VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

FRONTEND PRO GENERÁTOR TESTOVACÍCH STRUKTUR

FRONT-END FOR GENERATOR OF STRUCTURED TEST DATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **TOMÁŠ VOSTŘEJŽ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ALEŠ SMRČKA, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Vostřejž Tomáš**
Program: Informační technologie
Název: **Frontend pro generátor testovacích struktur**
Front-End for Generator of Structured Test Data
Kategorie: Web

Zadání:

1. Nastudujte technologie pro tvorbu webových aplikací. Pozornost věnujte nástrojům a knihovnám pro prohlížení a editaci kódu.
2. Navrhněte aplikaci pro tvorbu testovacích dat stromových struktur (JSON, XML, apod.). Aplikace bude generovat testovací data na základě uživatelem dodaných reálných dat a uživatelem definovaných kritérií pro generování testovacích dat.
3. Implementujte navržený generátor jako webovou aplikaci.
4. Správnost funkcionality podpořte automatizovanými testy zahrnující testování webového rozhraní.

Literatura:

- Žára, O.: JavaScript - Programátorské techniky a webové technologie. Computer Press, 2015.
- Domovská stránka projektu Gestr v platformě Testos. https://pajda.fit.vutbr.cz/testos/gestr

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

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 31. října 2019

Abstrakt

Bakalářská práce se zabývá tvorbou webové aplikace, která umožňuje generování stromových datových struktur vhodných pro testování softwaru. Aplikace je rozdělena na klientskou a serverovou část. Klientská aplikace poskytuje uživateli grafické rozhraní, kde si vytvoří vstupní stromovou datovou strukturu ve formátu JSON nebo XML. Na ni aplikuje vhodné modifikace pro testování s možností dodání vlastních kritérií pro generování. Implementována je pomocí platformy Angular. Serverová aplikace slouží ke generování testovacích dat a nabízí seznam dostupných modifikací. Prostředí Node.js a framework Express jsou použity pro implementaci. Serverová aplikace poskytuje rozhraní REST. Produkční nasazení obstarává nástroj pro virtualizaci Docker. Obě aplikace jsou nasazeny na serverech Heroku.

Abstract

The bachelor's thesis deals with the creation of a web application that allows the generation of tree data structures suitable for software testing. The application is divided into client and server part. The client application provides the user with a graphical interface where he creates an input tree data structure in JSON or XML format. It applies appropriate modifications for testing with the possibility of supplying its own criteria for generation. It is implemented using the Angular platform. The server application is used to generate test data and offers a list of available modifications. The Node.js environment and the Express framework are used for implementation. The server application provides a REST interface. Production deployment is provided by the Docker virtualization tool. Both applications are deployed on Heroku servers.

Klíčová slova

Angular, REST API, Node.js, Express, Jasmine, Protractor, mutační testování, injekce chyb, JSON, XML, stromové datové struktury

Keywords

Angular, REST API, Node.js, Express, Jasmine, Protractor, mutation testing, fault injection, JSON, XML, tree data structures

Citace

VOSTŘEJŽ, Tomáš. *Frontend pro generátor testovacích struktur*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Frontend pro generátor testovacích struktur

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Tomáš Vostřejž
4. června 2020

Poděkování

Rád bych poděkoval vedoucímu práce, panu Ing. Aleši Smrčkovi, Ph. D., za inspiraci a čas věnovaný společným konzultacím.

Obsah

Kapitola 1

Úvod

Testování hraje důležitou roly při vývoji aplikací. Může mít ale několik různých účelů. Za podpory testů můžeme kontrolovat jednotlivé části systému, nebo systém jako celek. Další testy mohou měřit, jak velkou část z cílové funkcionality splňuje současný stav aplikace. Tato práce se zaměřuje na testování odolnosti systému proti neočekávaným vstupům.

Pro představu se můžeme podívat třeba na automat na kávu. Zvolíme si oblíbenou kávu, množství cukru a vhodíme mince. Počkáme, až automat práci dokončí a spokojeně odcházíme. Toto je ideální scénář a automat pracuje správně. Co se ale stane, když uživatel zmáčkne různé druhy kávy najednu? Nebo vhodí mince jiné měny, než je uvedena na automatu? A nebo zvolí kávu, zaplatí část částky a poté odejde. Ne vždy jdou věci podle plánu a to stejné platí i pro aplikace.

Cílem práce je vytvořit webovou aplikaci, která umožní zapsat textovou formou vhodná vstupní data, která přijímá reálný systém. Poté vybrané části textu pozměnit podle předem definovaný pravidel s možností přidání vlastních kritérií. Dalším krokem je vygenerování sady souborů. Každý soubor se liší od původního a testuje tak různé vstupní hodnoty, pro různé vstupy.

Kapitola 2 se zabývá analýzou požadavků projektu, jako kdo bude aplikci používat a co se očekává, že bude splňovat. Dále obsahuje informace o existujících webových editorech kódu, představuje metodiky pro tvorbu webové aplikace a popisuje klíčové prvky projektu.

V Kapitole 3 jsou technologie pro vývoj webových aplikací. Zaměřuje se na populární programovací jazyky a s nimi spojené knihovny a frameworky. Uvedeny jsou klíčové vlastnosti nástrojů pro řešení klientské a serverové části aplikace v projektu. Poslední část kapitoly popisuje virtualizační nástroj Docker.

Následující kapitola 4 obsahuje návrh grafického uživatelského rozhraní podpořené obrázky rozložení stránek. Popsán je i komunikační protokol serverové aplikace. Dále obsahuje formát pravidel pro modifikaci vytvořeného textu.

Implementace je téma kapitoly 5. Popsány jsou důležité části klientské i serverové aplikace. Je zde k nalezení popis struktury a sdílení dat v klientské části aplikace, řešení komunikace se serverem a způsob ověření správného formátu modifikačních/mutačních pravidel na serveru. Poslední část je věnována generování výsledné sady souborů.

Předposlední kapitola 6 se věnuje testování a nasazení aplikace. Na příkladech jsou ukázány jednotkové a další testy. Jejich součástí je konfigurace pro automatizované testování na serveru verzovacího systému Git. Další čast je věnována nastavení Dockeru a konfiguračních souborů pro nasazení do produkce na serverech Heroku, ale i ostatních.

Na závěr kapitola 7 shrnuje výsledky práce a možnosti dalšího vývoje projektu.

Kapitola 2

Analýza požadavků a klíčové prvky projektu

Aplikace se vytváří za nějakým účelem, nejen aby bylo o jednu více. Pomocí aplikace realizujeme myšlenku, nebo řešíme problém. Než ale začne samotný proces vytváření produktu, v první řadě je třeba se věnovat sběru informací. Porozumění požadavků na systém je klíčové pro návrh aplikace [18]. Každá aplice má nějakou cílovou skupinu uživatelů, prostředí, ve kterém se bude používat, a funkce, které má plnit.

2.1 Požadavky na systém

Jedním ze drojů informací je samotné zadání projektu. Z něho lze vyvodit několik požadavků. Program bude implementován jako webová aplikace. Uživateli umožní vytvoření stromové datové struktury a poté následuje generování testovacích dat s dodáním uživatelských parametrů. Návrh aplikace však vyžaduje více konkrétní požadavky. Následující materiály a poznatky byly vytvořeny za spolupráce s vedoucím práce[1].

Cílová skupina uživatelů

První otázkou je, kdo vlastně aplikaci bude používat. Do cílové skupiny uživatelů patří softwarový testeři, vývojáři, ale třeba i studenti informatiky. Při návrhu lze tedy předpokládat, že uživatelé disponují minimální zdatností v oblasti vývoje a testování softwaru. S tím souvisí i odhad, jak velká skupina uživatelů bude aplikaci používat. Předpokládá se, že nástroj využijí jednotky až stovky uživatelů v průběhu jednoho dne.

Funkcionalita a průběh interakce

Další otázkou je, jaké funkce navrhovaná aplikace uživatelům nabídne. V první řadě by měla být dostupná přes webový prohlížeč. V úvodní části interakce umožňuje uživateli práci se stromovou datovou sktrukturou (dále jen *struktura*). Může být načtena ze souboru, nebo napsána pomocí editoru v prohlížeči. Součástí editoru je i syntaktická kontrola pro daný typ struktury. Podporované typy souborů jsou JSON[2] a XML[3]. O výsledku syntaktické

[1] Vedoucí práce – Ing. Aleš Smrčka,Ph.D.
[2] JSON – JavaScript Object Notation
[3] XML – Extensible Markup Language

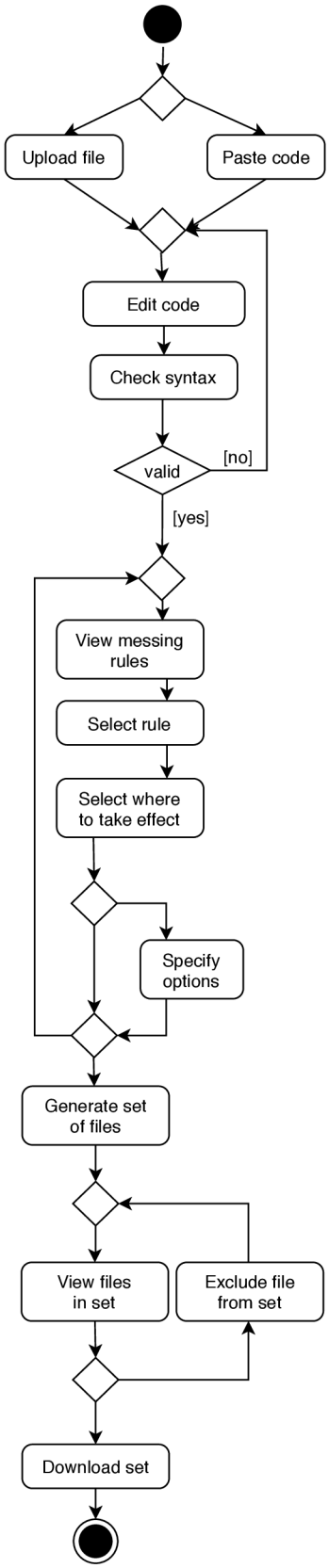
kontroly je uživatel řádně informován. Další kroky v aplikaci jsou podmíněny syntaktickou správností.

Hlavní účel aplikace je generování testovacích dat. To se provádí pomocí modifikací částí kódu zadané struktury. Součástí aplikace je sada pravidel, kde každé pravidlo obsahuje potřebné informace k modifikaci – mutaci, kterou reprezentuje. Uživatel si vybere pravidlo a aplikuje ho na zvolené části kódu. Každé pravidlo má předem definované chování a uživatel volitelně přidává vlastní parametry. Na žádost uživatele se poté generuje výstupní sada souborů. Do obsahu jednotlivých souborů lze nahlédnout a případně ho ze sady odebrat. Posledním krokem je zabalení sady do archivního souboru a jeho stažení.

Seznam formálních požadavků na systém je k nalezení v tabulce 2.1. Ve fázi implementace a testování poslouží k ověření, zda aplikce splňuje všechny očekávané funkce. Posloupnost uživatelské interakce znázoňuje diagram aktivit na obrázku 2.1. Ve fázi návrhu grafického rozhraní pomůže při volbě rozložení informací a funkcí na stránce.

Tabulka 2.1: Seznam formálních požadavků systému

Číslo	Popis požadavku
1	Nástroj je webová aplikace
2	Součástí nástroje je uživatelsky přívětivé grafické rozhraní
3	Backend poskytuje mutační pravidla pomocí REST API
4	Frontend má informativní stránku o projektu a jak se má používat
5	Nástroj pracuje se stromovými datovými strukturami (dále jen struktury)
6	Podporované typy souborů jsou JSON a XML
7	Nástroj podporuje různé způsoby pro vytvoření struktur
8	Struktura může být načtena se souboru
9	Struktura může být napsána uživatelem
10	Uživatel je schopen editovat text struktury a její název
11	Frontend na vyžádání uživatele provádí syntaktickou kontrolu struktury na základě typu souboru
12	Případná syntaktická chyba je nahlášena zpět uživateli, chybná část kódu je vyznačena
13	Provádění mutací nad strukturou je podmíněno správnou syntaktickou kontrolou
14	Uživatel volí, která mutační pravidla se mají použít
15	Uživatel rozhoduje, pro které části kódu se zvolené pravidlo aplikuje
16	Každé mutační pravidlo definuje, jakým způsobem se změní hodnota dané části kódu
17	Uživatel má možnost předgenerování hodnot pro zvolenou část kódu
18	Uživatel může přidávat vlastní hodnoty a mazat existující
19	Na vyžádání uživatele backend generuje sadu souborů na základě aplikovaných mutačních pravidel a parametrů
20	K názvům vygenerovaných souborů je přidán prefix odpovídající názvu aplikovaného pravidla a jeho pořadí
21	Uživatel může procházet obsahy jednotlivých souborů v sadě
22	Uživatel má možnost odstranit vybraný soubor/soubory ze sady
23	Nástroj umožňuje kompresy souborů do ZIP archivu a jeho stažení



Obrázek 2.1: Diagram aktivit znázorňuje interakci uživatele s webovou aplikací. (zdroj: autor)

2.2 Existující webové editory

Analýza se týká webových aplikací, které umožňují vytvoření a editaci kódu. Podle nich se lze inspirovat při návrhu uživatelského rozhraní. Mimo to se věnuje i funkcím, které aplikace nabízí uživatelům při práci se soubory.

