VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

WEBOVÝ PROHLÍŽEČ REPORTŮ ANALÝZY ZDROJOVÝCH KÓDŮ

WEB-VIEWER OF REPORTS OF SOURCE CODE ANALYSIS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JAKUB DOLEJŠÍ**
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:	**Dolejší Jakub**
Program:	Informační technologie
Název:	**Webový prohlížeč reportů analýzy zdrojových kódů** **Web-Viewer of Reports of Source Code Analysis**
Kategorie:	Analýza a testování softwaru

Zadání:

1. Seznamte se s volně dostupnými moduly pro zobrazování zdrojových kódů na webových stránkách.
2. Analyzujte požadavky na interaktivní prohlížení zdrojových kódů pro účely revize. Specifikujte požadavky na report analýzy zdrojových kódů pro účely revize zdrojových kódů. Navrhněte aplikaci pro interaktivní revizi zdrojových kódů na základě reportu z analýzy zdrojových kódů.
3. Implementujte webovou aplikaci pro interaktivní revizi zdrojových kódů.
4. Vytvořte umělou sadu zdrojových kódů a ukázkové reporty pro účely demonstrace implementované aplikace.
5. Navrhněte testovací sadu grafického uživatelského rozhraní implementované aplikace.

Literatura:

- Domovská stránka projektu hilite: https://github.com/alexkay/hilite.me
- Domovská stránka projektu Ace: https://ace.c9.io/
- OWASP Foundation: Přehled statických analyzátorů kódů: https://www.owasp.org/index.php/Static_Code_Analysis

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání

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:	31. července 2020
Datum schválení:	31. října 2019

Abstrakt

Tato bakalářská práce se zabývá návrhem a vývojem webové aplikace RepView. Tento nástroj slouží k interaktivní revizi zdrojových kódů na základě příslušného reportu analýzy zdrojových kódů. Aplikace se skládá ze dvou oddělených služeb, které běží v separátních kontejnerech technologie Docker. Cílem aplikace je usnadnit interpretaci reportu a jeho vazbu na zdrojové kódy. Výsledku je dosaženo za použití moderních webových technologií (Vuejs, Quasar), které umožňují provést uživatelsky přívětivou revizi zdrojových kódů.

Abstract

This bachelor thesis deals with design and development of web a application named RepView. The tool is used for interactive revision of source code based on a report of the source code analysis. The Application contains two main services that are running in separate docker containers. The main goal of the application is to simplify interpretation of a report and it's context with source code. The result is achieved by using modern web technologies (Vuejs, Quasar), which allow perform friendly source code revision.

Klíčová slova

interaktivní prohlížeč reportů, revize zdrojového kódu, REST API, Vue.js, Flask, Docker, jednostránková aplikace, RepView

Keywords

interactive report viewer, revision of source code, REST API, Vue.js, Flask, Docker, single page applicaton, RepView

Citace

DOLEJŠÍ, Jakub. *Webový prohlížeč reportů analýzy zdrojových kódů.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Webový prohlížeč reportů analýzy zdrojových kódů

Prohlášení

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

....
Jakub Dolejší
29. července 2020

Poděkování

Rád by jsem poděkoval vedoucímu práce, Ing. Aleši Smrčkovi, Ph.D za konzultace, cenné rady a vedení práce.

Obsah

1 Úvod **3**

2 Přístupy návrhu webových aplikací **4**
2.1 Více stránkové aplikace 4
2.2 Jednostránkové aplikace 4
2.3 Použité technologie 6

3 Analýza a specifikace požadavků **13**
3.1 Specifikace využití aplikace 13
3.2 Požadavky na prohlížení zdrojových kódů 14
3.3 Požadavky na server 15
3.4 Požadavky na aplikaci 15

4 Návrh aplikace **16**
4.1 Konceptuální návrh 16
4.2 Návrh frontendu 20
4.3 Návrh backendu 21

5 Implementace prohlížeče reportů **26**
5.1 Frontend 26
5.2 Backend 30
5.3 Spuštění aplikace 33

6 Uživatelské rozhraní webové aplikace **35**
6.1 Wireframe 35
6.2 Návrh grafického rozhraní 36
6.3 Výsledné řešení 38

7 Testování **43**
7.1 Automatizované testování 43
7.2 Manuální testování 45
7.3 Vývojové nástroje 46

8 Závěr **48**

Literatura **49**

A Obsah přiloženého média **51**

B Ukázková sada reportů **52**

C Módy zobrazení GUI **55**

Kapitola 1

Úvod

Tato práce popisuje návrh, implementaci a tvorbu webové aplikace **RepView**, která slouží k interaktivnímu prohlížení zdrojových kódů na základě příslušného reportu. Report zde představuje výstup z nástroje provádějícího statickou analýzu nebo běhovou chybu zvanou traceback. Tyto reporty zpravidla obsahují značné množství informací, přičemž ne všechny mohou být pro uživatele podstatné.

Cílem této práce je poskytnout uživateli možnost provést přehlednou revizi zdrojových kódů s vyznačením důležitých částí ve zdrojovém kódu na základě informací obsažených v reportu. To může být vhodné pro demonstrační či edukativní účely. Dále, nástroj může usnadňovat interpretace reportu a jeho souvislosti se zdrojovými kódy začínajícím programátorům, kteří mohou mít problém report správně interpretovat. Aplikace cílí na provedení této revize přes webové rozhraní rychle a interaktivně, bez nutnosti přihlašování. Interaktivnost je podpořena možností dynamického zobrazení podrobnějších informací o reportovaných řádcích, libovolného přecházením mezi nahranými soubory, uchováním si uživatelské historie či možností sdílení nahraných souborů přes webové rozhraní. Veškeré akce prováděné přes webové rozhraní probíhají asynchronně, bez nutnosti obnovení stránky.

