji webových aplikácií pre spúšťanie a testovanie aplikácie počas vývoja, ako aj pre spúšťanie rôznych skriptov, ktoré transformujú zložité UI knižnice, prípadne frameworky typu Angular na jednoduchšie HTML stránky, CSS štýly a JavaScriptové kódy, ktoré sú spustiteľné v internetovom prehliadači. Súčasťou Node.js je aj balíčkovací systém **npm**, ktorý slúži na inštaláciu a správu knižníc z najväčšieho repozitára JavaScriptových knižníc[4] [16]. Práve do tohto repozitára je nahrané výsledné riešenie, aby bolo dostupné čo najväčšiemu počtu vývojárov.

[2]https://nodejs.org/en/

[3]**V8 engine** je interpret jazyka JavaScript, vytvorený spoločnosťou Google pre open-source projekt Chromium a webový prehliadač Chrome. https://v8.dev/.

[4]https://www.npmjs.com/

### 3.4.2 TypeScript[5]

Aj keď výsledná vygenerovaná kostra aplikácie má byť v jazyku TypeScript, nie je nutnosťou, aby aj nástroj, ktorý túto kostru generuje, musel byť v jazyku TypeScript. Keďže sa však bude jednať o rozsiahlejší nástroj, je vhodnejšie použiť práve jazyk TypeScript 2.2.5, ktorý umožňuje písať typovo bezpečnejší a čistejší kód. Pred publikovaním však musí byť výsledné riešenie transpilované do JavaScriptu, aby bolo možné tento nástroj použiť v prostredí Node.js.

### 3.4.3 Handlebars[6]

Handlebars je jazyk pre písanie šablón, z ktorých je možné okrem HTML kódov, pre ktoré je tento jazyk primárne určený, generovať rôzne textové výstupy. Generovanie prebieha pomocou knižnice[7], ktorej je na vstupe predaná šablóna napísaná v jazyku Handlebars a objekt, z ktorého budú čerpané dáta pre doplnenie do šablóny. Jazyk Handlebars vychádza z jazyka Mustache[8], ale pridáva naviac ďalšie funkcionality. Jednou z nich je napríklad možnosť definície vlastných pomocných metód, ktoré je možné volať v šablóne a tým môžu zjednodušiť generovanie obsahu [9].

### 3.4.4 Visual Studio Code[9]

Ako vývojové prostredie bolo zvolené Visual Studio Code. Jedná sa o textový editor zdrojových kódov vytvorený spoločnosťou Microsoft. Jeho výhodou je, že je veľmi nenáročný na systém oproti rozsiahlejším a ťažkopádnejším IDE[10], ktoré sa príliš pri vývoji webových aplikácií nepoužívajú. Ďalšou výhodou je, že má integrované nástroje pre JavaScript, TypeScript a Node.js, ktoré sú pri tomto projekte potrebné a dostačujúce. Navyše je možné vďaka rozsiahlemu systému doplnkov hocikedy rozšíriť tento editor o ďalšie nástroje [12].

### 3.4.5 ESLint[11] a Prettier[12]

Pre lepšiu čitateľnosť a vzhľad kódu bola použitá kombinácia nástrojov ESLint a Prettier. Nástroj ESLint pridáva do prostredia Visual Studio Code statickú analýzu typov pre TypeScript, a tak isto pridáva aj nápovedy pre automatické dopĺňanie častí kódu, ako sú napríklad typy alebo mená premenných.

Nástroj Prettier sa zasa stará o vizuálnu stránku kódu. Je možné si v ňom definovať rôzne formátovacie pravidlá, ktoré majú byť dodržiavané naprieč všetkými zdrojovými textami v konkrétnom programovacom jazyku. Potom už je len jednoducho spustené formátovanie súboru a ten je správne transformovaný, aby vyhovoval definovaným pravidlám. V týchto pravidlách môže byť špecifikované napríklad používanie medzier alebo tabulátorov, písanie bodkočiarok za jednotlivými výrazmi v kóde, či používanie jednoduchých alebo dvojitých úvodzoviek.

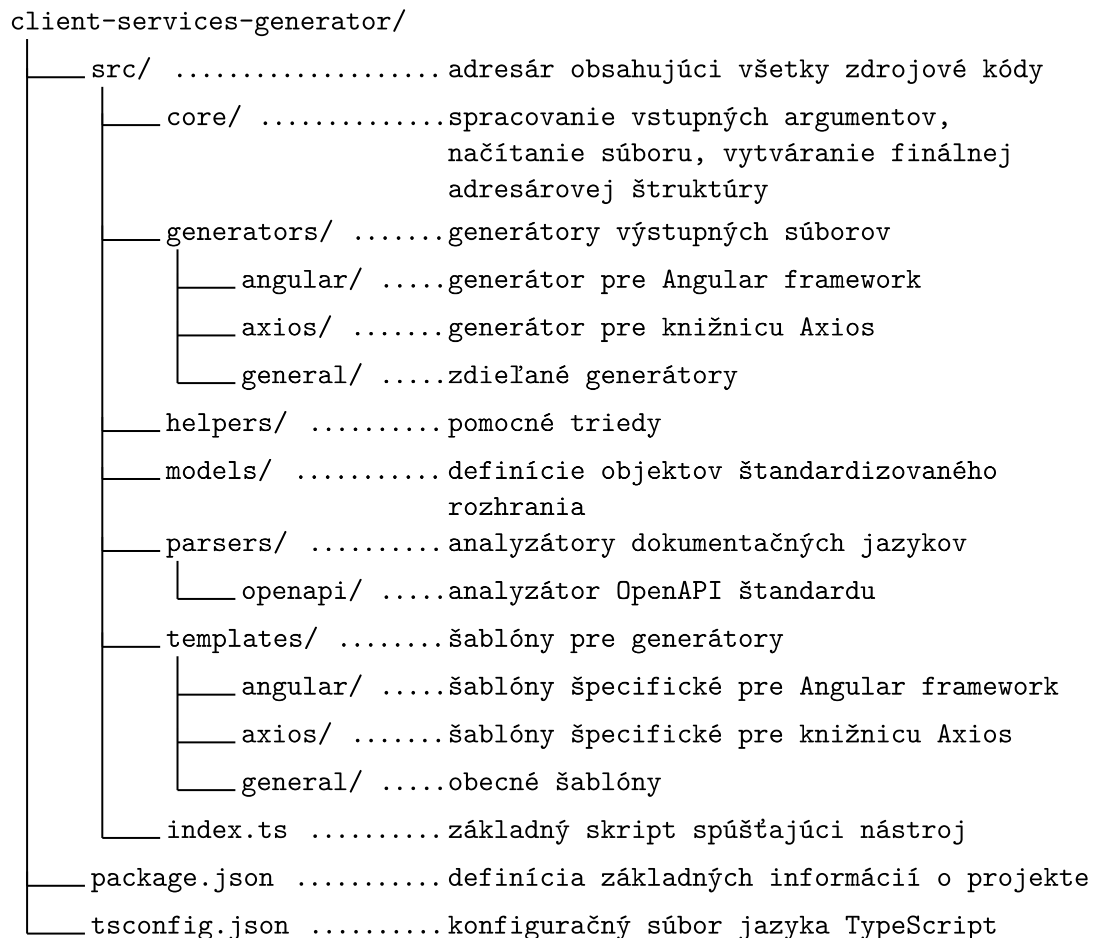
[5] https://www.typescriptlang.org/
[6] https://handlebarsjs.com/
[7] https://www.npmjs.com/package/handlebars
[8] https://mustache.github.io/
[9] https://code.visualstudio.com/
[10] **IDE** je skratka z anglického **I**ntegrated **D**evelopment **E**nvironment (vývojové prostredie) - jedná sa o komplexný program, ktorý obsahuje okrem textového editoru aj iné nástroje, ako prekladač pre určité jazyky a iné nástroje potrebné pri vývoji.
[11] https://eslint.org/
[12] https://prettier.io/