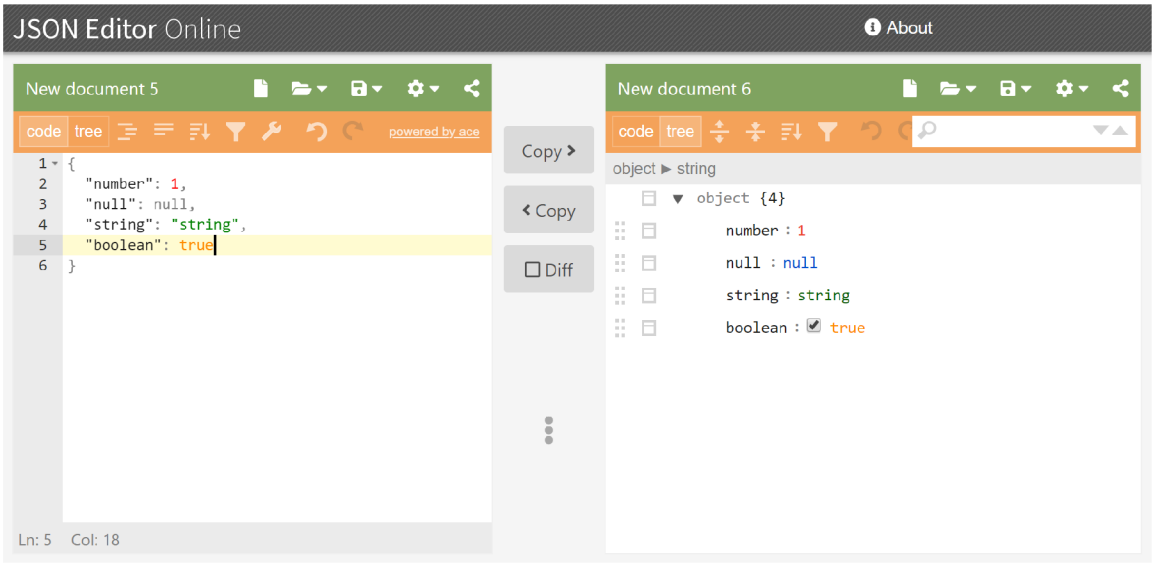
JSON Editor Online

Již z názvu aplikace JSON Editor Online[4] je patrné, že pracuje se soubory typu JSON. Při vstupu na stránku editor textu zabírá většinu místa. Soubor lze nahrát z lokálního úložiště, adresy url, nebo začít editovat nový prázdný dokument. Stránka umožňuje vytvoření i druhého souboru. V tom případě jsou zobrazeny dva editory vedle sebe, které slouží pro porovnání obsahů. Aplikace je implementována v jazyce JavaScript a používá mimo jiné i knihovny Ajv[5] a Ace[6]. Ukázka aplikace je na obrázku 2.2.

Aplikace nabízí:

- **Syntaktickou kontrolu v reálném čase** – řádek obsahující syntaktickou chybu je označený křížkem. Při najetí kurzoru na křížek se zobrazí informace o chybě.
- **Zvýrazňování částí textu v reálném čase** – různé typy hodnot jsou zvýrazněny rozdílnými barvami.
- **Transformaci textového zápisu do stromové podoby** – porovnávání obsahu dvou souborů editor umožňuje pouze ve stromové podobě.
- **Formátování zápisu** – z textu odebere veškeré bílé znaky, aby byl co nejkratší, nebo naopak vloží. Takový formát je lépe čitelný a uživatel se v něm dobře orientuje.
- **Ukládání informací o souborech pomocí cookies** – při návratu na stránku tak uživatel může pokračovat v předchozí editaci.

[4] JSON Editor Online – dostupné z: https://jsoneditoronline.org/
[5] Ajv – Another JSON Schema Validator, dostupné z: https://github.com/ajv-validator/ajv
[6] Ace – dostupné z: https://ace.c9.io/



Obrázek 2.2: Obrázek obsahuje grafického rozhraní JSON Editor Online. Syntaktická kontrola a barevné zvýrazňování částí kódu probíhá už během psaní textu. (zdroj: https://jsoneditoronline.org/)

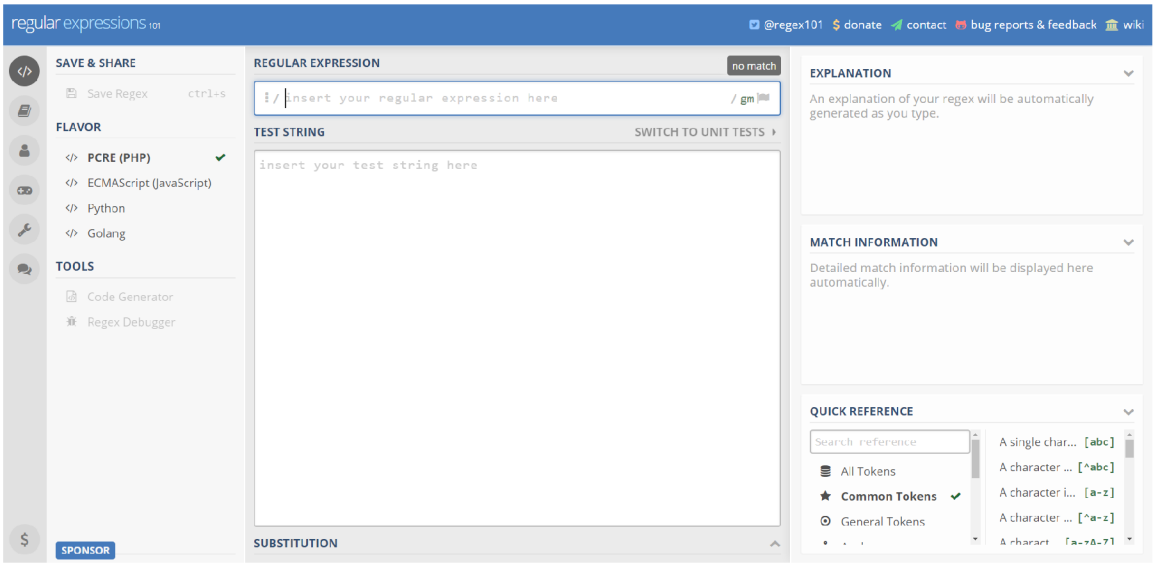
Regular Expressions 101

Regular Expressions 101[7] je webový nástroj pro tvorbu regulárních výrazů. Uživatel zadá regulární výraz a testovací řetězec, ve kterém se hledají shody. Používán je lidmi, kteří se pohybují v informatice a tomu odpovídá vzhled stránky. Podobné rozložení informací najdeme i v populárních IDE[8], jako je třeba Visual Studio Code. Na obrázku 2.3 je ukázka z aplikace.

Aplikace nabízí:

- **Známé rozložení informací pro uživatele IDE** – vývojáři tak mohou předpokládat, kde se co nachází a používání nástroje je díky tomu intuitivní a rychlé.
- **Barevné zvýrazňování v reálném čase** – pozitivní shody v testovacím řetězci jsou jednoznačně odlišeny od zbytku textu. V regulárním výrazu jsou zvýrazněny například párové závorky nebo znaky se speciálním významem.
- **Nápovědy pro tvorbu regulárních výrazů** – všechny potřebné informace jsou dostupné na stejné stránce bez dalšího navigování.

[7] Regular Expressions 101 – dostupné z: https://regex101.com/
[8] IDE – vývojové prostředí



Obrázek 2.3: Na obrázku je grafické rozhraní Regular Expresions 101. Aplikace slouží pro vytváření a testování regulárních výrazů. (zdroj: https://regex101.com/)

2.3 Architektura rozhraní REST

Jeden z formálních požadavků stanovuje, že serverová aplikace musí poskytovat REST[9] rozhraní. Serverová část se tak oddělí od klientské a v budoucnu mohou vznikat další klientské aplikace pro různá zařízení v jiných programovacích prostředí. Navíc umožní navrhnout serverou aplikaci tak, že bude pracovat s jakýmkoliv textem a ne jen s formáty JSON a XML, se kterými pracuje aplikace na straně klienta. REST je architektura, kterou popsal Roy Fielding v roce 2000 ve své disertační práci [14]. Staví na několika základních principech, které je třeba dodržet při návrhu i implementaci [23].

Principy REST:

- **Klient-server** – v tomto rozdělení rolí klient obstarává komunikace s uživatelem, zatím co se server stará o ukládání a poskytování dat.
- **Komunikace je bezestavová** – server neudržuje žádné aktivní spojení s klientem. Klient v žádosti musí poskytnout dostatek informací, aby server mohl jednoznačně určit, v jakém stavu se klient nachází a jakou činnost požaduje.
- **Využívání mezipaměti** – data získaná ze serveru jsou pro klienta označena, zda se mají, nebo nemají uložit do mezipaměti. Pokud jsou data v mezipaměti, při stejném dotazu se použijí místo dotazování serveru.
- **Jednotné rozhraní** – pro veškerou komunikaci mezi klienty a serverem. Rozhraní je potom jednoduché, předvídatelné a dobře rozšiřitelné. Definuje 4 základní omezení:
  - každý zdroj má jednoznačný identifikátor, který se během interakce nemění
  - požadavky reprezentují data a s nimi spojené operace

[9] REST – akronym pro REpresentational State Transfer

- požadavky musí obsahovat všechny potřebné informace k porozumění zprávy
- data ze serveru pro klienta obsahují informace, jaké další kroky mohou následovat

- **Systém je uspořádán do vrstev** – platí zde určitá hierarchie a každá vrstva komunikuje jen s předchozí a následující vrstvou.
- **Programový kód na vyžádání** – je jediný volitelný princip a umožňuje rozšířit funkcionalitu klienta pomocí appletu nebo skriptu.

2.4 Stromové datové struktury

Data slouží k uchování informací a občas je třeba je nějak uspořádat, aby s nimi šlo dobře pracovat. V programování se k tomu účelu používají různé datové struktury a mezi ně patří i stromové struktury. Struktury jsou abstraktní, definují vnitřní strukturu a možné operace. Ve stromové struktuře jsou data uspořádána hierarchicky od tzv. kořene až po koncové listy. Následující část textu se zabývá dvěma formáty zápisu stromových struktur, se kterými klientská aplikace pracuje.

JavaScript Object Notation – JSON

Notace JSON byla poprvé představena v roce 2001 na webu JSON.org [1]. Často je používána při výměně dat a podporuje ji mnoho programovacích jazyků. Syntaxe se skládá z hranatých a složených závorek, čárek a dvojteček. Zápis jde snadno programově zpracovat a současně poskytuje dobrou čitelnost pro lidské oko.

JSON stavý na dvou strukturách:

- kolekce dvojic klíč/hodnota
- seřazený seznam hodnot

Ukázka zápisu dat ve formátu JSON:

```
{
  "record": {
    "id": 1,
    "message": "Lorem Ipsum"
  }
}
```

Extensible Markup Language – XML

Značkovací jazyk XML je používaný v dokumentech, které obsahují strukturované informace. První verzi 1.0 představila v roce 1998 společnost W3C [10]. Stejně jako JSON byl navržen pro přenos dat a to s dobrou čitelností pro lidi i počítačové programy [13]. Na rozdíl od notace JSON umožňuje používat v dokumentech komentáře. Popis struktury souboru XML je možné definovat pomocí různých schémat, mezi které patří například XML Schema Definition (zkr. XDS).

Základní stavební bloky XML:

[10] W3C – World Wide Web Consortium

- **Tag** – je řetězec začínající znakem „<“ a končící znakem „>“. Dělí se na 3 typy:
  - počáteční tag, jako <plant>
  - konečný tag, jako </plant>
  - nepárový tag, jako <break />
- **Element** – se skládá z dvojice počátečního a konečného tagu se stejným označením, nebo se jedná o tag nepárový.
- **Atribut** – je dvojice klíč/hodnota, která je zapsána uvnitř počátečního nebo nepárového tagu.

Ukázka zápisu dat ve formátu XML:

```
<record key="value">
  <id>1</id>
  <message>Lorem Ipsum</message>
  <!-- comment in XML -->
</record>
```

Dokument může začínat XML deklarací, která udává informace o souboru, jako je verze XML, použité kódování a další.

```
<?xml version="1.0" encoding="UTF-8"?>
```

2.5 Testování, mutační testování a vkládání chyb

Testování softwaru je proces, při kterém se spouští program za účelem nalezení chyb. Také zahrnuje aktivity spojené s ověřováním, do jaké míry program splňuje požadavky, které jsou na něj kladeny. Na rozdíl od fyzicky hmatatelných nástrojů software netrpí na opotřebování nebo výrobní vady. Většina problémů souvisí s jeho návrhem a implementací [20].

Některé chyby jsou snadno předvídatelné, jako například dělení nulou. Někdy ale chyby vznikají za bizardních okolností, kterým předcházelo několik kroků. Kromě toho jsou programy velmi komplexní. Otestování každé části pro každý stav, který může nastat, by zabralo neúměrné množství času. Proto se při testování hledá vhodný kompromis mezi investovanými prostředky a výsledky. Odhalování chyb vyžaduje od testerů kreativitu, zkušenosti a intuici. Práci jim usnadňují dostupné nástroje a velké množství testovacích metod.

Mutační testování

Kdybychom chtěli najmout editora pro kontrolu nějakého textu, jak zvolit toho nejlepšího. Jednou z možností je uchazečům dát text, který obsahuje překlepy a gramatické chyby. Tím dosáhneme dvou cílů. Dostaneme opravený text a přehled o schopnostech uchazečů. Není ale zaručeno, že všechny chyby budou nalezeny [21].

Na stejné myšlence je založeno mutační testování softwaru. Vygeneruje se sada mutantů programu, které se liší od původního alespoň v jedné jeho části. Většinou taková mutace vede k chybě, kterou dobré testy odhalí. Někdy se může stát, že se s chybou program vypořádá a výsledek chování zůstane stejný, jako v případě originálního programu. Tímto postupem lze ověřit kvalitu používané sady testů a dobrá sada testů vede k odhalení chyb v původním programu. Mutační testování se skládá ze tří kroků:

- **Generování mutantů** – mutant je kopie původního programu, která obsahuje nějakou změnu. Změny se dějí na základě mutačních operátorů, předem definovaných pravidel, která obsahují informace potřebná k provedení změn. Automatizované nástroje provádí generování mutantů, kterých může být velké množství.
- **Spouštění původního programu a mutantů** – se provádí proti sadě testů na základě porovnávání výsledků. Pokud test odhalí změnu ve výsledku, mutant je odhalen a „zabit“.
- **Analýza výsledků** – slouží v určení kvality testů. Výsledkem mutační skóre, což je číslo mezi 0 (0 %) a 1 (100 %). K výpočtu je třeba identifikovat ekvivalentní mutanty.