Kapitola 2 se věnuje vývoji webových aplikací, a to zejména jednostránkových aplikací, což je kategorie webových aplikací, do kterých tato práce spadá. Dále jsou popsány nejdůležitější použité technologie, jazyky a knihovny, které v práci byly použity. Kapitola 3 se zabývá analýzou a specifikací požadavků. Zde jsou uvedeny požadavky, které jsou na aplikace kladeny. Tyto požadavky pokrývají nejen samotnou aplikaci, ale i její konkrétní části. Kapitola 4 obsahuje návrh aplikace, a to hned v několika rovinách. Zde je popsán konceptuální návrh aplikace, rozdělení služeb aplikace do kontejnerů a samotný návrh těchto služeb. V kapitole 5 je popsána implementace aplikace. Podobně jako v kapitole 4, i zde je text rozdělen na popis implementace jednotlivých služeb. Kapitola 6 obsahuje návrh, tvorbu a implementaci uživatelského rozhraní. Jsou zde popsány postupy, které byly aplikovány při tvorbě grafického uživatelského rozhraní. Kapitola 7 shrnuje testovací praktiky, které byly využity pro řádné otestování aplikace, a to na několika úrovních. Poslední, 8. kapitola shrnuje dosažené výsledky včetně návrhů na možná rozšíření.

Kapitola 2

Přístupy návrhu webových aplikací

Při vývoji webových aplikací je v dnešní době možné využít dvou nejpoužívanějších návrhových vzorů[1]: vícestránková aplikace (angl. multiple-page application, dále jen `MPA`) a jednostránková aplikace (angl. single-page applicaton, dále jen `SPA`) [15].

2.1 Více stránkové aplikace

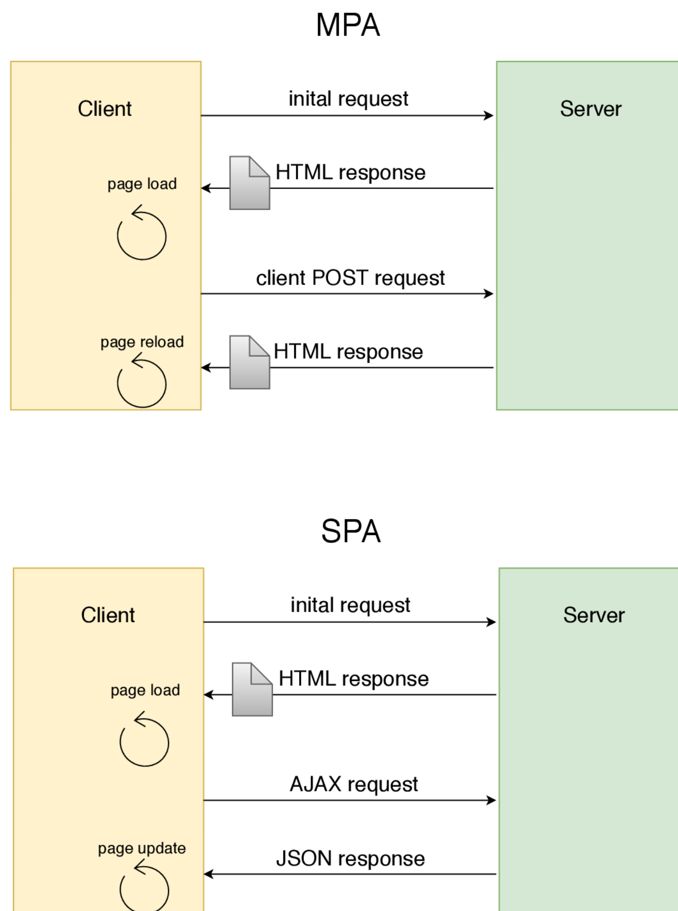
MPA je tradiční cesta pro vývoj webových aplikací, potažmo webových stránek. Zde se většina aplikační logiky provádí na serveru, který je většinou s klientskou stranou úzce spojen. Typicky na většinu dotazů, které klient odešle, server odpoví patřičnou HTML šablonou. Přijetí, respektive zobrazení této šablony probíhá synchronně, tudíž je signalizováno obnovením stránky. Z tohoto důvodu jsou zde větší nároky na síťovou komunikaci. Schéma komunikace je zobrazeno na obrázku 2.1. Tento přístup je vhodné využít tehdy, pokud má aplikace jednoduché, povětšinou statické uživatelské rozhraní. To se týká zejména různých e-shopů, informačních systémů či zpravodajských portálů. Ovšem pokud potřebujeme dynamickou, interaktivní aplikaci, je vhodné použít přístup SPA. Z tohoto důvodu je i nástroj **RepView** implementován jako SPA.

2.2 Jednostránkové aplikace

Jednostránková aplikace využívá opačných principů než MPA. Při tomto přístupu se většina aplikační logiky provádí přímo v prohlížeči na klientské straně. Komunikace s webovým serverem je zajištěna pomocí aplikačního uživatelské rozhraní (angl. application user interface, dále jen API).

Vzhledem k tomu, že u SPA se většina zdrojů (HTML+CSS) načte pouze jednou, je tento přístup rychlý. Po prvním načtení stránky se již její obsah pouze dynamicky překresluje dle obrázku 2.1. Překreslování jednotlivých částí uživatelského rozhraní probíhá asynchronně; uživatel nečeká na načtení či obnovení stránky. Díky tomu je zlepšen uživatelský zážitek. Obsah stránky, tedy data, můžou být nahrána do lokálního úložiště, které slouží jako úložiště dat na klientské straně. Dynamické překreslování stránky s sebou nese drobnou nevýhodu ve formě nalezitelnosti stránky. Stránky s převážně statickým obsahem jsou pro webové prohlížeče snáze nalezitelné oproti těm, v kterých se obsah razantně mění za běhu.