# Kapitola 4

# Implementácia riešenia

Nasledujúca kapitola sa zaoberá výslednou implementáciou nástroja. Najprv bude priblížená štruktúra projektu a formát zdrojových textov, ako aj ich logické členenie. Následne bude popísaná implementácia a princíp analýzy vstupného dokumentačného súboru. Na záver tejto kapitoly bude objasnená implementácia a spôsob generovania výslednej kostry aplikácie pomocou šablón.

## 4.1 Štruktúra a rozdelenie zdrojových textov riešenia



Toto je v skratke popísaná výsledná štruktúra riešenia projektu. Adresárová štruktúra je rozdelená podľa jednotlivých funkčných celkov systému a zodpovedá návrhu architektúry nástroja. Obsah jednotlivých adresárov bude priblížený v nasledujúcich podkapitolách.

Keďže bol ako jazyk implementácie zvolený práve TypeScript, sú zdrojové texty písané s využitím potenciálu tohto jazyka, čiže zdrojové texty sa skladajú z rozhraní, tried a metód. Jedinou výnimkou z použitia triedy pre zapuzdrenie funkcie (metódy) je v prípade základného skriptu, ktorý spúšťa nástroj. Funkcia obsiahnutá v tomto skripte je spúšťaná priamo v Node.js prostredí a keď bola obsiahnutá v triede dochádzalo k problémom so spustením. Každý súbor obsahuje práve jednu triedu, čo umožňuje väčšiu prehľadnosť projektu aj za cenu väčšieho počtu súborov. V prípade adresáru *models*, obsahuje každý súbor ešte definíciu rozhrania pre možnosť použiť niektorý z modelov ako parameter metódy. Trieda je členená tak, že na začiatku sa nachádzajú atribúty triedy, po ktorých nasleduje konštruktor danej triedy. Po konštruktore nasledujú verejné metódy a po nich neverejné metódy. Tento prístup som zvolil pre väčšiu prehľadnosť, keďže po otvorení súboru programátor vidí najprv metódy, ktoré sú dostupné aj mimo triedy a až po bližšom skúmaní triedy vidí metódy, ktoré slúžia pre vnútornú funkcionalitu triedy.

## 4.2 Spracovanie vstupných argumentov

Pre prácu so vstupnými argumentami bola vytvorená trieda `ArgumentParser`, ktorá sa nachádza v adresáre *core*. Táto trieda obsahuje jedinú verejnú metódu `parse()`, ktorá je volaná hneď po spustení nástroja. Po jej zavolaní sa spustí analýza zadaných argumentov. Túto analýzu a spracovanie sprostredkúva knižnica *yargs*[1]. Táto knižnica okrem spracovania argumentov generuje z množiny povolených argumentov aj prehľadnú nápovedu v prípade, že boli zadané nevalidné argumenty alebo argument „`-h`“.

Výsledný nástroj je možné spustiť dvoma spôsobmi. Jedným z nich je priame zadanie argumentov:

- „`-s`“ - cesta k súboru s dokumentáciou (tento argument je povinný)
- „`-t`“ - typ výslednej štruktúry (Angular alebo Axios)
- „`-o`“ - cesta k adresáru, kde budú umiestnené vygenerované súbory

```
$ client-services-generator -s ./apiDocs/openApi.json -t axios -o ./myApp
```

Výpis 4.1: Príklad spustenia nástroja.

Druhou možnosťou je zadanie prepínača „`-i`“ alebo plnej verzie „`--interactive`“. Vtedy sa v termináli spustí nástroj a bude postupne vyzývať užívateľa o zadanie argumentov. Jedná sa o tzv. CLI[2], čo je vlastne jednoduché užívateľské rozhranie v termináli. Pre vygenerovanie CLI a spracovanie užívateľom zadaných hodnôt je použitá knižnica *inquirer*[3]. Pri tomto prístupe je možné hneď po zadaní napríklad cesty k súboru s dokumentáciou overiť, či je zadaná cesta validná a v prípade, že nie je, požiadať užívateľa o zadanie platnej cesty k dokumentačnému súboru.

[1] https://www.npmjs.com/package/yargs
[2] **CLI** je skratka z anglického **C**ommand **L**ine **I**nterface.
[3] https://www.npmjs.com/package/inquirer

## 4.3 Systém analýzy vstupného súboru s dokumentáciou API

Jednou z dvoch najdôležitejších častí nástroja je analýza vstupného súboru. Podporované sú štyri formáty súborov: YAML a JSON pre OpenAPI dokumentácie, WADL a XML pre WADL dokumentácie. Vďaka tomu, že sa tieto formáty líšia štruktúrou aj príponou súboru, nie je nutné pri spúšťaní nástroja špecifikovať či sa v odkazovanom súbore nachádza dokumentácia typu OpenAPI alebo WADL.

### 4.3.1 Načítanie obsahu vstupného súboru

Na získanie obsahu vstupného súboru slúži trieda **`InputFileResolver`**, ktorá sa nachádza v adresáre *core*. Táto trieda obsahuje verejnú metódu **`getRawApiData(path)`**, ktorá má ako parameter cestu k vstupnému súboru. Následne je tento súbor po zistení formátu konvertovaný na JSON formát. Tento formát bol zvolený, pretože sa z neho dá jednoducho v TypeScripte spraviť objekt a potom je možné pristupovať k jeho jednotlivým častiam ako v prípade hocijakého iného objektu, čo výrazne uľahčuje prácu s týmto súborom. Nevýhodou však je, že tento vytvorený objekt nemá žiadne typy ani definované rozhranie, preto je nutné, aby jeho obsah bol štandardizovaný, čo v prípade OpenAPI aj WADL je splnené, keďže sa jedná o dokumentačné jazyky, ktoré majú definovanú štruktúru. Návratovou hodnotou metódy **`getRawApiData`** je teda objekt, ktorý obsahuje dáta získané zo vstupného súboru.