Vkládání chyb

Správnosti chování sotwaru lze ověřit pomocí kladných testů. Slouží pro ověření, zda se software chová správně za očekávaných podmínek. Na druhé straně negativní testy se snaží software narušit, ukázat, že nefunguje. Od softwaru se očekává, že do určité míry je schopen negativním testů odolat. Vkládání chyb mezi takové testy patří [20].

Do testovaného programu se záměrně vkládají poruchy a následně se sleduje, jaké stavy a případné chybové výstupy nastanou. Odhaleny jsou slabiny, které by běžným testům mohly zůstat skryty. Často se chyby vkládají do takových částí kódu, které mají komunikační a kooperační charakter. Jsou totiž dostatečně interaktivní, aby pro ně měla tato metoda smysl [22].

Metoda vkládání chyb je vhodná například pro ověření robusnosti webových služeb. Vyměňují si mezi sebou informace v různých formátech, mezi které patří i XML a JSON, a jejich fungování může být závislé na dalších službách. Chyba v jedné může zapříčinit poruchu celého systému [12].

Vhodné nástroje umožní komunikaci narušit a přivést je tak do výjimečných stavů. Dobře navržená služba by měla na testy reagovat a v její funkčnost by neměla být narušena. Existují různé nástroje pro automatizované vkládání chyb a analýzu zpětné vazby testované služby. Označují se zkratkou SWIFI[11] a zaměřují se na specifické části softwaru, zbytek zůstává neporušen. Podle techniky vkládání chyb se dělí do dvou kategorií:

- **Vkládání chyb během kompilace** – je způsob, kdy se mění zdrojový kód programu. Nehýhodou je, že vyžaduje přístup ke zdrojovému kódu a taky může dojít k nezáměrným chybám, které mohou zkleslit zpětnou vazbu.
- **Vkládání chyb za běhu** – používá různé spouštěče, které chyby do programu vkládají. Spouštěče mohou být závislé na časových intervalech, nebo na událostech, které v programu běžně nastávají.

[11] SWIFI – SoftWare Implemented Fault Injection

Kapitola 3

Technologie pro vývoj webových aplikací

V kapitole jsou popsány dostupné technologie pro tvorbu webových aplikací a služeb. První část se zabývá vhodnými nástroji pro tvorbu grafického uživatelského rozhraní, které je součástí klientské aplikace. Druhá část obsahuje informace o vhodných knihovnách k implementaci serverové části podle architektury REST. V poslední části je zmíněn Docker a jak přispívá při nasazení aplikace do produkce.

3.1 JavaScript a knihovna React

JavaScript je interpretovaný skriptovací programovací jazyk a pro tvorbu webových aplikací je velmi populární volbou. Nebylo tomu tak ale vždy. Vznikl v roce 1995 a využíván byl zřídka, protože prohlížeče jeho specifikaci řádně neimplementovaly. V roce 2009 vyšla specifikace ECMAScript 5, která přitáhla větší poroznost vývojářů a od roku 2012 všechny moderní prohlížeče podporují ECMAScript verze 5.1 [7].

Použití JavaScriptu není limitováno pouze na webové prohlížeče. Ke spuštění zdrojového kódu je potřeba interpretační nástroj, jako je například V8[1]. Jedná se o volně dostupný software napsaný v jazyce C++. Používá ho prohlížeč Google Chrome, ale také prostředí Node.js, ve kterém běží různé serverové aplikace. Interpretačních nástrojů existuje několik a každé zařízení, které nějaký obsahuje, dokáže pracovat s JavaScriptovým kódem.

Node Package Manager

V mnoha JavaScriptových aplikacích se setkáme se správcem balíčků NPM – Node Package Manager. Implementovat každou funkcionalitu v projektu by zabralo mnoho času, navíc existuje šance, že někdo v minulosti podobnou funkcionalitu potřeboval. Znovupoužitelný program je možné sdílet s ostatními ve formě balíčku. Součástí projektu tedy může být několik balíčků, na kterých je závislý, a potom vlastní implementace.

Nástroj NPM se stará o projekt, přidávání balíčků, aktualizování verzí a další. Soubor `package.json` obsahuje všechny podstatné informace jak o projektu, tak o balíčcích, na kterých závisí [9]. Dále může obsahovat krátké skripty, například ke spuštění aplikace a testů. Hlavní výhodou je, že k přenosu projektu na jiné místo stačí zdrojový kód a jeden soubor, podle kterého se stáhnou a nainstalují všechny potřebné balíčky s odpovídajícími verzemi.

[1] V8 – dostupné z: https://v8.dev/

Také lze rozlišit, které balíčky se používají jen ve fázi vývoje a v produkčním nasazení jsou přeskočeny.

Knihovna React

React je JavaScriptová knihovna používaná k vytváření uživatelských rozhraní. Základní stavební bloky Reactu jsou komponenty, které lze opakovaně používat [11]. Často se používá při tvorbě jednostránkových aplikací. Obsah takové stránky se mění dynamicky přepisováním obsahu současné stránky. Při interakci tak nedochází k načítání celých stránek. React neřeší problémy spojené s navigací nebo správy stavu aplikace. K vytvoření komplikovanější aplikace je třeba přidat další knihovny.

Komponenty

Celá aplikace je složena z několika komponent. O jak velké komponenty se jedná je zcela v rukou programátora. Identicky vypadající stránky mohou mít pár komponent, stejně tak jich ale můžou být i desítky. V Reactu se vytváří pomocí funkcí nebo tříd a lze je do sebe libovolně vkládat.

Hlavním úkolem komponenty je vykreslit nějakou část stránky. K zápisu je možné použít klasické HTML elementy, ale častěji se používá syntaxe JSX. Webové prohlížeče JSX nedokáží interpretovat, takže se používají nástroje, jako je Webpack[2], pro transpilaci do interpretovatelné podoby. Komponenty lze rozdělit na:

- **Bezstavové komponenty** – jsou plně závislé na parametrech, které dostanou. Příkladem může být tlačítko, které v parametrech dostane popisek a funkci. Při vykreslení zobrazí popisek a po kliknutí provede funkci.
- **Stavové komponenty** – jsou kromě parametrů ještě závislé na svém vnitřním stavu, který se během interakce může měnit. Například rozbalovací menu dostane jako parametr seznam položek. Zároveň si drží vnitřní stav, jestli má být seznam rozbalený, nebo skrytý. Kliknutím se změní vnitřní stav i způsob vykreslení položek.

Virtuální DOM

Webová stránka je v podstatě HTML dokument a DOM[3] pro ni poskytuje objektově orientované aplikační rozhraní. V něm jsou části stránky uspořádány do stromové struktury. Pomocí DOMu jde s dokumentem dobře manipulovat, přidávat a odstraňovat jednotlivé části, nebo měnit jejich obsah. Problém ale nastává při vykreslování uživatelského rozhraní, které může být pomalé. Obvzlášť v případě, když se změní element blízko „kořene“ a všechny elementy „pod ním“ (child elementy) musí být vykresleny znova.

React používá svoji vlastní verzi DOMu, které se říká virtuální DOM. Ten je uchovaný v paměti a aplikace na něm provádí veškeré změny na místo reálného DOMu. Když se změní virtuální DOM, React ho porovná s reálným a identifikuje elementy, kterými se liší. V reálném DOMu se poté aktualizují jen změněné elementy, což je rychlejší, než přímá manipulace.

[2]Webpack – dostupné z: https://webpack.js.org/
[3]DOM – Document Object Model

3.2 TypeScript a platforma Angular

TypeScript je volně šiřitelný programovací jazyk vyvíjený společností Microsoft. Jedná se o nadstavbu nad JavaScriptem a každý validní JavaScriptový kód je i validním TypeScriptovým kódem. Oproti JavaScriptu navíc nabízí například výčtový datový typ (`enum`) a statické typování [10].

Statické typování zlepšuje čitelnost kódu a do určité úrovně dovoluje kontrolovat jeho správnost. Také umožňuje IDE[4] napovídat programátorům při psaní kódu. Používání statického typování je zcela volitelné, pokud není specifikovaný datový typ, je prováděno dynamicky.

TypeScript se pro programování webových aplikací nepoužívá přímo, protože webové prohlížeče s ním neumí pracovat. Zdrojový kód se kompiluje do JavaScriptu. Výhodou je, že během kompilace lze odhalit chyby v kódu, které by se v běžném JavaScriptu projevily až při spuštění nebo během testování aplikace. JavaSript je dynamicky typovaný jazyk, takže typovost specifikovaná v TypeScriptu za běhu aplikace nemusí být ve skutečnosti dodržena.

Platforma Angular

Angular je vývojová platforma a framework pro návrh jednostránkových aplikací. První verze byla postavena na JavaScriptu a vyšla pod názvem AngularJS v roce 2010. Celý framework byl ale přepsán do TypeScriptu a roku 2016 vyšla novější verze označena pouze Angular, někdy označována jako Angular 2. Všechny následující verze rovněž používají TypeScript [3].

Stejně jako v Reactu se v i Angularu setkáme s konceptem dekompozice webové stránky do znovupoužitelných komponent. Místo virtuálního DOMu Angular používá inkrementální DOM. Každá komponenta je zkompilována do řady instrukcí. Pomocí instrukcí vytváří stromovou strukturu DOMu a k aktualizování dochází přímo na místě, když se změní zobrazovaná data. Angular toho ale jako framework a platforma nabízí daleko více. Součástí je i vlastní CLI[5] pro vytvoření, spuštění a testování aplikace.

Navigace v aplikaci

Navigace v jednostránkové aplikaci probíhá tak, že se zobrazují a skrývají jednotlivé komponenty, bez potřeby opětovného načítání stránky. O změnu zobrazených komponent se stará Angular Router. Adresa URL zadaná do prohlížeče je interpretována jako sekvence instrukcí měnící vzhled stránky. Router také nabízí možnost přidání podmínek, které musí být splněny pro přístup k daným stránkám.

Vkládání závislostí

Angular má vlastní framework pro vkládání závislostí, který zvyšuje efektivnost a modularitu aplikace. Závisloti jsou v Angularu služby nebo objekty, které třída potřebuje ke správnému fungování. Podstatou tohoto přístupu je, že třída žádá o zdroje vnějšího poskytovatele, než aby si zdroje sama vytvářela. Při vytváření instancí se framework stará o poskytnutí všech definovaných závislostí.

[4] IDE – Integrated Development Environment, česky: Vývojové prostředí
[5] CLI – Command Line Interface, česky: Příkazový řádek

Knihovna RxJS

Součástí Angularu je i knihovna RxJS, která obsahuje konstrukce pro sdílení dat mezi komponentami, asynchronní programování a obsluhu událostí. Používá návrhový vzor pozovatele, kde pozorovaný objekt udržuje seznam svých pozorovatelů, které má upozornit, když nastane změna stavu. Pozorovaný objekt definuje funkci pro poskytování dat, která se provede v momentě přihlášení odběratele. Odběratelům přichází upozornění na změnu dat do doby, než se odhlásí.

Moduly

Výsledkem dekompozice stránky je nějaké množství komponent. K nim v Angularu přibudou další funkční části, jako služby pro vkládání závislostí a jiné. Moduly slouží k uspořádání spolu souvisejících částí aplikace. Rozdělení do modulů určuje konfiguraci kompilace a jak budou dodávány instancím závisloti za běhu aplikace.

3.3 Node.js a framework Express

Doposud zmíněné technologie byly spojeny s tvorbou uživatelského rozhraní a jazykem JavaScript. Použít stejný jazyk k vývoji serverové aplikace by bylo výhodné. K tomu je ale třeba vhodné běhové prostředí, které poskytne Node.js [8].

Node.js je postaven na interpretačním nástroji V8 pro jazyk JavaScript. Z něj byly odstraněny některé nepotřebné funkce, které používaly pouze webové stránky. Na druhé straně byl nástroj rozšířen například o programové aplikační rozhraní souborového systému. Díky němu může Node.js číst a zapisovat soubory a složky na lokální úložiště.

Z pohledu obsluhy velkého množství požadavků je velmi výkonný. Používá neblokující a událostmi řízený vstupně/výstupní model. Tok programu je řízený výskytem událostí, které jsou zachytávány takzvanými „posluchači“. Posluchač definuje události, na které reaguje, a funci, která se následně provede. Kód je asynchronní a při obsluze požadavků se něčeká na dokončení těch předchozích.

Framework Express

K vytvoření jednoduché serverové aplikace stačí Node.js, ale obsluhování požadavků na různých koncích rozhraní REST usnadní Express. Jedná se o malý, flexibilní a volně šiřitelný framework [6]. Umožňuje snadnou definici posluchačů pro příchozí požadavky na základě URL adresy a typu požadavku. Také nabízí velké množství middleware, které lze použít například pro parsování těla HTTP požadavku, odhalování chyb ve fázi vývoje aplikace a nebo konfiguraci CORS[6].

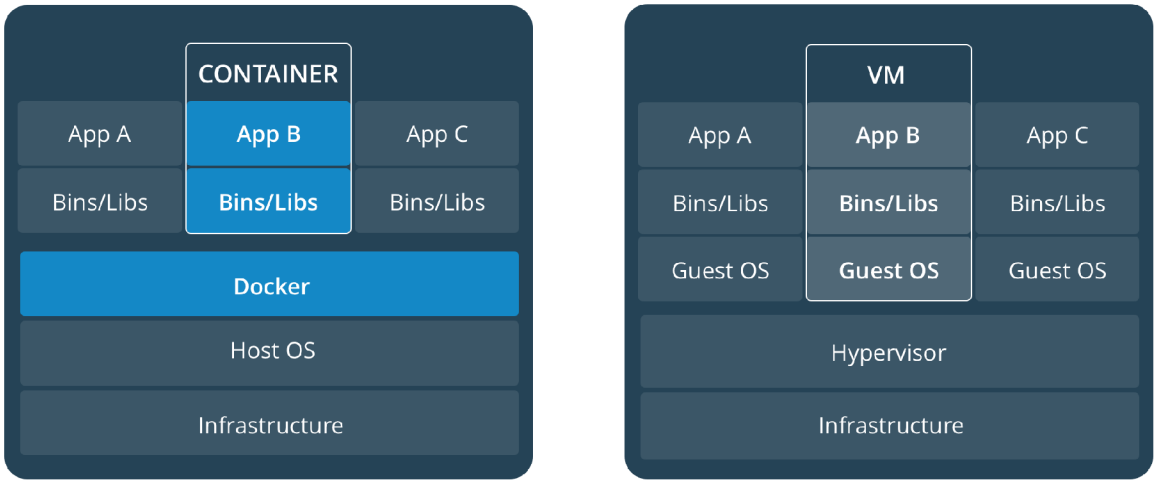
3.4 Virtualizační nástroj Docker

Když je aplikace navržena a naprogramována, dalším krokem je vytvoření verze pro produkci a následné spuštění na serveru. Jeden z používaných přístupů je pomocí virtualizace. Vytvoří se vlastní vituální prostředí pro aplikaci, které je izolované od zbytku systému. Lze mu přiřadit část z dostupných prostředí, jako je výpočetní výkon a úložiště stroje. Pro-

[6]CORS – Cross-Origin Resource Sharing

středí jde libovolně přizpůsobit požadavkům aplikace a umožňuje snadný přesun z jednoho zařízení na jiné [4].