[1] Existují ještě hybridní přístupy, které kombinují vlastnosti obou návrhových vzorů



Obrázek 2.1: Rozdíl komunikace klienta se serverem u MPA vs SPA

Pro vývoj interaktivní SPA je nutné využít jazyk Javascript. Samotný Javascript ovšem pro vývoj moderní, komplexnější webové aplikace nestačí. Je tudíž vhodné použít již existující framework či knihovnu, případně takovou knihovnu vytvořit. Využitím frameworku s sebou nese řadu výhod. Následující seznam zmiňuje nejdůležitější z nich.

- **Rychlost vývoje** - Při použití frameworku programátor nemusí řešit nízkoúrovňové problémy, jakými jsou například validace formulářových dat, které provede framework.
- **Bezpečnost**[2] - Použitím frameworku je méně náchylné na bezpečností chyby, kterých by se jinak programátor mohl dopustit.
- **Podpora** - Kvalitní a používaný framework by měl obsahovat kvalitní dokumentaci, podpořenou o vývojářská fóra a komunitu.

Ideální knihovna či framework neexistuje, vždy je nutné nastudovat její vlastnosti na základě požadavků, které jsou na aplikaci kladeny.

[2]Bezpečnost je zde myšlena z pohledu programátora. Kód frameworku je veřejný, tudíž může být náchylnější k potencionálním útokům.

2.3 Použité technologie

Tato podkapitola shrnuje použité technologie, včetně jejich popisu, srovnání a způsobu výběru.

Javascript

Javascript[3] je dynamický, multiparadigmatický jazyk, který obvykle[4] běží na klientské straně v prohlížeči. Samotný způsob zpracovávání jazyka záleží na konkrétní implementaci `enginu` v prohlížeči. Tento engine se napříč prohlížeči liší [19], nicméně jedná se buď o `interpretaci` či `just-in-time` kompilaci, tedy způsob, kde se kombinuje kompilace a interpretace [1]. Tím je dosaženo urychlení běhu [5].

Javascript běží na jednom vlákně, obsahuje tedy jeden zásobník volání a jednu paměť typu heap. Z tohoto popisu by se mohlo zdát, že nelze vykonávat asynchronní operace. Nicméně je jazyk podporuje, a to kombinací běžného zásobníku volání, který je pod správou javscript enginu, služeb Web APIs[5], event-loopu a callback fronty[2]. Web APIs jsou služby dostupné z prohlížeče, nicméně jejich zpracování nenáleží přímo pod javascript engine.

Zjištění asynchronních služeb javascriptu je fundamentální pro vývoj webových aplikací, zejména těch jednostránkových, kde jiná než asynchronní změna obsahu je nepřípustná.

Vue.js

Vue.js[6](dále již jen Vue) je progresivní javascriptový framework založený na komponentách. Vue využívá virtualního DOMu, pomocí kterého sleduje změny regulárního DOMu, a v případně jeho změny aktualizuje jen ty prvky, které byly změněny [4]. Není tudíž nutné překreslovat celý DOM znovu. Taktéž vyhledávání prvků ve virtuálním DOMu je méně nákladné než přes běžné DOM API[7].

Jak již bylo zmíněno, Vue je založen na komponentách. Každá komponenta musí být obsažena v souboru *.vue. Tento soubor obsahuje vlastní HTML šablonu, dále popis samotné komponenty v javsacriptu a následně stylování. Nyní je možné namítnout, jestli tento styl zápisu splňuje podstatu architektury oddělení zodpovědností (Separation of Concerns). Zde je nutné dodat, že oddělení zodpovědností neznamená rozdělení do souborů dle typu. V moderním vývoji uživatelského rozhraní dává větší smysl třídit kód do menších komponent, kde každá komponenta má vlastní logiku, šablonu a styl, namísto rozdělení celého kódu do tří velkých vrstev. Jednotlivé komponenty se tak stávají znovupoužitelné a jednoduché na údržbu. Jak již bylo avizováno, každá Vue komponenta se skládá ze tří základních částí.

- **template** - Sekce `template` obsahuje běžný HTML kód, kde je navíc možné použít speciální direktivy, díky nimž můžeme docílit větvení či iterace. Dále je možné jednotlivým HTML elementům přiřadit dynamické vlastnosti, například styly či třídy, které se můžou měnit za běhu aplikace. Vue také implementuje vlastní naslouchání událostí. Každému elementu v šabloně lze přiřadit událost, jejíž logika je popsána v sekci `script`. Mezi naslouchané události může patřit stisknutí tlačítka, klávesy či vlastní uživatelská akce.

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript

[4] Javascript je možné využít na serverové straně (běhové prostředí Node.js) či pro vývoj desktopových aplikací (framework Electron).

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

[6] https://vuejs.org/

[7] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

- **script** - Každá komponenta v sekci `script` je zpřístupněna pomocí výrazu `export default{}`. V případě nutnosti využití komponenty je nutné ji naimportovat. Tímto způsobem má programátor přehled, jaké závislosti má daná komponenta. Tuto skutečnost lze dále posunout vytvořením kontejnerů, které zajišťují vkládání závislostí dle vzoru IoC (Inversion of Control[8]). Jednotlivé komponenty lze do sebe libovolně vnořovat.

- **style** - Stylování HTML elementů dané komponenty probíhá v sekci `style`. Krom běžného CSS je možné využít různé preprocesory, jakými jsou Sass[9], Less[10] či Stylus[11].

Každá komponenta má vlastní životní cyklus. K jednotlivým částem životního cyklu lze přistupovat přes specifické metody. To je obzvláště užitečné, pokud je třeba před vykreslením komponenty získat určitá data kupříkladu z API.