### 4.3.2 Analýza obsahu vstupného súboru

Pre analýzu objektu získaného zo vstupného súboru slúži trieda **`OpenApiParser`**, ktorá sa nachádza v adresári *parsers/openApi*. Obsahuje jedinú verejnú metódu `parse(apiDoc)`, v ktorej sa objekt obsahujúci dokumentáciu (`apiDoc`) postupne analyzuje pomocou ďalších analyzátorov, ktoré budú popísané nižšie. `OpenApiParser` sa využíva aj pre analýzu WADL dokumentácie, ktorá je pred analýzou transformovaná z WADL jazyka do OpenAPI jazyka pomocou knižnice *api-spec-converter*[4]. Tento spôsob bol zvolený na základe testov, ktoré odhalili problémy s konverziou XML formátu do JSON, a tak isto s určitou neštandardnosťou jazyka WADL.

#### Analýza základnej URL servera

```
"servers" : [ {
    "url" : "https://virtserver.swaggerhub.com/StazriN/Test3.0/1.0.0",
    "description" : "SwaggerHub API Auto Mocking"
}, {
    "url" : "https://petstore.swagger.io/v2"
} ],
```

Výpis 4.2: Ukážka časti dokumentácie popisujúcej adresy serverov.

Ako prvá je analyzovaná tá časť dokumentácie, ktorá definuje URL serverov, na ktorých sa nachádza zdokumentované rozhranie. Na analýzu slúži trieda **`ServerInfoParser`** a v nej nachádzajúca sa metóda `parseUrl(openApiServers)`. Táto metóda má ako parameter pole objektov, ktoré v sebe obsahujú URL serveru. Toto pole je extrahované z objektu, ktorý

[4] https://www.npmjs.com/package/api-spec-converter

bol získaný v predchádzajúcom kroku. Návratová hodnota tejto metódy je pole obsahujúce adresy serverov. Aj keď vo väčšine prípadov je API umiestnené na jednom serveri, OpenAPI dokumentácia povoľuje viacero adries k serverom, napríklad jedna je určená pre vývoj aplikácie a druhá pre produkčné prostredie. Na túto skutočnosť je teda v analyzátore braný ohľad a spracúvajú sa všetky dostupné URL adresy serverov.

### Analýza modelov

```
"components" : {
    "schemas" : {
        "Category" : {
            "type" : "object",
            "properties" : {
                "id" : {
                    "type" : "integer",
                    "format" : "int64"
                },
                    "name" : {
                    "type" : "string"
                }
            },
        },
        ...
```

Výpis 4.3: Ukážka časti dokumentácie popisujúcej jednoduchý model.

Ďalšou analyzovanou časťou dokumentácie sú modely. Modely sú definície tried, ktoré sa používajú pri komunikácii v API. Niektoré API nevyužívajú modely a komunikujú len za použitia jednoduchých typov ako reťazec, číslo alebo pravdivostná hodnota. Väčšinou sa však modely v API využívajú. Pre analýzu a spracovanie týchto modelov do štandardizovaného rozhrania slúži trieda `ModelParser`, ktorá obsahuje metódu `parse(openApiSchemas)`. Metóda ako parameter berie pole objektov z dokumentácie, ktoré obsahujú informácie o jednotlivých modeloch. Pri analýze modelu sa najprv kontroluje dedičnosť. Ak model dedí z nejakej nadradenej triedy, zahrnie sa táto trieda ako `superType` do objektu reprezentujúceho model v štandardizovanom rozhraní. Následne sa analyzujú jednotlivé atribúty modelu. Každý atribút má nejaký typ, ktorý môže byť jedeným zo základných typov TypeScriptu alebo to môže byť ďalší model.

Medzi modelmi sa nachádzajú aj konečné množiny definovaných hodnôt (enumy), ktoré sú však analyzované pomocou triedy `EnumParser`. Aj keď sa nachádzajú v rovnakej časti ako modely ich štruktúra je odlišná, a preto je ich analýza oddelená.

Po sekcii s modelmi sa niekedy v dokumentácii nachádza sekcia „*RequestBodies*“, ktorá obsahuje definície štruktúr tiel požiadaviek (request body). Táto sekcia slúži podobne ako sekcia s modelmi na definovanie štruktúr, ktoré sa využívajú viackrát v rámci API, a preto je táto štruktúra extrahovaná mimo definície zdrojov na jedno miesto, aby nevznikali duplicitné definície. Pre analýzu tejto sekcie slúži trieda `RequestBodyParser`. Práve kvôli tejto sekcii vznikol mierny odklon od návrhu a to pridaním atribútu **requestBodies** do triedy **Application** 3.3.1. Tento atribút obsahuje práve jednotlivé definície štruktúr, ktoré sú typu **RequestObject** 3.3.7.

**Analýza zdrojov**

```
"paths" : {
    "/user/{userId}" : {
        "get" : {
            "summary" : "Returns a~user name.",
            "description" : "Optional extended description.",
            "operationId" : "geUserName",
            "parameters" : [ {
                "name" : "userId",
                "in" : "path",
                "description" : "ID of user",
                "required" : true,
                "schema" : {
                  "type" : "integer"
                }
              } ],
            "responses": {
                "200" : {
                    "description" : "Successful operation",
                    "content" : {
                        "application/json" : {
                            "schema" : {
                                "type" : "string",
                            }
                        }
                    }
                }
            }
        }
    }
...
```

Výpis 4.4: Ukážka časti dokumentácie popisujúcej jednoduchý zdroj.

Poslednou analyzovanou časťou dokumentácie je časť obsahujúca informácie o jednotlivých zdrojoch REST API. Pre analýzu tejto časti dokumentácie slúži trieda **`PathParser`**, ktorá obsahuje verejnú metódu `parse(openApiPaths)`. Táto metóda má ako parameter objekt obsahujúci informácie o všetkých zdrojoch nachádzajúcich sa v danom REST API. Tieto zdroje sú následne postupne analyzované. Najprv sa zistí aké metódy obsahuje (GET, POST, PUT, ...). Z každej tejto metódy sa získava jej názov, ktorý je povinný a v prípade, že metóda tento názov neobsahuje, jedná sa o nevalidnú dokumentáciu. Ignoruje sa však len táto metóda a analýza pokračuje ďalej, aby neboli zahodené validné metódy.

Následne sa z metódy získavajú informácie či podporuje, prípadne vyžaduje, nejaké parametre pri jej volaní. Z parametru je potrebné zistiť jeho názov, umiestnenie (súčasť URI, v hlavičke požiadavky, ...), typ, štruktúru a či je povinný pri volaní metódy.