Populárním nástrojem v oblasti virtualizace je Docker. Klasické nástroje poskytují virtuálním strojům pouze abstrakci hardwarové úrovně. Jednotlivá prostředí tak musí mít vlastní operační systém a další prostředky v podobě knihoven a nástrojů. Docker na druhé straně sdílí část operačního jádra s virtuálními prostředími. K fungování jim stačí už jen nějaký virtuální disk s knihovnami, zdrovými soubory atd. [5]. Znázornění obou přístupů je na obrázku 3.1.



(a) Docker pro virtuální prostředí používá kontejnery, které sdílí část operačního jádra s hostitelským systémem.

(b) Tradiční virtualizační nástroj poskytuje abstrakci hardwarové úrovně svým virtuálním strojům.

Obrázek 3.1: Grafické porovnání Dockeru s tradičním virtualizačním nástrojem. (zdroj obou obrázků: Docker Inc. [5])

Obrazy disků

Docker používá obrazy disků pro vytvoření kontejnerů. Jedná se o soubor obsahující souborový systém a konfiguraci prostředí. Je možné si vytvořit vlastní, Docker ale nabízí na webu Docker Hub [7] celou řadu volně dostupných disků. Při vytváření virtuálního prostředí je možné použít vhodný obraz a přidat vlastní úpravy pro provoz aplikace. Takto upravený disk je možné uložit a sdílet s ostatními uživateli Docker Hubu.

Kontejnery

Kontejner je spuštěná instance nějakého obrazu disku. Hlavní výhodou Dockeru je, že když více kontejnerů používá stejný obraz, na stroj se uloží pouze jednou a použije se znovu. Některé jsou si natolik podobné, že svoje společné části sdílejí a tím šetří úložný prostor. Současně může běžet více instancí založených na stejném obrazu. Kontejnery mají jeho vlastní kopii a jsou od sebe izolované.

[7] Docker Hub – dostupné z: https://hub.docker.com/

Dockerfile

Dockerfile je soubor obsahující instrukce k automatickému vytvoření obrazu disku. Typicky obsahuje název existujícího obrazu z Docker Hubu, který poslouží jako základ, a potom řadu instrukcí, které ho upraví tak, aby poskytoval vhodné prostředí pro chod aplikace. Pomocí příkazu `docker build <cesta-k-Dockerfile>` dojde k vytvoření nového obrazu a vypsání jeho generického názvu. Pomocí `docker run <nazev-obrazu>` se potom spustí nová instance kontejneru.

Kapitola 4

Návrh systému

Následující kapitola obsahuje návrh systému a jednotlivých částí. Vychází ze seznamu požadavků v kapitole 2. V první sekci je návrh grafického rozhraní a jeho výsledná podoba. Další sekce obsahuje návrh a formát zápisu mutačních pravidel. Kapitolu uzavírá popis koncových bodů REST rozhraní serverové aplikace.

4.1 Uživatelské rozhraní klientské aplikace

Hlavním účelem aplikace je generování testovacích dat stromových struktur. Takový nástroj pravděpodobně vyhledají lidé pohybující se v oblasti informatiky. Nabídnout uživatelům podobné prostředí, se kterým se již mohli setkat, pomůže zlepšit jeho přívětivost.

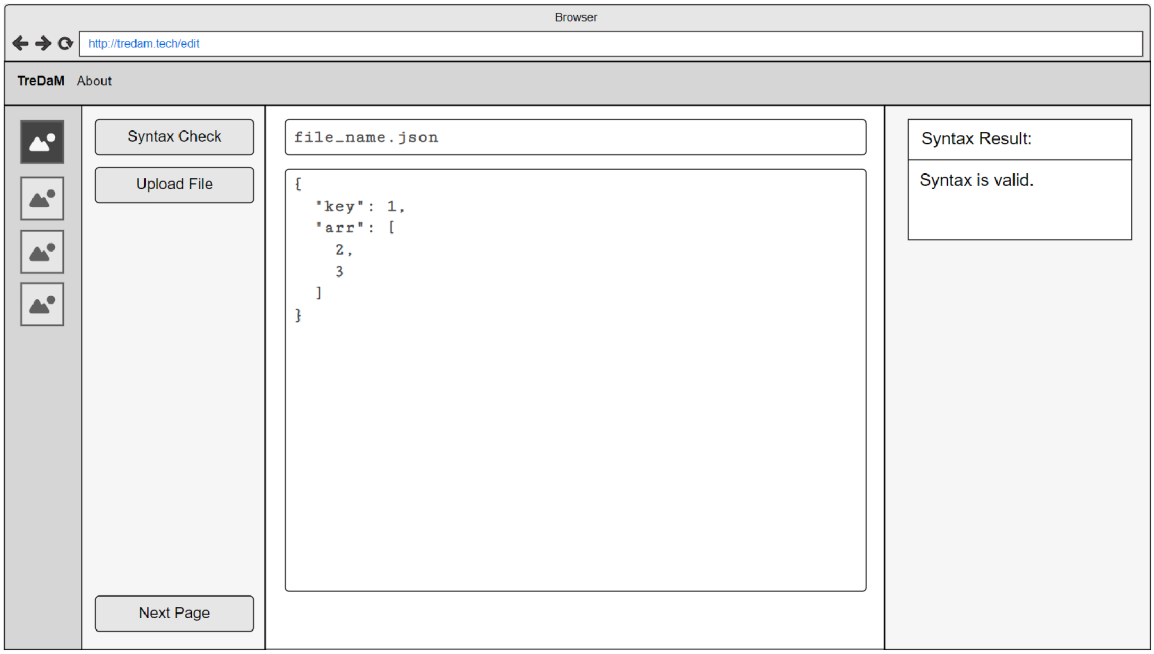
Nástroj v první řadě plní funkci textového editoru a v další části aplikace s vytvořeným textem následně pracují. Návrh rozhraní tak vychází z rozložení informací, jaký používají populární vývojová prostředí, jako je například Visual Studio Code [2]. Podobný přístup volí i stránka pro tvorbu regulárních výrazů Regular Expresions 101[1].

Návrh zaměřený na aktivity

Při návrhu zaměřeném na aktivity (Activity-Centered Design – ACD) se větší část pozornosti věnuje plnění úkolů, než individuálním potřebám uživatelů [19]. Ten potřebuje splnit nějakou činnost, poklikání si v hezké aplikaci je na druhém místě. V první fázi návrhu se aktivity analyzují. V projektu je to znázorněno pomocí diagramu aktivit v kapitole 2. Podle něj lze identifikovat, jaká data jsou pro uživatele relevantní v daných částech aplikace. Tento přístup umožňuje pohled na systém jako celek. Postup uživatele aplikací má lineární charakter a umožňuje interakci rozdělit do několika fází:

- **Vytvoření souboru** – buď nahráním z lokálního úložiště, nebo přímo pomocí editoru. Možné je editovat název souboru a obsah souboru. Editor by měl barevně zvýrazňovat některé části kódu pro zlepšení přehlednosti. Dalším požadavkem je syntaktická kontrola. Výsledek kontroly je oznámen zpět uživateli. Při nesprávném kódu je řádek obsahující chybu označen a pro velké soubory k němu existuje rychlá navigace.
- **Aplikování mutací** – vyžaduje zobrazení seznamu dostupných mutačních pravidel a vytvořeného souboru. Obsah souboru už nelze měnit a předchozí funkcialitu nahradí

[1] Regular Expresions 101 – dostupné z: https://regex101.com/



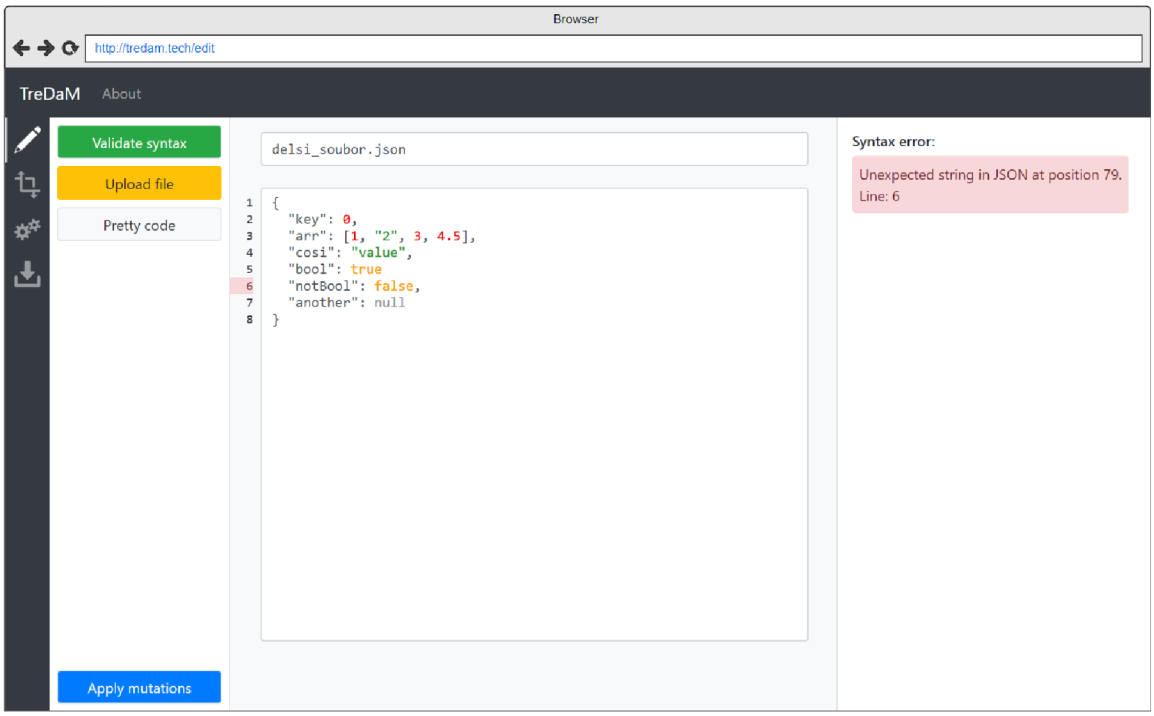
Obrázek 4.1: Návrh úvodní stránky aplikace obsahující editor. Horní lišta slouží pro přepínání mezi informační stránkou a samotným prostředím nástroje. Průchod nástojem je lineální, uživateli je nutné naznačit, kolik oken je zapojeno do celého procesu a v jaké fázi se zrovna nachází. K tomu slouží levý navigační panel, jehož součástí jsou i klíčové ovládací prvky pro danou stránku. Pravá strana je více flexibilní a zobrazuje doplňující informace k prováděné akci s možností přidání dalších ovládacích prvků. (zdroj: autor)

aplikování mutací. Při procházení jednotlivých pravidel se zvýrazňují místa, kde pravidla definují nějakou modifikaci textu. Současně se zobrazuje i stručný popisek, jak pravidlo funguje a co generuje. Při zvolení pravidla je možné přepínat mezi jednotlivými místy v kódu a jednotlivě aplikovat mutace. Hodnoty je možné předgenerovat a nebo uživatel dodá vlastní.

- **Generování sady souborů** – obsahuje přehled všech vygenerovaných souborů na základě aplikovaných pravidel, které dohromady tvoří testovací sadu. Uživatel jimi volitelně prochází a má možnost vybrané soubory ze sady odstranit. Možné je i smazané soubory zpětně na vyžádání dogenerovat.

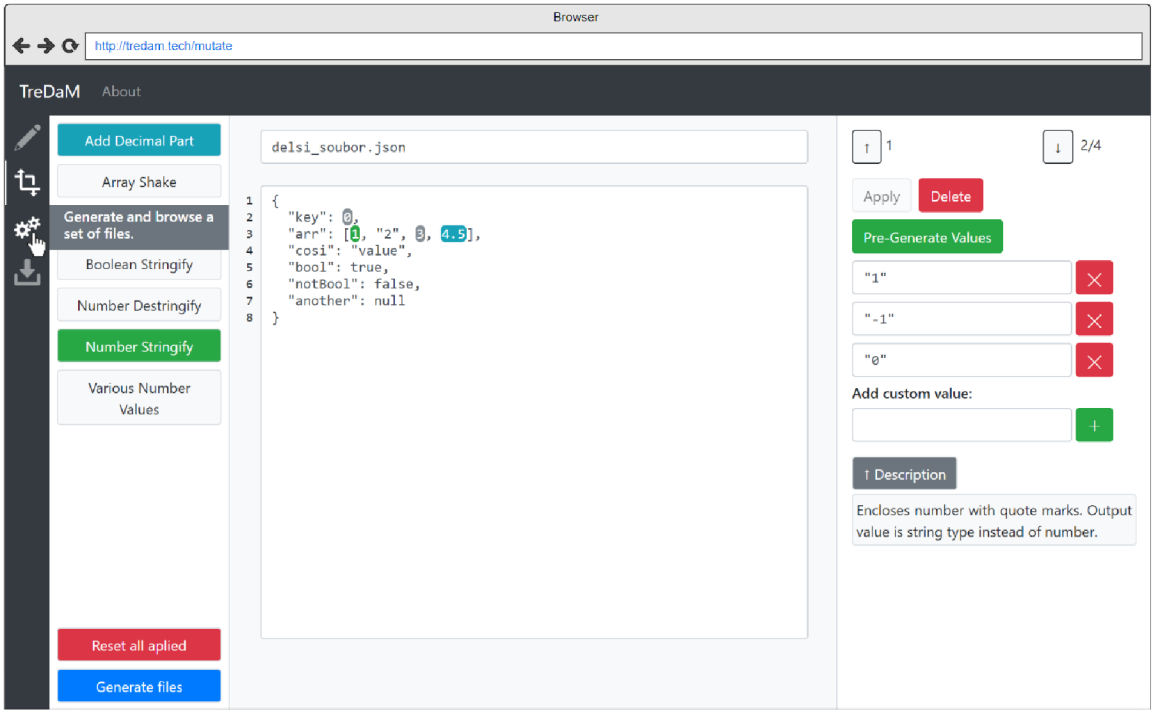
- **Stažení sady** – je posledním krokem aplikace. Zbylé soubory v sadě se na vyžádání uživatele zkomprimují do archivního souboru ZIP a stáhnou na lokální úložiště. Název archivu je předem daný, ale uživatel má možnost ho změnit.