Vue implementuje **jednostranné vázání dát** z rodičovské komponenty do synovské. Pokud se data v rodičovské komponentě změní, automaticky se upraví i v patřičné synovské komponentě, nikoliv však naopak. Nicméně každá komponenta může vytvářet a naslouchat na události. Pokud je potřeba zaslat data ze synovské komponenty do rodičovské, je nutné vytvořit událost pomocí metody `$emit`, která je následně zachycena v rodičovské komponentě. Co se týče vázání dat mezi šablonou a daty v komponentě, zde funguje **obousměrné vázání dat**. Každá změna dat v šabloně je reflektována změnou příslušných dat v komponentě, a naopak.

V porovnání s konkurenčními frameworky jako je Angular[12] či React[13] je Vue jednoduchý framework s dobrou učící se křivkou, z kterého modulárnost a snadná rozšiřitelnost dělá ideální volbu pro jednostránkovou aplikace.

Quasar

Quasar[14] je framework sloužící pro snadný návrh a vývoj uživatelského rozhraní. Obsahuje široký soubor parametrizovatelných komponent, které jsou založeny na již zmíněném frameworku Vue.

CodeMirror

Vzhledem k tomu, že cílem práce je vytvořit nástroj sloužící k revizi zdrojových kódů, je nutné vybrat vhodnou knihovnu, která interakci se zdrojovým kódem zprostředkuje. Samotný výběr předchází formální specifikace požadavků, kterým se detailněji věnuje kapitola 3.

CodeMirror[15] je javascriptová knihovna sloužící k zobrazování, případně editaci zdrojového kódu v prohlížeči. Výhodou toho nástroje je možnost ho využít jako Vue komponentu. Samotná knihovna poskytuje rozmanitou základní konfiguraci, přičemž k pokročilejším funkcím lze přistupovat díky kvalitnímu programovatelnému API. Užití této knihovny lze najít kupříkladu v nástrojích pro vývojáře pro prohlížeč Chrome[16].

[8] https://vuejs.org/v2/guide/components-edge-cases.html#Dependency-Injection
[9] https://sass-lang.com/
[10] http://lesscss.org/
[11] https://stylus-lang.com/
[12] https://angular.io/start
[13] https://reactjs.org/
[14] https://quasar.dev/
[15] https://codemirror.net/
[16] https://developers.google.com/web/tools/chrome-devtools

Knihovna funguje na bázi editoru, kde každý zdrojový kód je vložen do jedné instance editoru. Tato instance představuje javascriptový objekt, nad kterým již lze volat metody dle API. Objektu lze nastavit události, na které bude naslouchat. Jejich zpracování je opět provedeno pomocí metod API, není tedy potřeba přímo manipulovat s DOM. Dále CodeMirror nabízí možnost vytvoření vlastních klávesových zkratek, na které je možné připojit příkazy, které zajistí jejich zpracování.

Další uvažované nástroje

Tato sekce se věnuje ostatním nastudovaným nástrojům a modulům sloužícím k zobrazování zdrojových kódů na webových stránkách. Jejich celkové srovnání je znázorněno v tabulce 2.3.

Prism

Prism[17] je malá a rozšiřitelná knihovna sloužící k zvýraznění syntaxe napsaná v javascriptu. Samotná knihovna podporuje definici vlastního jazyka, potažmo zvýrazňovače syntaxe. Umožňuje označení specifického řádku, zvýraznění syntaxe pro vnořené jazyky či zobrazení netisknutelných znaků, jakými jsou tabulátory, znaky konce řádku, apod. Zároveň také disponuje širokou sortou dodatečných modulů či balíků, které je možné v případě zájmu využít. Samotná dokumentace je ovšem velmi minimalistická, bez jakéhokoliv API.

Pygments

Pygments[18] je knihovna napsaná v Pythonu taktéž sloužící k zvýraznění zdrojového kódu, ovšem na serverové straně. Tato vlastnosti z ní činí vhodnou volbu pro webové stránky či fóra, které mají převážně statický obsah. Server zde odešle zvýrazněný kód společně s HTML šablonou, která se následně zobrazí uživateli v prohlížeči. Ovšem pro aplikace, ve kterých se kód, který je určený ke zvýraznění dynamicky mění, tato knihovna není nejlepší volbou. Před každým překreslením stránky za účelem zobrazení jiného zdrojového kódu by musel být server znova dotázán kvůli jeho zvýraznění. Tato metoda je ovšem v **rozporu** s konceptem jednostránkové aplikace.

Google Code Prettify

Google Code Prettify[19] je javascriptový skript sloužící zvýraznění zdrojových kódů. Nástroj lze použít jako CSS třídu. Tato třída se nastaví HTML elementu se značkou `pre`, která obsahuje kód určený ke zvýraznění. Skript ovšem nelze použít jako Vue komponentu, a jeho oficiální repozitář není již spravován.

Ace.c9

Knihovna Ace[20] slouží jako editor zdrojového kódů na webových stránkách napsaný v jazyce javascript. Nástroj nabízí užitečné funkce pro editování či psaní programového kódu. Mezi nejdůležitější patří automatické odsazování, registrace vlastních klávesových zkratek, vyhledávání podle regulárních výrazů, vyznačení shodujících se závorek, zobrazení netisknutelných znaků, skrývání a odkrývání řádků, možnost více kursorů či živá kontrola syntaxe

[17] https://prismjs.com/
[18] https://pygments.org/
[19] https://github.com/googlearchive/code-prettify
[20] https://ace.c9.io/

společně s napovídáním. Ace je knihovna s přehlednou dokumentací, avšak její primární účel je sloužit jako živý editor kódu na webových stránkách.