Ďalšou časťou, ktorá sa získava z metódy je štruktúra tela požiadavky. Táto časť sa nachádza väčšinou len pri metódach PUT, POST a PATCH. U metód GET a DELETE nedáva táto časť veľmi zmysel, ale HTTP dovoľuje tento obsah aj pri týchto metódach,

takže sa štruktúra tela požiadavky zisťuje u každej metódy, kde je definovaná. Pre analýzu štruktúry tela požiadavky sa využíva trieda **RequestBodyParser**. U tejto štruktúry sa zisťuje či je typ jedeným zo základných typov jazyka TypeScript alebo je definovaný v rámci modelov, či je povinná pri volaní metódy, či jej štruktúra je pole alebo enum a či sa nenachádza v sekcii dokumentácie *RequestBodies*.

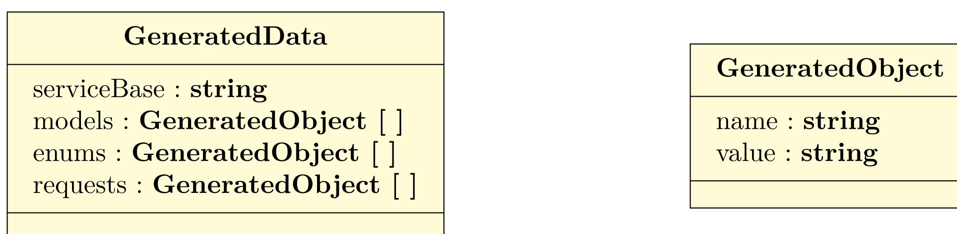
Poslednou časťou, ktorá sa získava z metódy je štruktúra odpovede zo serveru na volanie tejto konkrétnej metódy. Táto časť metódy je taktiež analyzovaná pomocou triedy **RequestBodyParser**, z dôvodu podobnej štruktúry ako má telo požiadavky. Jediným rozdielom oproti štruktúre tela požiadavky je, že štruktúra odpovede môže mať viacero typov a konkrétny návratový typ môže závisieť napríklad od toho či daná metóda skončila úspešne alebo neúspešne, alebo od hodnoty parametra, ktorý bol poslaný v rámci metódy.

## 4.4 Generovanie výstupných súborov a použitie šablón

Druhou najdôležitejšou časťou nástroja je generovanie výstupnej kostry aplikácie. Po spracovaní vstupného súboru je na základe zvoleného typu výstupu (Axios[5] alebo Angular) vygenerovaná výsledná štruktúra. Na generovanie výstupu pre framework Angular slúži trieda **AngularServiceGenerator**, ktorá obsahuje metódu `generate(applicationData)`. Táto metóda má ako parameter práve objekt typu `Application` 3.3.1, ktorý odpovedá návrhu komunikácie medzi analyzátorom a generátorom z kapitoly 3. Trieda, ktorá slúži na generovanie výstupu pre knižnicu Axios sa nazýva **AxiosServiceGenerator** a rovnako obashuje metódu `generate(applicationData)`, aby bolo zachované jednotné rozhranie.

### 4.4.1 Princíp a postup generovania výstupu

Triedy **AngularServiceGenerator** a **AxiosServiceGenerator** majú podobný princíp generovania výslednej kostry, preto bude popísaná len trieda `AngularServiceGenerator`. Výstupom oboch tried je objekt `GeneratedData`, ktorý obsahuje reťazce obsahujúce jednotlivé časti výslednej kostry aplikácie. Tieto jednotlivé časti sú generované do súborov.




Obr. 4.1: Trieda `GeneratedData`, ktorú vracajú generátory a trieda `GeneratedObject` obsahujúca atribút `name`, ktorý definuje názov súboru a atribút `value`, ktorý definuje obsah tohto súboru.

#### Generovanie základnej triedy pre volanie API

Najprv je triedou **ServiceBaseGenerator** vygenerovaný obsah atribútu **servicBase**. Vygenerovaná trieda **ServiceBase** bude obsahovať definovanie základnej URL serveru získa-

[5] `https://www.npmjs.com/package/axios` - jedná sa o jednu z najznámejších a najpoužívanejších knižníc pre komunikáciu medzi klientom a serverom.

nej z dokumentácie, nastavenie hlavičiek požiadaviek (napríklad nastavenie autorizačného kľúča), odchytávanie a formátovanie chýb, ktoré vzniknú pri posielaní požiadavky alebo pri prijímaní odpovede zo serveru.

### Generovanie modelov

Nasleduje generovanie modelov pomocou triedy `ModelGenerator`, ktorá vygeneruje jednotlivé modely do reťazcov. Tieto reťazce sú obsiahnuté v objektoch `GeneratedObject`. Postup generovania jednotlivých modelov pomocou šablón je priblížený v podkapitole 4.4.2. Po modeloch sú vygenerované v dokumentácii definované konečné množiny definovaných hodnôt (enumy). Pre generovanie enumov slúži trieda `EnumGenerator`. Šablóna pre enum je veľmi jednoduchá a obsahuje v podstate len cyklus, v ktorom sú postupne generované všetky hodnoty, ktoré môže daný enum nadobúdať. Podobne ako pri modeloch sú jednotlivé enumy vygenerované do reťazcov a tie sú následne uložené do objektov `GeneratedObject`.

### Generovanie požiadaviek

Ako posledné sú generované triedy, ktoré obsahujú metódy reprezentujúce jednotlivé požiadavky na konkrétne REST API zdroje. Trieda, ktorá slúži na generovanie týchto tried je `RequestGenerator`. Každá generovaná trieda obsahujúca požiadavky je generovaná po častiach, pretože do jednej triedy sa vkladajú všetky volania API nad konkrétnym zdrojom a aj jeho podzdrojmi. Metódy volajúce API sú teda zoskupené do tried a súborov podľa základného zdroja v URI. Tento prístup bol zvolený kvôli väčšej prehľadnosti, keďže všetky metódy, ktoré je možné nad daným zdrojom volať sa nachádzajú v jednom súbore, resp. v jednej triede. Generovanie po častiach je nutné, pretože nie vždy sa všetky možnosti volania konkrétneho zdroja a jeho podzdrojov nachádzajú v dokumentácii pokope a dodatočná úprava súborov by bola náročná vzhľadom k tomu, že by sme museli uchovávať veľa informácií o pozíciach jednotlivých častí triedy, aby bolo možné vložiť novú metódu na správne miesto do triedy. Časti (šablóny), na ktoré je výsledná trieda rozdelená sú *imports* (referencie na modely použité v metódach), *methods* (jednotlivé metódy volajúce API) a samotná trieda *requestClass*, do ktorej budú predchádzajúce časti nakoniec vložené.

Do časti *imports* sú postupne vkladané referencie na modely, ktoré sa používajú v metódach pre volanie API. Pri vkladaní referencie je nutné overovať či sa tam už daná referencia nenachádza, pretože viacnásobná referencia na ten istý model by v jazyku TypeScript znamenala chybu.