Během všech fází je možné přejít na informativní stránku, kde je stručný popis nástroje, k čemu slouží a jak se používá. Rozdělaná práce se nepřeruší a po návratu je v ní možné pokračovat na stejném místě. Jak vypadá grafický návrh uživatelského rozhraní je vidět na obrázcích 4.1 a 4.3. Obrázky 4.2 a 4.4 potom zobrazují už skutečnou implementaci rozhraní se všemi přidanými detaily. Slouží pouze pro ilustraci, zbylé stránky aplikace respektují stejné rozložení informací.



Obrázek 4.2: Realizace úvodní stránky. Navíc obsahuje číslování řádků textu souboru. V pravé části je vidět výsledek syntaktické kontroly a chybný řádek označuje stejná barva. Některé části textu jsou barevně zvýrazněny na základě typu souboru. (zdroj: autor)

Obrázek 4.3: Návrh stránky pro aplikování mutačních pravidel. Seznam pravidel levého panelu nahradil ovládací prvky předchozí stránky. Pravý panel obsahuje popisek zvoleného pravidla a přidány jsou další ovládací prvky s činností spojené. Následující stránky respektují rozložení informací a rozhraní je tak konzistentní. (zdroj: autor)



Obrázek 4.4: Realizace návrhu stránky s mutacemi. Při najetí kurzorem na ikonu levého navigačního panelu se zobrazí krátký popisek dané stránky. Zvolené a již aplikovaná pravidla se od ostatních liší barvou. Stejně jsou rozlišena i místa v kódu. (zdroj: autor)

4.2 Pravidla mutací a jejich formát

Modifikace textu musí být nějak předem definované. V projektu je to řešeno pomocí mutačních pravidel. Pravidla jsou navržena tak, aby byla dobře přenositelná mezi serverem a klientem a dala se jednoduše upravit a vytvořit. Většina pravidel v projektu vychází z práce Li a Millera [17] a práce Franzotte a Vergilia [15]. Každé pravidlo je zapsáno do samostatného souboru typu JSON, které načítá serverová aplikace. Obsahují následující informace:

- **usages** – obsahuje seznam typů souborů, pro které je pravidlo určeno a jeho funkcionalita je ověřena. Může ale fungovat i pro text jiného typu.
- **name** – slouží jako identifikátor pravidla, proto by mělo být pro každé unikátní. Zápis je v „snake case“ notace – mezery jsou nahrazeny znakem podtržítka „_“. Používá se na několika místech implementace klientské a serverové aplikace.
- **friendlyName** – je verze předchozího `name`, ale lépe čitelná pro uživatele. Proto se používá hlavně v klientské aplikaci např. na ovládacích prvcích grafického rozhraní.
- **description** – popisuje chování pravidla, s jakými hodnotami pracuje a jaký je přibližný výsledek generování. Slouží jako nápověda pro uživatele, ale zároveň i jako programová dokumentace pravidla.
- **pattern** – je regulární výraz, který jednoznačně identifikuje místa v textu, se kterými pravidlo provádí modifikace pomocí `transform`.
- **transform** – obsahuje zápis funkce v jazyce JavaScript. Přebírá právě jeden parametr a výsledkem je vždy pole primitivních datových typů.

Všechny hodnoty jsou typu `string`. Následující kus kódu ukazuje reálné mutační pravidlo systému pro lepší ilustraci:

```
{
  "usages": [ "text/xml" ],
  "name": "required_change",
  "friendlyName": "Required Change",
  "description": "Generate the rest of available values for \"required\" attribute.",
  "pattern": "(?<=required=\\\").*?(?=\\\")",
  "transform": "(param) => { param = param.toUpperCase(); return [ \"YES\", \"NO\" ].filter
      (item => item !== param) }"
}
```

Je nutné dodat, že chování regulárních výrazů `pattern` nemusí být ve všech webových prohlížečích stejné. Například konstrukce `lookafter` a `lookbehind` fungují pouze v prostředí Node.js a prohlížečích Chrome, Opera a Microsoft Edge, jelikož jsou založeny na stejném běhovém prostředí V8[2]. V budoucnu by také mohly vzniknout další klientské aplikace, které nepodporují ani regulární výrazy, ani funkce zapsané v JavaScriptu. Pro takové případy jsou součástí serverová aplikace koncové body nabízející stejnou funkcionalitu. Ty používá i klientská aplikace projektu při práci s textem, aby byla zaručena podpora všech dostupných webových prohlížečů.

[2] V8 – JavaScript and WebAssembly engine, dostupné z: https://v8.dev/

Jedno z možných rozšíření aplikace je používat `pattern` a `transform` přímo pro vybrané prohlížeče. Práce s textem by byla rychlejší a umožnila by stažení aplikace jako PWA[3] do počítače k offline upravování.

4.3 Koncové body rozhraní serverové aplikace

Aplikace jako celek má řadu funkcionalit a klientská část, kromě rozbrazování dat a ovládacích prvků, obsahuje jenom syntaktickou kontrolu. Všechny ostatní funkce poskytuje serverová část pomocí REST rozhraní, které je popsané v kapitole 2.

REST nespecifikuje formát zápisu odesílaných a přijímaných dat. Pro projekt je zvolen zápis pomocí notace JSON. Nabízené funkce a přístup k datům není nijak omezen, například rolí uživatele. Jejich uspořádání na koncových bodech je ilustrováno pomocí obrázku 4.5 a bližší popis v následujícím textu. Koncové body se skládají z URL adresy a HTTP metody.

Získání mutačních pravidel

Všechna mutační pravidla jsou poskytována na adrese `/rules` metodou GET. Výběr pravidel podle typu soubory lze získat na adresách `/rules/json` a `/rules/xml` stejnou metodou. Použití parametrů by také fungovalo, tento přístup je více flexibilnější, což se projeví spíše ve fázi implementace. Odpověď obsahuje pole pravidel v následujícím formátu:

```
{
  usages: [ string ],
  name: string,
  friendlyName: string,
  description: string,
  pattern: string,
  transform: string
}
```

Úprava textu

Mutační pravidla definují, se kterými částmi kódu pracují. Klientská aplikace může s textem pracovat přímo, nebo použít koncové body serveru. Adresa `/rules/:name` slouží pro ověření existence pravidla podle názvu a používá metodu `GET`. Odpověď obsahuje nalezené pravidlo nebo chybovou hlášku.

Koncový bod na adrese `/rules/:name/match` používá metodu `POST`. Slouží k nalezení všech míst v textu, nad kterými pravidlo provádí modifikace. Tělo žádosti musí obsahovat:

```
{
  text: string
}
```

Zdrojový `text` je prohledán na základě zvoleného pravidla v adrese a odpověď obsahuje pole nalezených řetězcových literálů.

Adresa `/rules/:name/highlight` používá metodu `POST`. Slouží k označení všech míst v textu, se kterými pravidlo pracuje. Tělo žádosti musí obsahovat:

```
{
  text: string,
  splitter: string,
  enclose: string
```

[3] PWA – Progresivní Webová Aplikace

```
}
```

Zdrojový `text` je opět prohledán na základě zvoleného pravidla v adrese a nalezená místa jsou z obou stran rozšířena o znaky `enclose` a potom `splitter`. Takto upravený text je odeslán v odpovědi.

Generování souborů a hodnot

Ke generování slouží dva koncové body. První z nich se používá k předgenerování hodnot pro konkrétní pravidlo. Adresa `/generate/:name` určuje pravidlo a metoda HTTP je `POST`. Tělo žádosti musí mít:

```
{
  value: any
}
```

Odpověď obsahuje vygenerované hodnoty uspořádané do pole. Transformační funkci definuje zvolené pravidlo a jako jediný parametr přebírá hodnotu `value`.

Generování sady souborů zajišťuje koncový bod na adrese `/generate` a zvolená metoda je opět `POST`. K provedení generování celé sady jsou potřeba informace o souboru a seznam aplikovaných pravidel a míst v kódu. Tělo žádosti musí obsahovat:

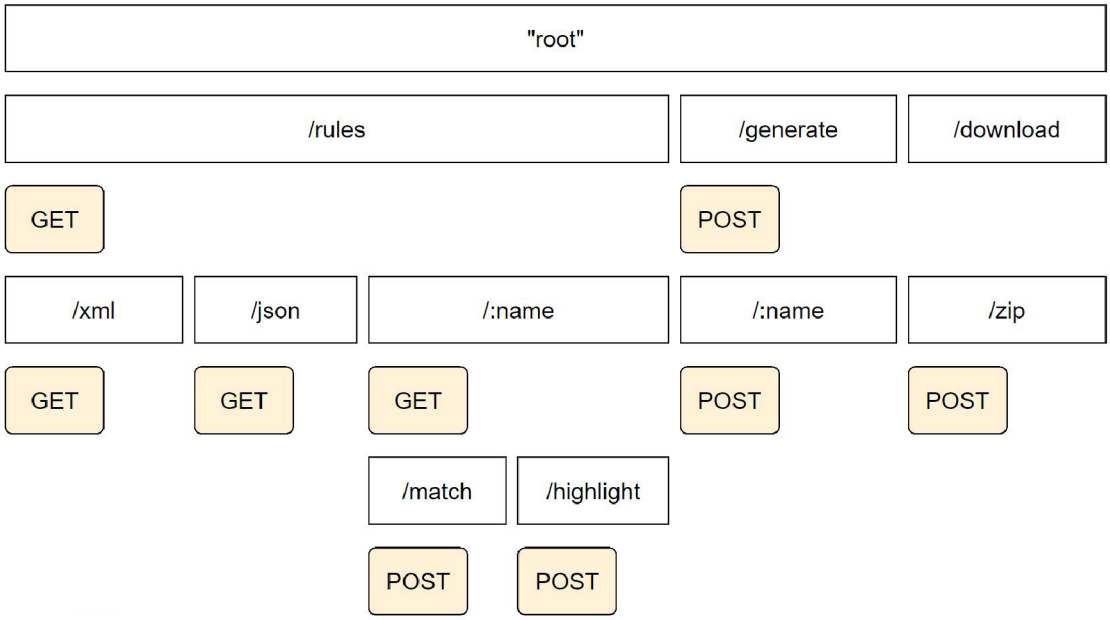
```
{
  file: {
    name: string,
    text: string,
    type: string
  },
  applied_rules: [
    {
      rule_name: string,
      apply: [
        { index: number, user_values: [ any ] },
        ...
      ]
    },
    ...
  ]
}
```

Odpověď obsahuje pole vygenerovaných souborů, které jsou reprezentovány objektem obsahujícím `name`, `text` a `type`, stejně jako je tomu v žádosti.

Vytvoření archivu

K vytvoření archivu slouží koncový bod na adrese `/download/zip`. Metodou `POST` jsou mu v těle žádosti zaslány soubory uspořádané do pole. Soubor je reprezentován stejným objektem, jako v případě generování. Vytvořený archiv je uložen na lokální úložiště serveru a v odpovědi odeslán jako `Blob`[4] objekt.

[4]Blob – Binary Large OBject, zdroj: https://developer.mozilla.org/en-US/docs/Web/API/Blob



Obrázek 4.5: Grafické znázornění koncových bodů REST rozhraní serveru. Každý koncový bod je definován adresou URL a metodou HTTP. Slouží pro poskytování dat a funkcionalit pro klientskou aplikaci. (zdroj: autor)

Kapitola 5

Implementace klientské a serverové aplikace

Klientská aplikace je implementována ve frameworku Angular, a to v nejnovější verzi 9. Angular používá TypeScript a poskytuje veškerou potřebnou funckionalitu pro realizaci uživatelského rozhraní. Serverová aplikace je napsána v JavaScriptovém frameworku Express. Podobnost obou zvolených programovacích jazyků ulehčí vývoj aplikace. Projekt byl nazván TreDaM - Tree Data Mutator.

5.1 Struktura klientské aplikace

Angular používá svůj vlastní příkazový řádek `ng` z balíčku `@angular/cli`. Kromě vytvoření a spuštění projektu slouží ke generování funkčních částí aplikace. Těch v Angularu existuje celá řada:

- **component** – je základní stavební kámen stránky. Skládá se z třídy v TypeScriptu, šablony v HTML, definice jednotkového (angl. `unit`) testu v TypeScriptu pro nástroj Jasmine a volitelného CSS souboru se styly.
- **service** – je součástí návrhového vzoru vkládání závislostí. Poskytuje data a funkce pro `component` a jiné `service`. Data mohou být předem definována, nebo asynchronně získána ze serveru.
- **module** – uspořádává spolu související `component` a `service` do modulů. Uspořádání zvyšuje efektivitu překladu projektu do JavaSriptu a za běhu aplikace efektivnější vkládání závislotí. Také volitelně umožňuje „lazy loading“, kdy k načtení částí dojde až v době zobrazení.
- **guard** – se volitelně používá při navigaci v aplikaci. Definuje podmínky, které je třeba splnit pro přístup na vybrané stránky.
- **pipe** – transformuje zobrazená data v šabloně `component`. Je možné použít i několik na sebe navazujících `pipe`.