HighlightJs

HighlightJS[21] je dalším z modulů sloužícím pro zvýraznění syntaxe na webu. Výhoda této knihovny spočívá zejména ve snaze o automatickou detekci jazyka. Dále je kompatibilní s většinou javascrpitových frameworků, včetně Vue. Lze je tedy využít jako Vue komponentu. Knihovna ovšem v základní verzi, tj. bez dodatečných pluginů či modulů nepodporuje číslování řádků.

název	**Prism**	**Pygments**	**Google Code Prettify**	**Ace.c9**	**Code Mirror**	**Highlight js**
udržovaný	ano	ano	ne	ano	ano	ano
jazyk	javascript	python	javascript	javascript	javascript	javascript
číslování řádků	ano	ano	ano	ano	ano	ne
automatické rozpoznání jazyka	ne	ano	ne	ne	ne	ano
zvýraznění specifického řádku	ano	ano	ano	ano	ano	ano
skrývání řádků	neoficiální plugin	ne	ne	ano	ano	ne
přepínání tabů	ne	ne	ne	ano	ne	ne
online editace kódu	ano	ne	ne	ano	ano	ne

Tabulka 2.1: Přehled nástrojů sloužících k zobrazování zdrojových kódů na webu

Z výše uvedené tabulky lze odvodit, že žádný nástroj stoprocentně nevyhovuje všem požadavkům, které jsou na něj kladeny. Nicméně s přihlédnutím na jednotlivé váhy požadavků (automatické rozpoznání jazyka je například méně důležité než udržovanost knihovny) společně s nastudováním jednotlivých modulů byl vybrán již zmíněný nástroj `CodeMirror`.

Nginx

Nginx[22] je populární webový server. Samotná aplikace ovšem nemusí sloužit čistě jako HTTP server, nicméně může poskytovat mailové proxy, HTTP kešování, reversní proxy, vyvažování zátěže či komprimace odpovědí.

Python

Python[23] je vysokoúrovňový, dynamický, multiparadigmatický skriptovací jazyk. Pro python existující různé implementace, nicméně standardní implementace je v jazyce C, taktéž nazývána CPython. Tato implementace zdrojový kód nejdříve zkompiluje do mezikódu, někdy označeném jako bajtkód. Tento mezikód je následně interpretován nad virtuálním strojem [13].

[21] https://highlightjs.org/
[22] https://www.nginx.com/
[23] https://www.python.org/

Ačkoliv existuje spousta jazyků, které jsou podstatně rychlejší, Python disponuje velkým množstvím knihoven či modulů, které z něj činí velmi variabilní jazyk. Python disponuje uživatelsky velmi přívětivou syntaxí, která minimálně pro začínající programátory může vést k rychlejšímu pokroku. Kombinace již zmíněný vlastností z něj dělá jazyk vhodný nejen pro prototypování, ale i k plnohodnotnému vývoji.

Flask

Flask[24] je webový mikro framework pro Python. V základu Flask nenabízí žádnou databázovou abstraktní vrstvu, ověřování formulářových dat či striktní rozdělení MVC architektury do patřičných složek, jako tomu je například u frameworku Django[25]. Jeho zprovoznění či konfigurace je také podstatně jednoduší. Díky těmto vlastnostem je Flask ideální volbou pro tvorbu API.

K jeho tvorbě bylo využito rozšíření Flask-RESTful[26], které poskytuje další podporu pro rychlý, efektivní a přehledný vývoj API.

uWSGI

uWSGI[27] je webový aplikační server, který představuje rozhraní mezi webovým serverem a webovou aplikací. V případě této práce se jedná o rozhraní mezi frameworkem Flask a webovým serverem Nginx. S webovým serverem může uWSGI komunikovat buď pomocí protokolu HTTP[28], nebo binární protokolu uwsgi[29] za předpokladu, že to webový server podporuje. Konfigurace uWSGI se nachází v souboru uwsgi.ini. Zde je mimo jiné možné nakonfigurovat počet procesů či vláken, na kterých bude Flask pracovat, počet maximálních dotazů, které obslouží jeden proces než se restartuje či možnost restartu procesu po alokování určitě paměti za účelem zabránění případného úniku paměti.

Docker

Docker[30] je projekt s otevřeným zdrojovým kódem sloužící k automatizaci nasazení aplikací ve formě samostatných, přenositelných virtuálních kontejnerů. Tyto kontejnery lze spustit v cloudu či lokálním prostředí. Kontejnerizace je velmi vhodná pro architekturu mikroslužeb [8]. Díky kontejnerizaci lze vyřešit problémy se škálovatelností, režijními náklady či portabilitou software. Oproti klasické, plné virtualizaci se při kontejnerizaci virtualizuje jádro operačního systému. Všechny kontejnery bežící v rámci jednoho operačního systému mají sdílenou paměť, knihovny a zdroje. Při plné virtualizaci se na hostitelský operační systém nainstaluje hypervisor software [16], který má na starost vytváření virtuálních strojů. Každý tento virtuální stroj se chová jako samostatný operační systém. Docker se skládá ze dvou základních konceptů, obrazu a kontejneru.

[24] https://flask.palletsprojects.com/en/1.1.x/
[25] https://docs.djangoproject.com/en/3.0/
[26] https://flask-restful.readthedocs.io/en/latest/
[27] https://uwsgi-docs.readthedocs.io/en/latest/
[28] https://devdocs.io/http/
[29] https://uwsgi-docs.readthedocs.io/en/latest/Protocol.html
[30] https://www.docker.com/

Obraz

Obraz, neboli Docker image je samostatná část software vytvořená dle předpisu v souboru `Dockerfile`. Tento obraz je v podstatě šablona k vytváření kontejnerů. Na základě toho obrazu může Docker démon spustit patřičný kontejner.

Kontejner

Kontejner je jedna konkrétní instance aplikace vytvořená dle korespondujícího obrazu. Tento kontejner běží jako samostatný proces, který má definované vlastnosti jako interní/externí porty či připojené diskové svazky [16]. Kontejnery můžou komunikovat buď mezi sebou v rámci dockerové sítě, nebo i s hostitelským systémem.