Do časti *methods* sú vkladané jednotlivé metódy, ktoré je možné volať na zdrojom a jeho podzdrojmi. Vygenerovanie takejto metódy je pomerne zložité a je nutné kontrolovať viacero faktorov. Najprv sa generuje meno metódy. Následne sa generujú parametre, ktoré budú použité pri požiadavke, či už ako telo tejto požiadavky alebo parametre nachádzajúce sa v URI. U parametrov je vyhodnotené či sú z pohľadu požiadavky povinné alebo voliteľné a podľa toho upravený ich zápis. Ako ďalší krok je vygenerovaný typ (štruktúra), prípadne typy odpovedí na požiadavku. Následne sú vygenerované kontrolné časti metódy, ktoré overujú či parameter obsahuje nejaké dáta a v prípade, že neobsahuje, nebude zahrnutý do výslednej požiadavky. Nakoniec je vygenerovaná samotná požiadavka, do ktorej sú doplnené všetky potrebné vlastnosti, ako typ metódy HTTP, hlavičky, telo požiadavky a parametre nachádzajúce sa v URI.

### 4.4.2 Použitie šablón

Pre generovanie jednotlivých častí výslednej kostry sú použité šablóny, napísané v jazyku Handlebars 3.4.3. Tento prístup je vhodnejší ako definovanie reťazcov, ktoré obsahujú jednotlivé časti a to najmä z dôvodu možnosti použitia rôznych pomocných funkcií a podmieňovacích výrazov vzhľadom na dáta, ktoré sa majú zobraziť v šablóne. Šablóny je možné samozrejme znovu použiť na viacerých miestach v nástroji, čiže riešia aj duplicity reťazcov. Taktiež je správa týchto šablón jednoduchšia a ich štruktúra ukazuje, ako bude vyzerať výsledná štruktúra v súbore.

V jazyku Handlebars sa volanie pomocných metód, podmieňovacie výrazy a premenné zapisujú do zdvojených zložených zátvoriek, prípadne do trojitých, ak dáta obsahujú znaky, ktoré sú v HTML považované za špeciálne a nie je možné ich zapísať priamo, ale len pomocou sekvencie (znaky ako <, >, ", ', &, ...). Ak by niektorá z premenných zapísaná len v zdvojených zložených zátvorkách obsahovala jeden zo špeciálnych znakov HTML, knižnica zodpovedná za vygenerovanie výstupu zo šablóny by takýto znak vygenerovala ako sekvenciu, ktorá je platná v HTML. Keďže sa však pomocou šablón generujú zdrojové texty v jazyku TypeScript, je niekedy potrebné zobraziť aj priamo znaky, ktoré sú klasifikované ako špeciálne v HTML.

```
{{#generateModelImports model}}
import {{name}} from './{{path}}';
{{/generateModelImports}}

export default class {{model.name}}
{{#if model.superType}} extends {{model.superType}}{{/if}} {

    {{#generateModelProps model.properties}}
    public {{name}}: {{{type}}};
    {{/generateModelProps}}

    {{#generateModelConstructor model}}
    public constructor({{{constructorParams}}}) {
        {{#if superConstructor}}
        super({{superConstructor}});
        {{/if}}

        {{#each constructorAssigns as |assign|}}
        this.{{assign}} = {{assign}};
        {{/each}}
    }
  {{/generateModelConstructor}}
}
```

Výpis 4.5: Šablóna pre generovanie modelu napísaná v jazyku Handlebars.

Na začiatku šablóny sa nachádza kontrolná sekcia **generateModelImports**, ktorá je volaním pomocnej metódy a ako parameter má objekt **model**. Táto metóda ukazuje možnosť jazyka Handlebars vytvárať užívateľsky definované pomocné metódy. Táto metóda zanalyzuje objekt `model`, či niektorý z jeho atribútov má typ, ktorý je iným modelom. V prípade nálezu takéhoto atribútu je potrebné vložiť odkaz na tento model, aby analyzátor jazyka

TypeScript vedel zistiť, o aký typ sa jedná a či jeho hodnota vyhovuje definícii. Metóda vracia dve hodnoty a to `name`, ktorá obsahuje meno modelu, ktorý je použitý ako typ pre nejaký atribút a `path`, ktorá obsahuje cestu k danému modelu v rámci adresárovej štruktúry výstupu.

Následne sa doplní do šablóny meno triedy a overuje sa či táto trieda dedí z inej a v tom prípade je doplnené kľúčové slovo `extends` a názov nadradenej triedy.

Ďalšia pomocná metóda `generateModelProps`, slúži na generovanie jednotlivých atribútov danej triedy. Metóda má ako parameter pole atribútov získaných z dokumentácie a vracia dve hodnoty a to `name`, ktorá obsahuje názov atribútu a `type`, ktorá obsahuje typ daného atribútu. Práve premenná `type` je v trojitých zložených zátvorkách, pretože môže obsahovať jednoduché alebo dvojité úvodzovky, ktoré sú v HTML špeciálnym znakom a boli by zobrazené pomocou sekvencie.

Pomocná metóda `generateModelConstructor` slúži na generovanie konštruktora danej triedy. Ako parameter má táto metóda objekt `model`. Metóda vracia reťazec obsahujúci jednotlivé parametre konštruktora vychádzajúce z atribútov danej triedy, prípadne aj nadradenej triedy. U parametrov je vyznačené či sú povinné pre vytvorenie objektu alebo nie. Následne sa kontroluje či daná trieda dedí z inej triedy a v tom prípade je vygenerované volanie konštruktora `super()` s potrebnými parametrami pre túto nadradenú triedu. Nakoniec sú v cykle pomocou kľúčového slova `each` vygenerované priradenia parametrov konštruktora k atribútom daného modelu.

Po skončení celého tohto procesu je vygenerovaný reťazec obsahujúci model zo všetkými vyplnenými dátami. Tento model je následne vložený do poľa `generatedModels` a po vygenerovaní všetkých ostatných častí výslednej kostry bude tento model spolu s ďalšími časťami vygenerovaný do súboru.

### 4.4.3 Generovanie výslednej adresárovej štruktúry

Po vygenerovaní jednotlivých častí výslednej kostry je potrebné vygenerované reťazce zapísať do súborov a vytvoriť správnu adresárovú štruktúru. Pre tento účel slúži trieda `OutputFilesResolver`, ktorá obsahuje metódu `generateOutput(path, genResult)`. Táto metóda má ako parameter cestu k adresáru, ktorá bola zadaná ako vstupný argument a objekt `genResult`, ktorý obsahuje všetky vygenerované modely, triedy a metódy. Ako prvé sa overí či v zadanom adresáre už neexistuje adresár *services* a v prípade, že existuje, je na rozhodnutí užívateľa či chce celý obsah tohto adresára odstrániť a vygenerovať nový alebo ukončiť nástroj. Následne sa vytvorí v tomto adresári nový súbor *serviceBase.ts*, ktorý obsahuje triedu `ServiceBase`. Potom sa vytvorí adresár *models*, do ktorého sú následne vygenerované súbory obsahujúce definície jednotlivých modelov. Ako posledné sú vygenerované do adresára *requests* súbory obsahujúce triedy s metódami, ktoré obsahujú volanie jednotlivých HTTP metód nad REST API zdrojmi. Po vygenerovaní všetkých adresárov a súborov je nástroj ukončený.