Nově vytvořená aplikace definuje jen základní strukturu projektu. Pro samotné zdrojové kódy aplikace ale neudává žádnou. Vzhledem k různorodosti a množství částí aplikace je vhodné nějakou navrhnout. Struktura projektu, vytvořeného pomocí `ng`, je následující:

```
  |-- e2e
  |-- node_modules
  |-- src
    |-- app
    |-- assets
    |-- environments
```

Kořenový adresář obsahuje celou řadu souborů, mezi kterými je třeba konfigurace NPM, ale prozatím nejsou podstatné. Složka `e2e` obsahuje konfiguraci end-to-end testů nástroje Protractor. Lokálně uložené balíčky projektu se nachází v `node_modules`. Ve složce `src` najdeme vstupní soubory aplikace, jako `index.html`, `main.ts` a globální CSS styly. Doplňuje je `assets` s obrázky použitými v komponentách a definice běhových prostředí v `environments`.

Navržená struktura zdrojových kódů funkčních částí se však nachází v adresáři `app`. Inspirována je návrhem Mathise Garberga [16] a přizpůsobena specifickým požadavkům aplikace. Uspořádání je následující:

```
|-- components
|-- core
  |-- guards
  |-- services
  |-- footer
  |-- header
|-- pages
|-- shared
  |-- components
  |-- pipes
app-routing.module.ts
app.module.ts
```

Aplikace je uspořádána do jednoho modulu. Pro větší projekt by bylo vhodné zvážit modulů více. Jeho definici obsahuje soubor `app-module.ts` a skládá se pouze z deklarací funkčních částí a importů dalších užitečných modulů, které nabízí samotný Angular.

Složka `components` obsahuje komponenty, které používá právě jedna další komponenta. Příkladem je třeba výsledek syntaktické kontroly, který obsahuje jedna stránka a jeho logika je natolik komplexní, že je rozumné ji uzavřít do samostatné komponenty.

Klíčové části aplikace jsou ve složce `core`. Zde najdeme komponenty `header` a `footer` u kterých se dá bezpečně předpokládat, že budou na každé stránce. Názvy složek `services` a `guards` napovídají, že obsahují korespondující funkční části aplikace popsané na začátku sekce.

Speciální případ komponent je ve složce `pages`. Tyto komponenty odpovídají jednotlivým stránkám aplikace. Typicky se zkládají z celé řady dalších komponent a starají se o jejich responzivitu. O tom, kdy se má která komponenta vykreslit, rozhoduje `Router` popsaný dále v textu.

V `shared` jsou složky, jejichž obsah je znovupoužitelný ve více komponentách, ale ne natolik, aby byl součástí jádra aplikace. Příkladem je obsah `components`, kde jsou komponenty použité více než v jedné další komponentě. Následuje složka `pipes`, kde najdeme dekorátory dat `pipe`. Dekorátory taktéž používají šablony různých komponent.

Router

Router je modul definovaný v souboru `app-routing.module.ts`, který se stará o navigaci aplikace. Výhodou použití je, že URL adresa reflektuje uživateli jeho akce při navigování

v aplikaci, což ne všechny SPA[1] nabízejí. Jednotlivé cesty jsou zapsány do objektů, které popisují její chování. Co taková cesta obsahuje ilustruje příklad z projektu:

```
{
  path: 'mutate',
  component: MutateComponent,
  canActivate: [ SyntaxValidGuard ],
  pathMatch: 'full'
}
```

Cílové komponenty se v aplikaci vykreslují v `AppComponent`, která slouží jako „spojka“ mezi vstupním souborem `index.html` a Routerem. Místo v šabloně je určeno párovým tagem:

```
<router-outlet></router-outlet>
```

Mimo definice seznamu všech cest, Router exportuje seznam použitých komponent, který importuje modul aplikace. Tím je zajištěn jeden zdroj informací a zabráněno opakovanému importování stejných komponent.

Klasické navigování pomocí odkazů v aplikaci funguje, ale dojde při něm k obnově stránky. Proto se používá propojení s Routerem pomocí speciální syntaxe Angularu přímo v HTML elementu:

```
<div routerLink='/mutate'>Mutate</div>
```

Angular je natolik pokročilý, že není třeba programovat detekování události, při které uživatel klikl na odkaz. Potřebný kód se automaticky generuje při kompilaci a třída komponenty nemusí Router importovat.

5.2 Vytvoření a syntaktická kontrola vstupního souboru

Aplikace nemá tradiční hlavní stránku, kde bývají informace o webu. Místo toho je uživatel postaven přímo před editor souboru. Vytvořený soubor bude důležitý ve všech dalších částech aplikace. Pro sdílení dat mezi více komponentami je použita knihovna RxJS a návrhový vzor vkládání závislotí (angl. Dependency Injection – DI). Implementováno je to pomocí služby, kterou třída komponenty deklaruje v konstruktoru. Ukázka kódu DI:

```
constructor (
  private _inputFileService: InputFileService
) { }
```

Angular má vlastní DI framework `Injector`, který se stará o poskytování instancí služeb jednotlivým třídám. Služba `InputFileService` je registrována na úrovni aplikace pomocí dekorátoru `@Injectable()`. Její instance vzniká a zaniká společně s celou aplikací, stejně tak i data. Počítá se s tím, že uživatel práci s nástrojem dokončí během jednoho sezení. Pro ukládání rozpracované práce by bylo možné použít `localStorage`, nebo projekt uložit jako soubor.

Zmíněná služba definuje a exportuje několik rozhraní. Slouží k popisu struktury uchovávaných informací o vytvořeném souboru a výsledku syntaktické kontroly. Díky typovosti TypeScriptu je možné určit datové typy položek, což ilustrují následující kusy kódu:

```
export interface IInputFile {
  file: IFile,
  error: IError
}
```

```
export interface IFile {
  name: string,
  type: string,
  text: string
}
```

```
export interface IError {
  syntax_valid: boolean,
  message: string,
  line: number,
}
```

[1] SPA – Single Page Application, česky: Jednostránková aplikace.

Služba si v době inicializace vytvoří rozhraní `IInputFile` jako nový objekt `BehaviorSubject` z knihovny RxJS. Ten se komponentám předává pomocí funkce `asObservable()` jako pozorovatelný objekt. Pro přihlášení k datům se používá asynchronní funkce `subscribe()`, jejíž `callback` obsahuje data. Její návratový typ je `Subscription`, který je vhodné si uložit a na konci životního cyklu komponenty se z odběru odhlásit. Bez odhlášení dochází k únikům paměti.

Komponenta `EditComponent` je stránka pro editaci kódu a je složena z celé řady dalších komponent, se kterými sdílí část svých dat. Díky tomu nemusí mít každá komponenta vlastní DI, přesto je zaručeno, že změny hodnot se promítnou do všech ostatních komponent.

Kromě tříd je třeba hodnoty promítnout do HTML šablon. K tomu se používá obousměrné propojení pomocí následující konstrukce:

```
<textarea [(ngModel)]="input_file.file.text">{{ input_file.file.text }}</textarea>
```

Hodnota elementu je tak vždy shodná s hodnotou proměnné třídy. Propojení funguje z obou stran, když uživatel píše text (externě), a nebo když nahraje soubor (interně). Nahraný soubor komponenta předá službě, která obstarává čtení obsahu a aktualizaci současného souboru.

Syntaktická kontrola

Kontrolu syntaxe provádí služba `SyntaxValidationService`, kterou používá služba spravující vstupní soubor. Typ souboru je rozpoznán automaticky při nahrání, nebo když uživatel píše název souboru v editoru. Nově nahraný soubor je automaticky zkontrolován, jinak se kontrola provádí na vyžádání. Přechod na další stránky nástroje je podmíněn syntaktickou správností souboru. Odpovídající cesty Routeru tedy chrání `SyntaxValidGuard`. Pokud není soubor zkontrolován, provede kontrolu a při chybě zabrání přechodu.

Soubor typu JSON je kontrolován pomocí `JSON.parse()`. Metoda je součástí jazyka JavaScript. Pokusí se zpracovat text do JavaScriptového objektu, nebo hodnoty. Když uspěje, syntaxe je správná. Text typu XML je parsován pomocí instance `DOMParser()`. Ten je součástí každého webového prohlížeče a kontrola funkcí `parseFromString()` má obdobný charakter, jako v případě JSON.

Výsledek kontroly reprezentuje rozhraní `IError`. Pokud kontrola odhalí chybu, z hlášky je většinou možné získat i pozici v textu a dopočítat tedy chybný řádek. Formát chybové hlášky DOMParseru se liší podle prohlížeče, někdy ani neobsahuje pozici v textu. Pro každý prohlížeč je tedy třeba analýzu hlášky upravit.

K zobrazení výsledku kontroly slouží komponenta `SyntaxResultComponent`. Vyžaduje `IError` jako vstupní parametr pomocí dekorátoru `@Input()`. Dále implementuje funkci, která posune stránku tak, aby byl chybný řádek rozbrazen uprostřed obrazovky při kliknutí na výsledek chybné kontroly.

Zvýraznění syntaxe textu

Pro barevné zvýraznění textu kódu by bylo možné použít knihovnu Ace[2], kterou používá JSON Editor Online[3]. Knihovna je napsána v čistém JavaScriptu. Nevýhodou ale je, že HTML tag `textarea` překryje desítkami, občas i stovkami `div` elementů pro několik řádků dlouhý text. Všechny elementy jsou relativně pozicovány na přesný počet pixelů.

[2] Ace – dostupné z: https://ace.c9.io/
[3] JSON Editor Online – dostupné z: https://jsoneditoronline.org/

Nakonec bylo zvoleno vlastní řešení. Stačí k tomu jeden `div`, který překrývá původní element `textarea`. Obsahuje stejný text, ale v zápisu HTML syntaxe. Element nezachytává kurzor a text je průhledný. Předem definované sekce kódu, jako numerické hodnoty, jsou zvýrazněny vkládáním `span` elementů se stylem, který obarví vymezený text. Realizováno je to pomocí několika `pipe`, které jsou v Angularu navrženy pro transformaci textu:

```
<div #shadowDiv
...
[innerHTML]="input_file?.file?.text | code:input_file?.file?.type | htmlTags | sanitize">
</div>
```

Text souboru prvně transformuje `code`, která má jeden parametr a to je typ souboru. Otazníky v zápisu nejdříve kontrolují existenci objektu, než se pokusí přistoupit k položce. Do původního textu jsou tak vloženy `span` elementy. Další je `htmlTags`, která text transformuje do HTML podoby. Poslední je `sanitize`, která jenom výsledný text označí jako bezpečný. Jelikož se mění vnitřní část elementu za běhu aplikace, některé konstrukce nejsou z důvodu bezpečnosti dovoleny. Veškeré provedené změny jsou pouze v rukou programátora, bezpečnost tak není narušena.

5.3 Aplikování mutací s uživatelskými kritérii

Stránku pro aplikování mutací tvoří komponenta `MutateComponent`. Sdílení dat a používání služeb je stejné, jak je popsáno v předchozí sekci. Kromě obsahu vytvořeného souboru zobrazuje seznam mutačních pravidel dostupných pro zvolený typ souboru. K jejich získání používá službu `MutationRulesService`, která zajišťuje komunikaci se serverem pomocí modulu `HttpClient`. Získání dat probíhá asynchronně:

```
getRules(forFormat: string): Observable<Array<IRule>> {
  return this._http.get<Array<IRule>>(this.cookUrl(forFormat));
}
```

Funkce `cookUrl()` vytváří adresu koncového bodu serveru. Adresa je různá při vývoji aplikace a nasazení do produkce. Pro detekci prostředí Angular nabízí funkci `isDevMode()`.

Rozhraní pravidel odpovídá struktuře popsané v kapitole 4, ale je rozšířeno o dvě další položky `highlight` a `matches`. První z nich, `highlight`, obsahuje text s označenými místy v kódu, kde je možné provádět mutace, a `matches` je pole hodnot těchto míst. Hodnoty jsou získány ze serveru při první interakci uživatele s ovládacími prvky rozhraní a uloženy pro následující interakce.

Vytvořený soubor už nejde dále upravovat a kromě barevného zvýrazňování částí kódu je třeba také označit místa kódu pro vybrané pravidlo. O to se stará sdílená komponenta `FilePreviewComponent`, kterou také používá stránka s přehledem vygenerované sady souborů pro testování.

Uchovávání aplikovaných pravidel

Služba `MutationRulesService` definuje a exportuje rozhraní pro vytváření záznamů o aplikovaných pravidlech a souvisejících místech kódu. Rozhraní jsou následující:

```
export interface IMutationRecord {
  rule_name: string,
  apply: Array<IApply>
}
```

```
export interface IApply {
  index: number,
  user_values: Array<any>
}
```

Všechny záznamy `IMutationRecord` jsou uspořádány do pole. Služba také implementuje podpůrné funkce pro vytváření záznamů, přidávání záznamů o aplikovaných místech v kódu a přidávání/odebírání hodnot. O propojení těchto funkcí s ovládacími prvky rozhraní se stará komponenta `RuleOptionsComponent`.

Uživatelská kritéria

Uživatel si na stránce zvolí pravidlo a zvýrazní se místa textu, kde je možné ho aplikovat. Místo vybere kliknutím přímo na zvýrazněný kus kódu, nebo použije navigační šipky, a poté má možnost zde pravidlo aplikovat. Volitelnými kritérii pro generování je myšlena všechna funkcionalita navíc dostupná po tomto kroku. Aplikace nabízí možnost předgenerování hodnot pomocí služby `GenerateService`. Slouží k tomu následující funkce:

```
getPreGeneratedValues(forRule: IRule, value: any): Observable<Array<any>> {
  const body = {value: value};
  const { name } = forRule;

  return this._http.post<Array<any>>(this.getBaseUrl()+`generate/${name}`, body);
}
```

Uživatel také může přidávat vlastní hodnoty. Všechny hodnoty také může odstranit a pokud existuje alespoň jedna, při generování testovací sady je potlačeno výchozí generování hodnot.