Docker Compose

V reálném světě ovšem málokdy využíváme pouze jeden obraz, resp. kontejner. Pokud je potřeba do jedné aplikace zakomponovat více kontejnerů, slouží k tomu služba `Docker Compose`. Pro náročnější orchestraci kontejnerů existují služby `Docker Swarm` či mocnější `Kubernetes` [11].

Docker Compose je nástroj sloužící k definici a spuštění více kontejnerů na jednom stroji. Podobně jako se obraz sestavuje z konfiguračního souboru jménem `Dockerfile`, tak zde je konfigurace zapsána v souboru `docker-compose.yml`. Pomocí tohoto předpisu lze předat proměnné běhového prostředí, specifikovat konkrétní soubory Dockerfile, dle kterého se kontejnery sestaví nebo specifikovat vzájemné závislosti mezi jednotlivými kontejnery. Na schématu 2.1 je vidět ukázkový konfigurační předpis, který zároveň slouží jako předpis pro produkční verzi aplikace.

```
version: "3.7"

services:
    # Backend container definition
    backend:
        container_name: backend
        build:
            context: ./backend/
            target: 'prod-stage'
            dockerfile: Dockerfile-prod
        ports:
            - ${BACKEND_PORT}
        volumes:
            # Attach docker volume to backend file system
            - data-volume:/backend/uploaded_files
            - ./backend:/backend
        environment:
            # Set environment variables
            - VERSION=${VERSION}
            - PORT=${PORT}
            - PROTOCOL=${PROTOCOL}
        privileged: true
      # Frontend container definition
    frontend:
        container_name: frontend
        build:
            context: ./frontend/
            dockerfile: Dockerfile-prod
            target: 'production-stage'
            args:
                # Load environment variables during build stage
                - VUE_APP_BACKEND_API_URL=${VUE_APP_BACKEND_API_URL}
        ports:
            - ${FRONTEND_PORT}:80
        volumes:
            - ./frontend:/frontend
        links:
         # Containers are reachable in docker network
          - backend
        restart: always
        privileged: true
# Definition of persistent volume named data-volume
volumes:
     data-volume:
```

Zdrojový kód 2.1: Ukázkový předpis souboru docker-compose.yml

Jednotlivých předpisů pro službu Docker Compose lze mít více, kupříkladu jeden pro vývojovou verzi aplikace a jeden pro produkční nasazení.

Kapitola 3

Analýza a specifikace požadavků

Před vývojem každého produktu je nejdříve nutné provést analýzu a specifikaci požadavků. Tato část je první etapou životního cyklu software a tvoří přibližně 8%-9% [9] času celkového vývoje produktu. Cílem této etapy je převést všechny neformální požadavky a nároky na aplikaci do strukturovaného popisu formálních požadavků.

3.1 Specifikace využití aplikace

Vzhledem k tomu, že cílem této práce je umožnit uživateli jednoduše interaktivně prohlížet zdrojové kódy, je nejdříve nutné specifikovat, co vše bude potencionální uživatel chtít provádět za akce. Diagram případů užití je zobrazen na obrázku 3.1.

Diagram případů užití

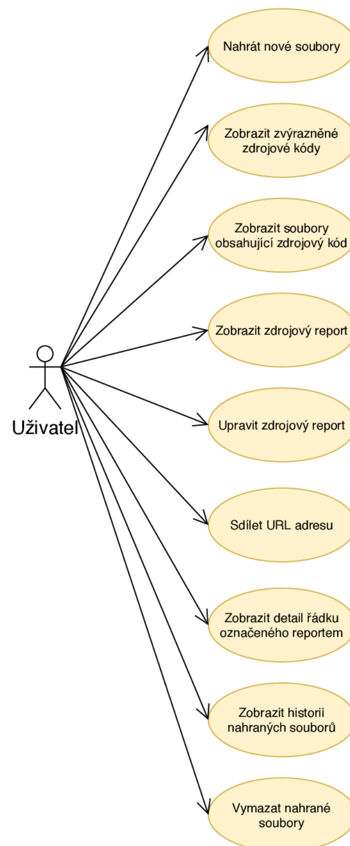
Diagram případů užití je v softwarovém inženýrství jeden z diagramů sloužící k popisu chování systému z pohledu uživatele. Diagram případů užití patří do UML, konkrétně do kategorie diagramů popisujících chování. Jedním z cílů tohoto diagramu je analyzovat potencionální užití systému uživatelem a nalézt hranice, které by měl systém splňovat co se týče funkcionality. Pro správnou interpretaci diagramu případů užití je nutné specifikovat jeho prvky.

- **účastník** - subjekt, který se systémem pracuje
- **případ užití** - funkce, kterou systém na základě akcí uživatele vykonává
- **interakce** - znázorňuje účast subjektu na provádění případu užití

Vztah mezi subjektem a případem užití je znázorněn plnou čarou mezi nimi. Daná čára zakončená šipkou indikuje zainteresovanost subjektu v daném případu užití. Pokud čára propojuje dva subjekty či případy užití, lze mluvit o zobecnění. Dále zde existují relace *include* a *extend*, nicméně jejich vhodné užití musí být velmi opodstatněné, protože často vedou spíše ke zmatení než ke zjednodušení diagramu.

Existují dvě skupiny požadavků - funkční požadavky a nefunkční.

- **Funkční požadavky** - popisují chování systému. Do této kategorie patří již zmíněný diagram případů užití.
- **Nefunkční požadavky** - popisují vlastnosti, charakteristiky a omezení systému, které musí být respektovány.



Obrázek 3.1: Základní diagram případů užití popisující očekávané akce uživatele

S ohledem na výše zobrazené případy užití je možné přistoupit k specifikaci jednotlivých požadavků, které jsou na aplikaci kladeny.

3.2 Požadavky na prohlížení zdrojových kódů

Následující seznam specifikuje požadavky na uživatelské rozhraní, jehož hlavní účelem je zobrazování a prohlížení zdrojových kódů na základě reportu.