# Kapitola 5

# Testovanie a nasadenie nástroja

Žiadny softvér nie je bezchybný, a preto je ho potrebné pred publikovaním dostatočne otestovať a pokúsiť sa odhaliť a následne odstrániť čo najviac chýb. Ak by bol proces testovania zanedbaný, publikovaný softvér s veľa chybami pôsobí neprofesionálne a odradí potencionálnych užívateľov od jeho používania.

V prvej časti tejto kapitoly budú popísané jednotlivé typy testov, ich priebeh a výsledky. V druhej časti bude popísaný proces publikovania nástroja do verejného úložiska balíčkov pre vývoj webových aplikácií.

## 5.1 Testovanie

Najdôležitejším cieľom testovania je overiť či nástroj spĺňa všetky stanovené požiadavky. Ďalšími cieľmi sú overenie spoľahlivosti, funkčnosti a užívateľskej prívetivosti nástroja.

### 5.1.1 Jednotkové testy

Ako prvé boli už počas vývoja použité jednotkové testy. Tieto testy sa zameriavajú na jednotlivé časti nástroja. Dôležité bolo najmä testovanie rôznych možností jednotlivých častí vstupnej dokumentácie. Pri tomto testovaní bolo potrebné pracovať so špecifikáciou OpenAPI 2.2.4, aby bolo možné pokryť čo najviac možností, ktoré sa môžu vyskytnúť v reálnych dokumentáciách rozhraní. Niektoré chyby súvisiace s dokumentačným jazykom WADL sa však nepodarilo úplne odstrániť, najmä kvôli nie úplne štandardizovanej štruktúre dokumentácií napísaných v jazyku WADL. Pre väčšinu dokumentácií je však nástroj plne funkčný. Pomocou týchto testov bolo možné zachytiť potencionálne chyby pri analýze vstupného súboru už pri procese vývoja nástroja.

### 5.1.2 Integračné testy

Dôležitou časťou sú aj integračné testy, pri ktorých bola overená správnosť komunikácie medzi analyzátorom a generátorom pomocou navrhovaného štandardizovaného rozhrania. Pri týchto testoch boli odhalené niektoré chýbajúce informácie, ktoré bolo potrebné ešte získať zo vstupného súboru. Následne bolo mierne poupravené štandardizované rozhranie 3.3.

### 5.1.3 Systémové testy

Asi najdôležitejšia testovacia časť sú práve systémové testy. Tieto testy slúžia na overenie funkcionality nástroja ako celku. Pri týchto testoch boli odhalené viaceré nedostatky, ktoré bolo potrebné opraviť. Najprv boli odhalené nedostatky pri načítavaní vstupného súboru a odchytávanie chýb, ktoré nastali pri zadaní nevalidnej cesty k súboru s dokumentáciou alebo pri načítaní poškodeného či neplatného súboru. Pri analyzátore a generátore nebolo odhalených veľa chýb vďaka predchádzajúcim testom. Ďalšie chyby boli odhalené pri generovaní reťazcov do súborov, ale jednalo sa skôr o chyby v štruktúre výsledných súborov.

Veľmi dôležité testy sa týkali použitia vygenerovanej kostry aplikácie v testovacích webových aplikáciách. Tie odhalili rôzne štrukturálne nedostatky, najmä v prípade kostry pre framework Angular, s ktorým som nemal veľa skúseností. Tieto nedostatky boli následne odstránené, aby bola kostra v aplikáciách postavených na frameworku Angular plne funkčná.

### 5.1.4 Testovanie s užívateľmi

Keďže výsledným riešením tejto práce je nástroj určený pre vývojárov webových aplikácií, je skupina užívateľov, s ktorými je možné nástroj testovať užšia. Nástroj taktiež nemá nejaké zložité alebo komplexné užívateľské rozhranie, keďže sa jedná len o konzolový nástroj, takže pri tomto nástroji nedochádza ani k veľkej užívateľskej interakcii. Nástroj testovali šiesti vývojári webových aplikácií. Napriek tomuto malému počtu užívateľov, ktorí tento nástroj testovali, mali zopár trefných pripomienok, z ktorých bola väčšina zapracovaná do výsledného riešenia.

- ***Odstrániť nutnosť špecifikovať typ dokumentácie na vstupe*** - Pôvodne nástroj požadoval špecifikovanie typu dokumentácie, ktoré však bolo odstránené kvôli ľahko rozoznateľnej odlišnej štruktúre, ako aj prípone súboru
- ***Odstrániť nutnosť špecifikovať adresár pre výstup*** - Očakáva sa, že vo väčšine prípadov bude nástroj spúšťaný práve z adresáru, kde sa nachádza aj webová aplikácia, preto je argument nástroja slúžiaci pre definovanie výstupného adresáru predvyplnený a obsahuje cestu k aktuálnemu adresáru, pokiaľ užívateľ nezadá explicitne iný adresár
- ***Pridať potvrdenie zmazania obsahu adresára „services“*** - Pôvodne nástroj po zistení, že výstupný adresár obsahuje adresár „*services*“ obsah tohto adresára odstránil a vygeneroval nový obsah podľa nového vstupného súboru. Po užívateľských testoch bola pridané potvrdenie či chce užívateľ naozaj odstrániť obsah adresára

Po publikovaní však užívateľské testovanie naďalej pokračuje a nájdené chyby je možné nahlásiť a zaznamenať do systému *GitHub Issues*[1]. V čase písania tejto práce má nástroj viac ako 200 stiahnutí a zatiaľ nebola nahlásená žiadna chyba, čo môže vypovedať o dobrom testovaní alebo o neochote užívateľov nahlasovať chyby. Ak bude počet užívateľov nástroja narastať, verím, že sa objavia nové chyby, ktoré sa nepodarilo zachytiť pri testovaní.