5.4 Struktura serverové aplikace

Ke spuštění aplikace je použita knihovna Nodemon. Umožňuje automatické restartování pokaždé, když dojde ke změně ve zdrojových souborech. To je užitečné při implementaci, ale také umožňuje automaticky načítat nově přidaná mutační pravidla. V repozitáři serverové aplikace byla vytvořena tato struktura:

```
|-- api
  |-- logic
  |-- routes
|-- downloadable-zip
|-- node_modules
|-- res
  |-- rules
  loader.js
  rule-validation-schema.js
app.js
server.js
```

Ve zbytku sekce jsou popsány jen některé soubory a složky. Ty nezmíněné jsou podstatné pro další sekce textu a popsány přímo v nich.

Vstupní soubor aplikace je `server.js`. Vytváří a spouští instanci serveru. Port aplikace je načten z proměnné prostředí `PORT`. Pokud neexistuje, je zvolen 3000.

Soubor `app.js` implementuje a exportuje router frameworku Express. Do odpovědi každého příchozího požadavku přidává CORS hlavičky, aby umožnil komunikaci s klienty na jiné doméně. Dále požadavky rozděluje mezi ostatní routery ve složce `api/routes`. Také definuje chybovou odpověď, která nastane, pokud adresa není zpracována žádným z předchozích routerů, a middleware, který udává jednotný formát všech chybových hlášek.

5.5 Načítání a validace mutačních pravidel

Každé mutační pravidlo je zapsáno do samostatného souboru a načítání probíhá při spuštění serveru. Soubor `loader.js` postupně načítá všechny soubory ve složce `res/rules`. Obsah každého pravidla je transformován pomocí `JSON.parse()` na JavaScriptový objekt.

Ke kontrole struktury pravidel je použita knihovna Joi[4]. Kromě pravidel je možné ji použít ke kontrole přijatých dat v těle požadavku. Metoda `Joi.validate()` přebírá objekt k ověření a schéma, které pomocí plynulého programového rozhraní popisuje strukturu objektu. Schéma pro kontrolu pravidel exportuje soubor `rule-validation-schema.js` a má následující podobu:

```
  const schema = {
    usages: Joi.array().min(1).items( Joi.string().min(1) ).required(),
    name: Joi.string().min(3).regex(/^[a-zA-Z_]+$/).required(),
    friendlyName: Joi.string().min(3).required(),
    description: Joi.string().min(10).required(),
    pattern: Joi.string().required(),
    transform: Joi.func().required()
};
```

Položka `transform` je v souboru uložena jako řetězcový literál, ale schéma ji ověřuje jako funkci. Před kontrolou je třeba ji vyhodnotit pomocí funkce `eval()`. Výhodou knihovny je, že při neúspěšné kontrole metoda vrací chyby v dobře čitelném zápisu. Výsledkem načítání je kolekce pravidel. Soubor `loader.js` exportuje funkci, která vrací kolekci, nebo její podmnožinu na základě filtru.

Pravidla poskytuje více koncových bodů serverové aplikace. Protože se za chodu nemění (při změně dojde k restartování), tak routery seznam pravidel ukládají do konstant pro optimalizaci. Příklad vytvoření koncového bodu routeru s asynchronním zpracováním:

```
const loader = require('../../res/loader');
const allRules = loader('all');

router.get('/', (req, res, next) => {
    res.status(200).json(allRules);
});
```

5.6 Generování testovacích dat

Aplikace nabízí dva různé koncové body pro generování dat. Jeden generuje data na základě jména pravidla a jedné hodnoty. Slouží předgenerování hodnot přímo v kódu a implementace logiky je přímo v routeru. Stačí vyhledat pravidlo v kolekci všech pravidel, vyhodnotit transformační funkci s předáním parametru a výsledek odeslat zpět. Ukázka kódu:

```
router.post('/:name', (req, res, next) => {
  const { name } = req.params;
  const { value } = req.body;

  const rule = rules.find(rule => rule.name === name);
  if (rule && value !== undefined) {
      res.status(200).json(eval(rule.transform)(value));
  } else {
      res.status(400).json({message: `Rule "${name}" not found or invalid data send.`});
  }
```

[4] Joi – dostupné z: https://www.npmjs.com/package/@hapi/joi

```
});
```

Druhý koncový bod generuje celou sadu souborů na základě seznamu aplikovaných pravidel. Router k tomu používá funkci `generate()`, kterou exportuje soubor `fileset-generator.js`. Pro každé aplikované pravidlo se opět získá skutečné pravidlo z kolekce na základě shody jména a poté se pro něj vygeneruje sada souborů. Kód vypadá takto:

```
function generate(file, mutation_records) {
  let file_set = [];

  mutation_records.forEach(record => {
      const rule = rules.find(item => item.name === record.rule_name);
      if (!rule) return;

      const generated_files = generateForRuleRecord(rule, file, record.apply);

      file_set.push(...generated_files);
  });

  return file_set;
}
```

Funkce `generateForRuleRecord()` prochází jednotlivá místa v souboru, kde jsou mutace aplikovány. Pokud uživatel nezadal žádné vlastní hodnoty a ani žádné nepředgeneroval, provede se dodatečné vygenerování. Poté se pro každou hodnotu vytvoří nový soubor, obsahující mutaci textu, a přidá se do sady souborů. Název souboru je odvozen z názvu pravidla, pořadí a originálního souboru.

Kapitola 6

Testování a nasazení aplikací

V kapitole jsou popsány zvolené metody testování webového rozhraní. Angular nabízí několik nástrojů pro testování, které jsou v projektu použity. Zmíněno je i automatizování testů pomocí GitLab CI/CD. Druhá sekce obsahuje informace o produkčním nasazení obou aplikací pomocí Dockeru. Pomocí Dockeru jsou obě spuštěny na serverech Heroku v bezplatně dostupném plánu.

6.1 Automatizované testování webového rozhraní

Klientská aplikace je testována pomocí funkčních testů. Testy kontrolují, zda implementované části pracují správně a že odpovídají požadavkům. Také pomáhají udržovat konzistenci při implementaci tím, že ověřují, zda nové funkce nenarušují správné fungování již existujících.

Unit testy

Unit testy slouží k otestování jednotlivých částí systému. V projektu je použit testovací framework Jasmine [1]. Součástí každé vygenerované funkční části pomocí příkazu `ng generate` je TypeScriptový soubor s příponou `specs.ts`. O samotné spouštění a vyhodnocování testů se stará prostředí Karma[2]. Karmu je možné používat z příkazového řádku, nebo pomocí webového prostředí s debugovacími nástroji.

Hlavní částí každého testu je funkce `describe()`, které přebírá dva parametry - název sady testů a funkci. Ve funkci jsou deklarovány jednotlivé testy. Sada se skládá z „životních“ cyklů. Před každým testem se provádí sekvence funkcí `beforeEach()`, které slouží k importování dalších funkčních částí, kompilaci a inicializaci závislostí (externích zdrojů). Testy jsou popsány funkcí `it()`. Takový test může vypadat následovně:

```
it('should have title h6', () => {
  expect(document.querySelector('.h6').innerHTML).toEqual('Syntax error:');
});
```

Test se pokusí naleznout element obsahující `class="h6"` a kontroluje, zda se uvnitř nachází text obsahující danou sekvenci. Funkce `querySelector()` používá selektor podobný CSS. Třídy jsou označeny tečkou a identifikátory mřížkou. Umožňuje i plynule vybrat vnořené elementy. Například získání `div` elementu uvnitř elementu s identifikátorem by vypadalo takto:

[1] Jasmine – dostupné z: https://jasmine.github.io/
[2] Karma – dostupné z: https://karma-runner.github.io/latest/index.html

```
document.querySelector('#someId > div');
```

Pokud testujeme jednoduchou komponentu, pro vybrání kořenového elementu je možné použít `fixture.nativeElement`. Fixture je vytvořené prostředí pro testovanou komponentu. Obsahuje důležitou metodu `detectChanges()`, která se musí použí, když chceme měnit vnitřní stav komponenty. Použití ilustruje následující kód:

```
it('should display success', () => {
  component.error = { syntax_valid: true, message: null, line: 0 };
  fixture.detectChanges();
  expect(fixture.nativeElement.innerText).toContain('Syntax is valid.');
});
```

Po každém testu se provede funkce `afterEach()`, která není tak často vyžita a ne každý test je obsahuje.

Pokud komponenta používá nějaký modul, musí ho importovat. Některé moduly Angularu mají vlastní verze určené k testování. Například `HttpClientTestingModule` slouží pro komponenty používající `Http` modul pro komunikaci se serverem.

Testy end-to-end

Testy end-to-end slouží k otestování funkcinality stránek jako celku. Cílem je testovat aplikaci podobným stylem, jako by ji používal skutečný uživatel. Jak plyne z názvu, aplikaci je možné v testu projít od začátku do konce a přitom testovat různé interakce.

V projektu je použit framework Protractor[3], který je součástí Angularu. Podporuje testovací framework Jasmine a pro spouštění používá Selenium server. Strukturu ilustruje obrázek 6.1. V kořenovém adresáři klientské aplikace se nachází složka `e2e` obsahující konfigurační soubory a složku s testy. Ve složce je soubor `app.po.ts`, který popisuje objekt stránky. Některé prvky aplikace se budou používat opakovaně a proto objekt stránky obsahuje metody, které usnadňují k takovým prvkům přístup.

Samotné definice testů jsou v souboru `app.e2e-spec.ts`. Jako u unit testů se setkáme s fukcí `describe()`. Zde se používá pro simulaci interakce, při které se uživatel dostane z jedné části aplikace do další. Předlohou testů může být diagram aktivit 2.1 v kapitole 2. Kromě navigačních prvků stránky je možné použít instanci prohlížeče. Ukázka kódu:

```
navigateTo(adress: string) {
  return browser.get('/' + adress) as Promise<any>;
}
```

K získání elementů webové stránky Protraktor používá vlastní rozhraní, které je plynulejší, než v případě instance `document`. Získání prvku, a jeho textové hodnoty, podle názvu CCS třídy vypadá následovně:

```
getErrorText() {
  return element(by.className('alert-danger')).getText() as Promise<string>;
}
```

Testování probíhá stejným způsobem, jako u unit testů. Opět se setkáme s životními cykly. Funkce `beforeEach()` před každým testem vytvoří instanci objektu stránky a provede přesměrování na výchozí stránku testů. Jednotlivé testy mohou vypadat takto:

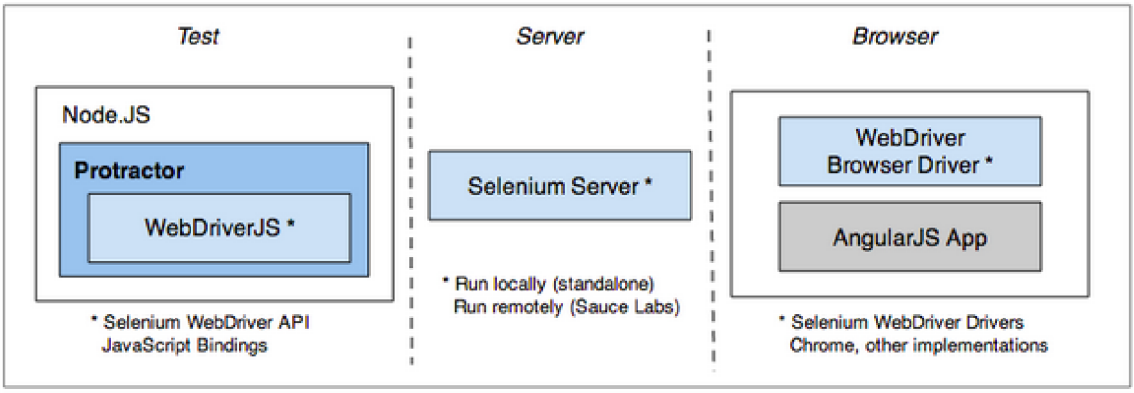
```
it('should validate correct JSON syntax', () => {
  page.writeFileName('soubor.json');
  page.writeJsonFile({ key: 1 });
```

[3]Protractor – dostupné z: https://www.protractortest.org/#/

```
    page.validateSyntax();

    expect(page.getSuccesText()).toBeTruthy();
});
```

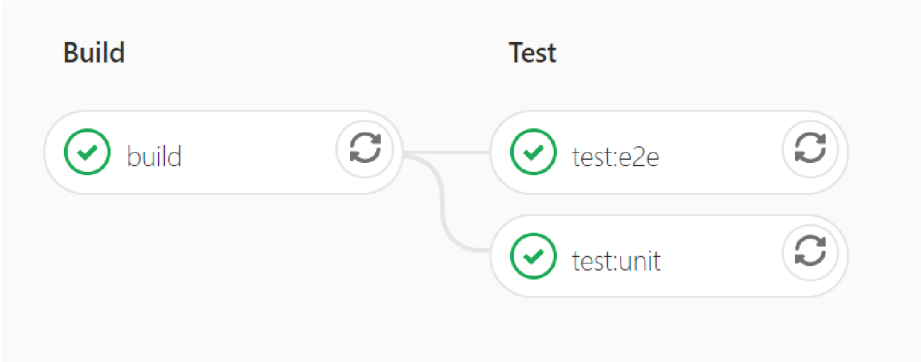
Protractor podporuje celou řadu funkcí pro interakci s prvky, jako kliknutí, psaní textu, posouvání stránek a umí i pořizovat snímky obrazovky.



Obrázek 6.1: Protraktor běží v prostředí Node.js a obaluje WebDriver.js. Na obrázku je vidět interakce mezi jednotlivými částmi systému. (zdroj: https://www.protractortest.org/#/infrastructure)

GitLab CI/CD

V projektu jsou automatizované testy spojeny s nástrojem GitLabu[4]. GitLab CI/CD při každé aktualizaci vzdáleného repozitáře provede etapy definované v souboru `.gitlab-ci.yml`. Nachází se vždy v kořenové složce aplikace a definované etapy pro projekt ilustruje obrázek 6.2.