- Zvýraznění syntaxe
- Jasné označení řádku, ke kterému se report vztahuje
- Zobrazení více souborů se zdrojovými kódy
- Zobrazení zdrojového reportu
- Přehledné zobrazení historie
- Třídění nahraných souborů do složek
- Zobrazení nahraných souborů ve složce
- Možnost přejmenování složky s nahranými soubory

- Možnost zobrazení libovolných nahraných souborů z uživatelské historie
- Možnost editace nahraného reportu
- Zobrazení dodatečných informací o chybovém řádku
- Možnost sdílení nahraných souborů
- Možnost přechodu z jednoho reportu do následující / předchozího, pokud mezi nimi existuje návaznost

3.3 Požadavky na server

Níže uvedený seznam specifikuje požadavky na backendovou část aplikace, jejíž úkolem je zpracování nahraných souborů.

- Server komunikuje s ostatními služby přes protokol HTTP
- Server implementuje REST rozhraní
- Server implementuje otevřené API
- Server je nezávislý na uživatelském rozhraní
- Na server je možné přistoupit jinak než přes uživatelské rozhraní
- Datové úložiště pro nahrané soubory je souborový systém
- Datové úložiště je persistentní
- Server funguje v konkurentním režimu
- V případě chyby server odpoví patřičným HTTP chybovým kódem
- Server zpracuje nahraný report do unifikovaného formátu
- Zpracovaný report v unifikovaném formátu bude uložen ve formátu JSON
- Server bude jednoduše rozšiřitelný

3.4 Požadavky na aplikaci

Následující seznam se věnuje specifikaci požadavků na aplikaci jako celek.

- Aplikace je spustitelná pomocí služby Docker Compose
- Jednotlivé služby aplikace běží uvnitř kontejnerů
- Jednotlivé služby aplikace jsou vzájemně nezávislé
- Aplikace je přístupná jako webová aplikace prostřednictvím protokolu HTTP na zadaném portu

Na základe výše uvedených požadavků je možné již přistoupit k návrhu aplikace, kterému se věnuje kapitola 4. Návrh jednotlivých služeb, potažmo celé aplikace musí reflektovat zmíněné požadavky, které jsou na ni kladeny.

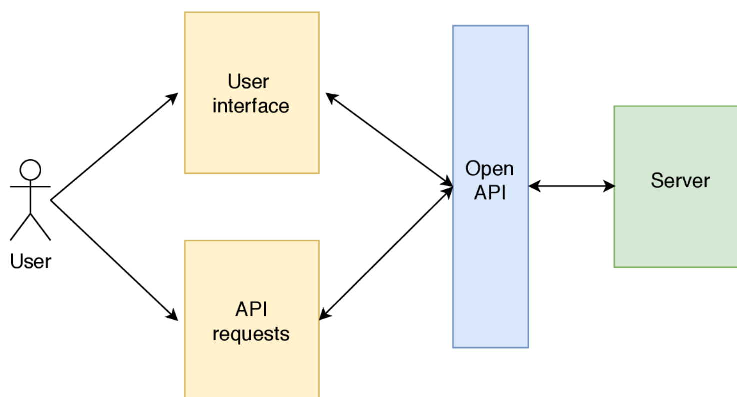
Kapitola 4

Návrh aplikace

V této kapitole je popsán celkový návrh aplikace. Bude popsána komunikace v rámci aplikace na konceptuální úrovni, dále kompozice služeb uvnitř Dockeru a v neposlední řadě návrh jednotlivých služeb uvnitř kontejneru, tedy frontendu a backendu.

4.1 Konceptuální návrh

Primární využití aplikace je přes grafické uživatelské rozhraní, čili webový prohlížeč. Zde je nutné vycházet ze specifikace požadavků sepsaných v kapitole 3, a ustanovit si formát a způsob komunikace se serverem, ke kterém slouží `REST API`, které je dále popsáno v kapitole 4.3. Ovšem z hlediska možného budoucího rozvoje a využití aplikace, mimo testování, je velice vhodné, aby zmíněné API bylo dostupné jinak přes grafické uživatelské rozhraní aplikace. Z tohoto důvodu je API implementováno jako otevřené, dle obrázku 4.1.



Obrázek 4.1: Konceptuální návrh aplikace

Tento přístup zajistí možnost potencionálního využití služeb backendu jinak než přes grafické uživatelské rozhraní. Přítomnost API dále usnadňuje možnost rozšíření serverových služeb.

Skladba Docker kontejnerů

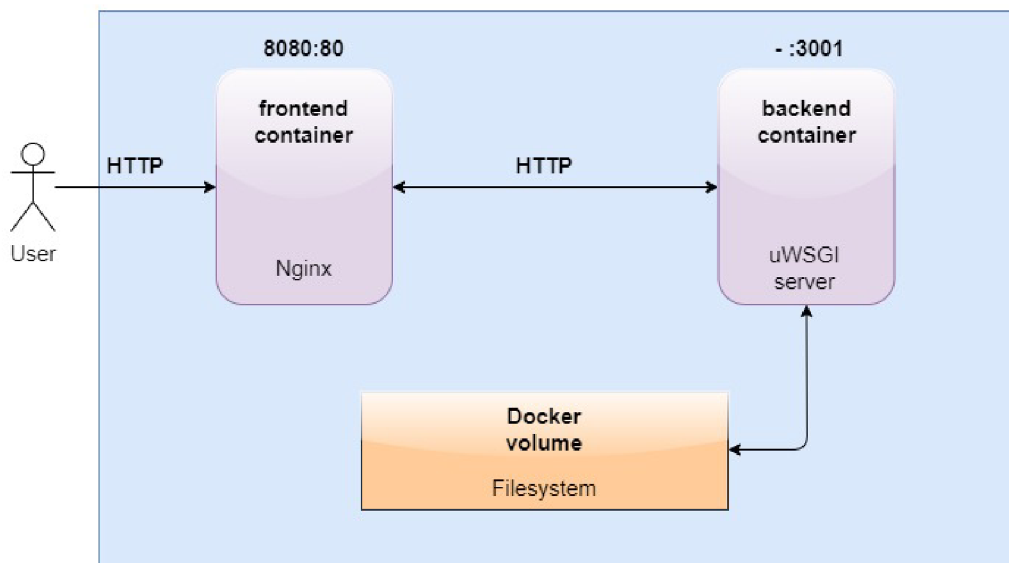
Jak již bylo zmíněno v kapitole 2.3, aplikace využívá služeb Dockerové kontejnerizace. Služby **frontend** a **backend** tedy běží v separátních kontejnerech. Ve frontendovém kon-