[1] https://github.com/StazriN/client-services-generator/issues

## 5.2 Nasadenie nástroja

Výsledný nástroj bol publikovaný ako balíček do najväčšieho úložiska balíčkov pre vývoj webových aplikácií **npm**[2]. Pre možnosť publikovať nástroj do tohto úložiska je potrebné mať účet v tejto službe. Pred publikovaním je potrebné do súboru „*package.json*“ špecifikovať názov, licenciu a verziu, pod ktorými bude balíček publikovaný. Pre názvy balíčkov existujú určité odporúčania a obmedzenia[15]:

- Meno balíčku musí byť unikátne
- Meno sa nesmie podobať názvu iného balíčka
- Meno balíčka nesmie obsahovať žiadne urážlivé alebo nevhodné slová
- Malo by vhodne popisovať balíček, aby bolo jasné k čomu slúži
- Nesmie sa jednať o balíček, ktorý už niekto vlastní

Ďalej je potrebné vložiť do balíčku súbor „*README*“, ktorý obsahuje informácie o tom, ako balíček spustiť a ako s ním pracovať. Je vhodné, aby balíček obsahoval odkaz na repozitár, v ktorom sa nachádzajú jeho zdrojové kódy. Tak isto je vhodné doplniť aj vhodné kľúčové slová, ktoré pomôžu pri vyhľadávaní balíčka.

Pred publikovaním je vhodné balíček otestovať pomocou príkazu **`npm install`** a zadaním cesty k novému balíčku, ktorý sa nachádza na disku. Je tak isto vhodné preveriť či balíček neobsahuje nejaké citlivé informácie, ktoré mohli zostať v balíčku po testovaní.

Následne je už len pomocou príkazu **`npm publish`** výsledný balíček publikovaný do úložiska balíčkov [14].

Výsledný balíček je možné stiahnuť a nainštalovať do projektu pomocou príkazu:

```
$ npm install client-services-generator --save-dev
```

Prípadne pre nainštalovanie balíčka globálne pomocou príkazu:

```
$ npm install client-services-generator -g
```

Detailnejší popis inštalácie a možností spustenia nástroja je popísaný v prílohe A.

[2] https://www.npmjs.com/

# Kapitola 6

# Záver

Cieľom tejto práce bolo vytvoriť nástroj, ktorý umožní generovanie základnej kostry webovej aplikácie z dokumentácie REST rozhrania. Podarilo sa splniť všetky špecifikované požiadavky a výsledný nástroj sa nachádza v najväčšom úložisku balíčkov pre vývoj webových aplikácií a je dostupný na adrese https://www.npmjs.com/package/client-services-generator. V čase písania tejto práce má nástroj približne dva mesiace od zverejnenia viac ako 200 stiahnutí z tohto úložiska.

Najprv bolo potrebné analyzovať dostupné dokumentačné jazyky pre popis REST rozhraní a z týchto jazykov vybrať tie, ktoré sú najpoužívanejšie a najrelevantnejšie. Následne bolo potrebné sa oboznámiť s frameworkom Angular, pretože som s ním doteraz nemal veľa skúseností. Po pochopení požiadaviek, ktoré vyžaduje či už framework Angular alebo knižnica Axios, som navrhol nástroj pre zautomatizovanie procesu vývoja kostry webovej aplikácie pre komunikáciu medzi serverom a klientom. Následne bolo navrhnuté štandardizované rozhranie, aby bol nástroj čiastočne modulárny a bolo možné jeho rozširovanie o ďalšie analyzátory dokumentačných jazykov alebo pridanie nových generátorov pre ďalšie webové frameworky a knižnice. Počas implementácie a aj po nej bolo vykonaných viacero testov, aby sa overila použiteľnosť nástroja a až po úpravách bol nástroj následne publikovaný pre verejné použitie.

Ako vývojárovi webových aplikácií mi tento nástroj uľahčí vytváranie nových webových aplikácií a vďaka tejto práci som nadobudol nové znalosti o frameworku Angular a jeho výhodách pri komplexnejších a väčších webových aplikáciách.

Jedným z možných rozšírení nástroja by bolo pridanie ďalších už spomínaných analyzátorov aj pre menej používané dokumentačné jazyky, a tak isto pridanie generátorov a šablón pre menej známe frameworky a knižnice pre komunikáciu medzi klientom a serverom. Ďalším možným rozšírením by bolo predanie vstupného dokumentačného súboru pomocou URL odkazujúcou na súbor s dokumentáciou, aby nebolo potrebné tento súbor sťahovať a až následne predať nástroju. Toto rozšírenie by bolo veľmi vítané nakoľko z užívateľského testovania vyplynulo, že väčšina dokumentácií API serveru sa pre lepšiu užívateľskú vizualizáciu nachádza v online knižnici. Pre použitie nástroja je však potrebné dokumentáciu stiahnuť, čo môže pri častých zmenách spomaliť proces vývoja webovej aplikácie.

# Literatúra


[1] Belmont, J.-M. *API Workshop - API Description Languages* [online]. Marcelbelmont with GitBook, 20. apríla 2018 [cit. 2020-04-21]. Dostupné z: https://www.marcelbelmont.com/api-workshop/docs/api-description-languages.html.

[2] Dabbs, M. *The Fundamentals of Web Application Architecture* [online]. Reinvently, 28. júla 2019 [cit. 2020-04-13]. Dostupné z: https://reinvently.com/blog/fundamentals-web-application-architecture/.

[3] Development, A. *Introducing the Open API Initiative!* [online]. Swagger, 5. novembra 2015 [cit. 2020-04-21]. Dostupné z: https://swagger.io/blog/api-development/introducing-the-open-api-initiative/.

[4] Ed douibi, H. *Modeling Web APIs: your best choices* [online]. Modeling Languages, 2. januára 2016 [cit. 2020-04-21]. Dostupné z: https://modeling-languages.com/modeling-web-api-comparing/.

[5] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, California, USA, 2000. Dizertačná práca. University of California. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[6] Hadley, M. *Web Application Description Language* [online]. 1. vyd. November 2006 [cit. 2020-04-25]. Dostupné z: https://www.w3.org/Submission/wadl/.

[7] Jansen, G. *Resources* [online]. Geert Jansen, 15. novembra 2012 [cit. 2020-04-11]. Dostupné z: https://restful-api-design.readthedocs.io/en/latest/resources.html.

[8] Joshua S. Ponelat, L. L. R. *Designing APIs with Swagger and OpenAPI*. 1. vyd. Manning Publications Co., 2019. ISBN 9781617296284. Dostupné z: https://www.manning.com/books/designing-apis-with-swagger-and-openapi.

[9] Katz, Y. *Handlebars Language Guide* [online]. Handlebarsjs, 20. februára 2020 [cit. 2020-05-05]. Dostupné z: https://handlebarsjs.com/guide/.

[10] Lafon, Y. *Team Comment on the "Web Application Description Language"Submission* [online]. W3C, 14. októbra 2009 [cit. 2020-04-25]. Dostupné z: https://www.w3.org/Submission/2009/03/Comment.

[11] Malagi, S. *Server-Side Rendering VS. Client-Side Renderinge* [online]. Clarion, 2. januára 2018 [cit. 2020-04-13]. Dostupné z: https://www.clariontech.com/blog/server-side-rendering-vs.-client-side-rendering.