Obrázek 6.2: Etapy GitLab CI/CD jsou rozděleny do fází. Fáze probíhají sekvenčně a pokud jedna neuspěje, následující jsou přeskočeny. (zdroj: autor)

Před každou etapou je provedena instalace knihoven příkazem `npm ci --silent`. Etapa **build** aplikaci přeloží a následně spustí. Zda aplikace běží je ověřeno příkazem `curl` s parametrem obsahujícím adresu úvodní stránky.

[4]GitLab – dostupné z: https://gitlab.com/

V etapě **test:unit** jsou spuštěny jednotkové testy pomocí nástrojů Jasmine a Karma. Před samotným testem je ale třeba doinstalovat prohlížeč Chromium a nastavit ho jako „root wrapper“. I když aplikaci spouští běžný uživatel, aplikace se chová, jako by ji spustil uživatel privilegovaný, ale s co nejbezpečnějším možným chováním. Samotné testy jsou spuštěny v prostředí Chromia bez grafického rozhraní.

Testování end-to-end provádí etapa **test:e2e** nástrojem Protractor. Místo Chromia je použit prohlížeč Chrome a spuštení testů je podmíněno zachytáváním obrazovky prohlížeče. K tomu slouží server `Xvfb` s virtuální obrazovkou o rozlišení 1920 × 1080 pixelů. Po vytvoření obrazovky se stáhne Selenium server a ovladačemi pro Chrome. Testování se spouští příkazem `node_modules/.bin/ng e2e`.

6.2 Docker a konfigurace produkčního nasazení

Projekt je rozdělen do dvou hlavních repozitářů. Klientská a serverová aplikace má vlastní nastavení knihoven, prostředí a také vlastní konfiguraci Dockeru. V době vývoje obou aplikací se používají nástroje, které umožňují automatické přeložení a spuštění při změně zdrojového kódu a jiné užitečné knihovny. Pro produkční nasazení ale nejsou podstatné a používá se jiná konfigurace.

Klientská aplikace

Aplikace vytvořená v Angularu obsahuje konfiguraci pro produkční nasazení a bez specifických požadavků ji není třeba nějak upravovat. Jelikož je projekt napsaný v TypeScriptu, překlad do JavaScriptu, a spojování jednotlivých modulů, probíhá znatelně delší dobu. K překladu slouží příkaz `ng build --prod`.

Konfigurace Dockeru

Konfiguraci obsahuje soubor `Dockerfile` a vytvoření výsledného obrazu disku (dále jen „image“) se skládá ze dvou fází.

První slouží k vytvoření zdrojových souborů pro produkci. Je založena na image obsahující běhové prostředí Node.js verze 13. Projekt se zkopíruje s výjimkou souborů a adresářů specifikovaných v souboru `.dockerignore`. Většinou se jedná testy a knihovny. Knihovny se instalují dodatečně pomocí `npm ci`. Jedná se o „čistou“ instalaci, která je rychlejší a vynechává knihovny používané jen při vývoji. Překlad probíhá v příkazovém řádku Angularu `ng`. Aby se nemusel instalovat globálně, je použit přímo ze složky knihoven `node_modules/.bin/ng`.

Druhá fáze obstarává běhové prostředí aplikace. Založena je na image obsahující webový server Nginx. Z předchozí fáze se zkopíruje jenom složka s vytvořenými zdrojovými soubory. Součástí projektu jsou dva konfigurační soubory serveru `server.conf` a `heroku.conf`. Nginx má vlastní chybové stránky, například 404 a 500. Oba soubory umožňují používat chybové stránky aplikace. Liší se však v konfiguraci portu aplikace. Soubor `server.conf` používá běžný port 80. Pokud má ale aplikace běžet v kontejneru na nějakém serveru, port si často nemůže zvolit. Proto soubor `heroku.conf` používá číslo portu uložené v proměnné prostředí `PORT`. Soubor je použit při nasazení projektu na stránkách Heroku[5].

[5]Heroku - dostupné z: https://www.heroku.com/

Serverová aplikace

Serverová aplikace je v porovnání s klientskou značně menší. Produkční nasazení se od vývojového prostředí skoro neliší. Konfigurace Dockeru má jen jednu fázi a výsledný image je založen na prostředí Node.js verze 13. Projekt se zkopíruje na image a proběhne instalace knihoven pomocí `npm ci`. Při vytvoření kontejneru se server automaticky spustí příkazem `npm start`, který je přidán do souboru `package.json`. Aplikace běží na čísle portu přiřazeném v proměnné prostředí `PORT`. Pokud není deklarována, zvolena je konkrétní hodnota. Kód vypadá takto:

```
const port = process.env.PORT || 3000;
```

Kapitola 7

Závěr

Cílem práce bylo vytvořit webovou aplikaci pro generování testovacích dat stromových struktur s kritérii dodanými uživatelem. Projekt se skládá ze serverové a klientské aplikace.

Klientská aplikace je implementována v moderním frameworku Angular. Nabízí uživateli grafické rozhraní pro vytvoření, editaci a syntaktickou kontrolu souborů typu JSON a XML. Dále slouží k aplikování mutačních pravidel na konkrétní místa v kódu. Volitelně lze hodnoty předgenerovat, přidat vlastní, nebo obojí. Podle aplikovaných pravidel je vygenerována sada testovacích souborů, ze které lze volitelně odebrat vybrané soubory. Výsledou sadu je možné stáhnout jako archivní soubor typu ZIP.

Součástí řešení je návrh a vytvoření sady pravidel, které definují chování mutací. O jejich validaci, a poskytování klientovi, se stará serverová aplikace postavená na prostředí Node.js a frameworku Express. Ke komunikaci používá REST rozhraní. Kromě generování testovací sady také poskytuje potřebnou funkcionalitu pro práci s textem a pravidly.

Rozhraní webové aplikace je testováno pomocí unit a end-to-end testů a je podpořené nástrojem GitLab CI/CD. Obě aplikace obsahují konfiguraci pro Docker, pomocí kterého lze aplikace nasadit do produkčního prostředí. Projekt byl nasazen na servery Heroku a můžou ho využít testeři a nebo IT nadšenci, kteří hledají podobný nástroj, nebo jenom inspiraci.

Projekt by mohl obsahovat více testů a zapracovat by se dalo i na programové dokumentaci. V budoucnu by nástroj určitě obohatila možnost uložení rozpracovaného projektu do souboru, aby v ní uživatel mohl pracovat později. Aplikace by také šla upravit, aby pracovala i v offline režimu jako progresivní webová aplikace.

Literatura

[1] *Final draft ECMA-404* [online]. 2. vyd. Ecma International, 2017 [cit. 24. 1. 2020]. Dostupné z: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[2] Most Popular Technologies. *Developer Survey Results 2019* [online]. Stack Oveflow, 2019 [cit. 18. 3. 2020]. Dostupné z: https://insights.stackoverflow.com/survey/2019#development-environments-and-tools.

[3] *Angular* [online]. Google, 2020 [cit. 10. 3. 2020]. Dostupné z: https://angular.io/.

[4] *Docker* [online]. Docker Inc., 2020 [cit. 13. 3. 2020]. Dostupné z: https://www.docker.com/.

[5] Get Started. *Docker Docs* [online]. Docker Inc., 2020 [cit. 13. 3. 2020]. Dostupné z: https://docs.docker.com/get-started/.

[6] Fast, unopinionated, minimalist web framework for Node.js. *Express* [online]. 2020 [cit. 11. 3. 2020]. Dostupné z: https://expressjs.com/.

[7] JavaScript. *MDN web docs* [online]. 2020 [cit. 8. 3. 2020]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[8] About Node.js. *Node.js* [online]. Joyent, 2020 [cit. 11. 3. 2020]. Dostupné z: https://nodejs.org/en/about/.

[9] About npm. *Npm Documentation* [online]. 2020 [cit. 8. 3. 2020]. Dostupné z: https://docs.npmjs.com/about-npm/.

[10] *TypeScript* [online]. Microsoft, 2020 [cit. 10. 3. 2020]. Dostupné z: https://www.typescriptlang.org/.

[11] Banks, A. a Porcello, E. *Learning React*. O'Reilly Media, Inc., 2017. ISBN 9781491954621.

[12] Bessayah, F., Cavalli, A., Maja, W., Martins, E. a Valenti, A. W. A Fault Injection Tool for Testing Web Services Composition. In:. Berlin, Heidelberg: Springer-Verlag, 2010, sv. 6303, s. 137–146. DOI: 10.1007/978-3-642-15585-7_13.

[13] Bray, T., Paoli, J., Sperberg McQueen, C. M. a Maler, E. *Extensible Markup Language (XML) 1.0* [online]. W3C Recommendation, 2000 [cit. 24. 1. 2020]. Dostupné z: http://www.renderx.com/~renderx/Demos/fo2html/xml.pdf.

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

[15] Franzotte, L. a Vergilio, S. Applying Mutation Testing in XML Schemas. In: *18th International Conference on Software Engineering & Knowledge Engineering (SEKE'2006)* [online]. Leden 2006, s. 511–516 [cit. 13. 4. 2020]. Dostupné z: https://www.researchgate.net/publication/221390157_Applying_Mutation_Testing_in_XML_Schemas.

[16] Garberg, M. *How to define a highly scalable folder structure for your Angular project* [online]. Medium, 2018 [cit. 28. 4. 2020]. Dostupné z: https://itnext.io/choosing-a-highly-scalable-folder-structure-in-angular-d987de65ec7.

[17] Jian Bing Li a Miller, J. Testing the semantics of W3C XML schema. In: *29th Annual International Computer Software and Applications Conference (COMPSAC'05)* [online]. 2005, sv. 2, s. 443–448 [cit. 13. 4. 2020]. ISBN 0-7695-2413-3. Dostupné z: https://ieeexplore.ieee.org/document/151006.

[18] Jindal, R. Web Development Life Cycle by a Professional Website Design & Development Company. *Signity Solutions* [online]. 2019 [cit. 5. 3. 2020]. Dostupné z: https://www.signitysolutions.com/blog/web-development-life-cycle/.

[19] Marai, G. E. Activity-Centered Domain Characterization for Problem-Driven Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*. 2018, sv. 24, č. 1, s. 913–922. ISSN 1077-2626.

[20] Pan, J. *Software Testing* [online]. Carnegie Mellon University, 1999 [cit. 1. 2. 2020]. Dostupné z: http://users.ece.cmu.edu/~koopman/des_s99/sw_testing/.

[21] Reales, P., Polo, M., Fernández-Alemán, J. L., Toval, A. a Piattini, M. Mutation Testing. *IEEE Software* [online]. IEEE. 2014, sv. 31, č. 3, s. 30–35, [cit. 1. 2. 2020]. DOI: 10.1109/MS.2014.68. Dostupné z: https://ieeexplore-ieee-org.ezproxy.lib.vutbr.cz/document/6802989.

[22] Slater, R. *Fault Injection* [online]. Carnegie Mellon University, 1998 [cit. 1. 2. 2020]. Dostupné z: http://users.ece.cmu.edu/~koopman/des_s99/fault_injection/index.html.

[23] Slobojan, R. *InfoQ Explores: REST* [online]. InfoQ, únor 2010 [cit. 16. 3. 2020]. Dostupné z: https://ress.infoq.com/minibooks/emag-03-2010-rest/en/pdf/ResteMag.pdf.

Příloha A

Obsah přiloženého paměťového média

thesis/ – Zdrojové soubory textu práce s včetně výsledného PDF.

tredam/ – Společná složka obou aplikací.

tredam/tredam-frontend/ – Zdrojové soubory klientské aplikace.

tredam/tredam-backend/ – Zdrojové soubory serverové aplikace.

Příloha B

Manuál

Pro spuštění obou aplikací musí být na systému nainstalován Node.js s verzí alespoň 12.13.

Klientská aplikace

Ve složce zdrojových souborů klientské aplikace je třeba doinstalovat balíčky pomocí následujícího příkazu:

```
npm install
```

Spuštění vyžaduje Angular CLI. Je možné ho nainstalovat na systém globálně:

```
npm install -g @angular/cli
```

Poté se aplikace spustí příkazem:

```
ng serve
```

Aplikaci je také možné spustit přímo pomocí nainstalovaných balíčků bez globální instalace Angular CLI:

```
node_modules/.bin/ng serve
```

Serverová aplikace

Ve složce zdrojových souborů serverové aplikace je třeba doinstalovat balíčky pomocí následujícího příkazu:

```
npm install
```

Poté se aplikace spouští příkazem:

```
npm start
```

Příloha C

Docker manuál

Pro spuštění obou aplikací je možné použít nástroj Docker, pokud je na systému nainstalovaný. Aplikace používají konfigureaci produkčního prostředí.

Klienstská aplikace

Vytvoření image ve složce zdrojových souborů klientské aplikace:

```
docker build -t tredam-frontend-image .
```

Spuštění kontejneru na portu 4200:

```
docker run --name tredam-frontend-container -d
-p 4200:80 tredam-frontend-image
```

Webový server je nakonfigutován na port 80. Pokud je potřeba číslo portu předat pomocí proměnné prostředí `PORT`, je nutné upravit soubor `Dockerfile`. V souboru jsou označené řádky, které je třeba odkomentovat a zakomentovat.

Aplikace také provádí dotazy na server na adrese `https://tredam-api.herokuapp.com/`. Pro změnu adresy produkčního serveru je třeba upravit jeden řádek služby `AdressService`.

Serverová aplikace

Vytvoření image ve složce zdrojových souborů serverové aplikace:

```
docker build -t tredam-backend-image .
```

Spuštění kontejneru na portu 3000:

```
docker run --name tredam-backend-container -d
-p 3000:3000 tredam-backend-image
```

Příloha D

Odkazy na Heroku

Obě aplikace jsou nasazeny na serverech Heroku pomocí bezplatného plánu. Pokud není aplikace v posledních 30 minutách aktivně používána, dojde k jejímu uspání. Probuzení může trvat i desítky vteřin, což se projeví při příchodu do klientské aplikace a při prvním dotazu na server.

Klientská aplikace:

`https://tredam.herokuapp.com/edit`

Serverová aplikace:

`https://tredam-api.herokuapp.com/rules`