[12] Microsoft. *Documentation for Visual Studio Code* [online]. Microsoft, 14. apríla 2016 [cit. 2020-05-05]. Dostupné z: https://code.visualstudio.com/docs.

[13] Morris, S. *TECH 101: REACT JS VS ANGULAR—WHAT'S THE DIFFERENCE?* [online]. Skillcrush, 20. mája 2019 [cit. 2020-04-30]. Dostupné z: https://skillcrush.com/blog/react-js-vs-angular/.

[14] npm. *Creating and publishing unscoped public packages* [online]. npm, Inc., 08. novembra 2018 [cit. 2020-05-18]. Dostupné z: https://docs.npmjs.com/creating-and-publishing-unscoped-public-packages.

[15] npm. *Package name guidelines* [online]. npm, Inc., 11. novembra 2018 [cit. 2020-05-18]. Dostupné z: inurl:https://docs.npmjs.com/package-name-guidelines.

[16] Patel, P. *What exactly is Node.js?* [online]. Freecodecamp, 18. apríla 2018 [cit. 2020-05-05]. Dostupné z: https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/.

[17] RESTfulAPI.net. *REST API Tutorial* [online], 14. júna 2016 [cit. 2020-04-10]. Dostupné z: https://restfulapi.net/.

[18] Rozentals, N. *Mastering TypeScript*. 2. vyd. Packt Publishing, 2017. ISBN 9781786467485. Dostupné z: https://books.google.sk/books?id=81MoDwAAQBAJ.

[19] Vega, J. *Client-side vs. server-side rendering: why it's not all black and white* [online]. Freecodecamp, 28. februára 2017 [cit. 2020-04-13]. Dostupné z: https://www.freecodecamp.org/news/what-exactly-is-client-side-rendering-and-hows-it-different-from-server-side-rendering-bd5c786b340d/.

# Príloha A

# Návod na inštaláciu nástroja

V nasledujúcej prílohe je popísaný postup inštalácie nástroja. Tento návod je rovnako dostupný v angličtine na https://www.npmjs.com/package/client-services-generator, prípadne priamo v GitHub repozitáre[1]. Nástroj je možné nainštalovať dvomi spôsobmi.

## A.1 Globálna inštalácia

Prvým spôsobom je nainštalovať nástroj globálne, čo umožní jeho spúšťanie vo viacerých projektoch bez nutnosti opätovnej inštalácie. V tomto prípade sa nástroj inštaluje pomocou nasledujúceho príkazu:

```
$ npm install client-services-generator -g
```

Následne je možné spustiť nástroj **`client-services-generator`** priamo z terminálu. Príklad spustenia:

```
$ client-services-generator -s ./apiDocs/openApi.json -t axios -o ./myApp
```

## A.2 Lokálna inštalácia

Druhým spôsobom je inštalácia nástroja len v rámci jedného projektu. Inštalácia pri tejto možnosti prebehia pomocou príkazu:

```
$ npm install client-services-generator --save-dev
```

Pre jednoduché spustenie nástroja pri tomto type inštalácie je potrebné do `package.json` pridať nasledujúcu definíciu skriptu:

```
"scripts": {
 "nazov-skriptu":
 "client-services-generator -s ./apiDocs/openApi.json -t axios -o ./myApp",
},
```

[1] https://github.com/StazriN/client-services-generator#readme

V prvej časti je možné nazvať skript podľa vlastného uváženia (nie je potrebné, aby sa nazýval `client-services-generator`). V druhej časti je zadaný podobný príkaz ako v prípade globálnej inštalácie.

Tento definovaný skript je následne možné spustiť pomocou príkazu:

```
$ npm run nazov-skriptu
```

## A.3 Možnosti spustenia nástroja

V tabuľke nižšie sú popísané všetky dostupné prepínače nástroja. Prepínač „`-s / --source`“ a „`-i / --interactive`“ sa navzájom vylučujú, a teda nie je možné ich použiť naraz. Jeden z nich je však potrebný pre spustenie nástroja.

| **Prepínač** | **Popis** | **Zákl. hodnota** | **Typ hodnoty** |
|---|---|---|---|
| --version | Zobrazí číslo verzie nástroja. | | |
| -h / --help | Zobrazí nápovedu k nástroju. | | [boolean] |
| -i / --interactive | Spustí nástroj v CLI móde. | false | [boolean] |
| -s / --source | Cesta k dokumentácii API. | | [string] |
| -t / --type | Typ výstupných súborov. | axios | [axios, angular] |
| -o / --output | Cesta pre výstupné súbory. | ./ | [string] |

Tabuľka A.1: Možnosti prepínačov nástroja

## A.4 Vygenerovaná adresárová štruktúra

Ukážka adresárovej štruktúry vygenerovanej nástrojom **`client-services-generator`**. Názvy adresárov **services**, **models**, **requests** a súboru **serviceBase.ts** sa nemenia a sú v každom vygenerovanom výstupe. Názvy modelov a volaní jednotlivých metód nad API sa menia podľa názvov definovaných v dokumentácii.

```
vystupny-adresar/services
├── models/ ................. adresár obsahujúci vygenerované modely
│   ├── myEnum.ts ........... príklad vygenerovaného enumu
│   ├── myModel.ts .......... príklad vygenerovaného modelu
│   ├── ⋮
│   └── yourModel.ts ........ príklad vygenerovaného modelu
├── requests/ ............... adresár obsahujúci volanie jednotlivých
│   │                         metód nad zdrojmi
│   ├── myRequests.ts ....... príklad súboru obsahujúceho volanie metód
│   ├── ⋮
│   └── yourRequests.ts ..... príklad súboru obsahujúceho volanie metód
└── serviceBase.ts .......... základná trieda obsahujúca definíciu URL
                              serveru, odchytávanie chýb, atď.
```

# Príloha B

# Obsah priloženého CD

Priložené CD obsahuje nasledujúce adresáre a súbory:

- adresár **demo** - obsahuje adresáre:
  - adresár **demo_bin** - obsahuje preloženú demo aplikáciu.
  - adresár **demo_src** - obsahuje zdrojové texty demo aplikácie.
  - adresár **documentation** - obsahuje dokumentáciu API vo formáte OpenAPI.
- adresár **doc** - obsahuje technickú správu vo formáte PDF.
- adresár **doc_src** - obsahuje zdrojové texty technickej správy.
- adresár **tool_bin** - obsahuje preložené výsledné riešenie.
- adresár **tool_src** - obsahuje zdrojové texty výsledného riešenia.
- súbor **README.md** - obsahuje informácie o jednotlivých adresároch a pokyny pre preloženie riešenia vo formáte Markdown.
- súbor **README.pdf** - obsahuje informácie o jednotlivých adresároch a pokyny pre preloženie riešenia vo formáte PDF.