tejneru běží služba Nginx, která se stará o servírování statických souborů, tedy ostylované HTML šablony společně s javascriptovým kódem. Dále slouží k reversní proxy na backend. Není tudíž nutné znát port, na kterém běží backend. Veškeré požadavky se posílají na frontend. V případě, že se jedna o požadavek na server, Nginx ho pomocí reversní proxy přesměruje na backendový kontejner. Tvar jednotlivých typů požadavků je popsán v kapitole 5.1. Backend má navíc přístup k docker volume, který slouží jako úložiště pro nahrané soubory.

Docker volume

Docker obrazy jsou založeny na sériích vrstev, které jsou označeny jako `read-only`, tedy pouze pro čtení. Pokud je na základě obrazu spuštěn kontejner, je na vršek přidána `read-write` vrstva, která již umožňuje zápis. Pokud proběhne modifikace či zápis nějakých dat, v `read-write` vrstvě se pouze vytvoří jejich kopie na základě `read-only` vrstvy. Patřičná `read-only` vrstva tedy obsahuje původní soubory, a `read-write` vrstva obsahuje nové či modifikované soubory. Pokud je kontejner smazán či znovu vytvořen, obsahuje data pouze z `read-only` vrstvy. Předchozí modifikace nebudou v novém kontejneru promítnuty.

Při potřebě vytvoření perzistentního úložiště Docker poskytuje koncept svazků [1]. Tyto svazky, zvané `Docker volume` poskytují perzistentní datové úložiště, data uložená na tomto svazku jsou tedy dostupná i po vypnutí či restartu aplikce, resp. kontejneru. Samotné svazky jsou uloženy mimo souborový systém Dockeru (Union File System) [12]. Data ve svazcích jsou uložena v hostitelském souborovém systému.



Obrázek 4.2: Skladba služeb uvnitř Dockeru

Z výše uvedeného obrázku je vidět, že kontejnery backend a frontend spolu komunikují jako oddělené služby. Tato architektura se nazývá **mikroslužby** a je blíže popsána v následující kapitole.

[1] https://docs.docker.com/storage/volumes/

Monolitická architektura

Monolitická architektura je tradiční způsob návrhu webových aplikací, potažmo stránek. Aplikace je zde složena z jedné, nedělitelné jednotky. Tato velká jednotka v sobě obsahuje uživatelské rozhraní, business logiku, aplikační logiku, databázi apod. Už z tohoto popisu je zřejmé jasné, že jakákoliv úprava či rozšíření jsou poměrně nákladné a postihují celou aplikaci. Přechod na jiné technologie či verze knihoven zde může být dosti problematický. Dále je monolitická architektura nákladnější na údržbu a pochopení. Pokud do vývojářského týmu přijde nový člen, zpravidla mu trvá delší dobu pochopit princip fungování celé aplikace. Oproti tomu je tento způsob návrhu aplikace jednoduší na vývoj, nasazení či testování.

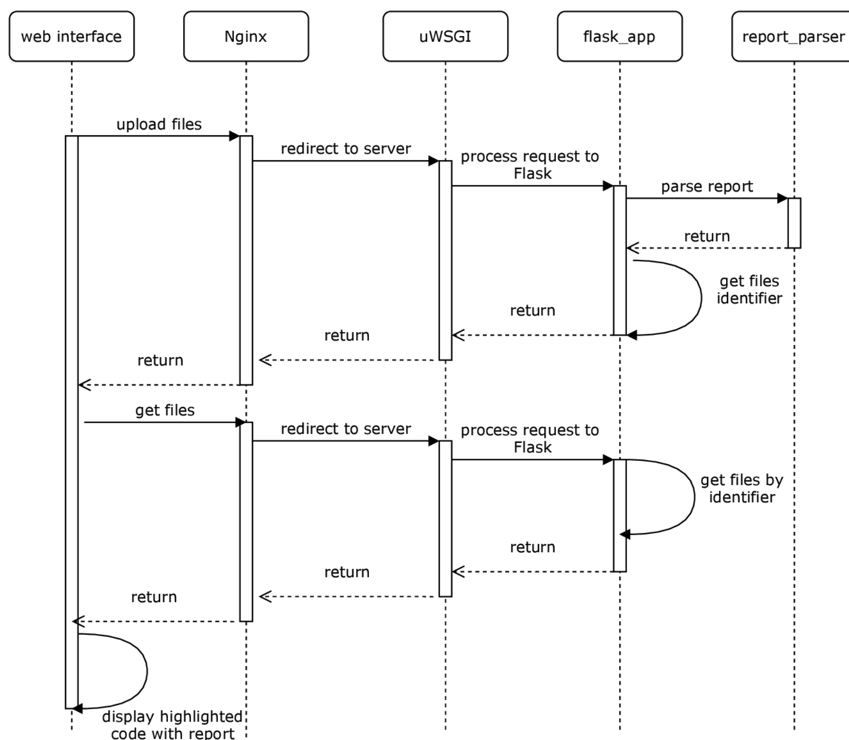
Architektura mikroslužeb

Princip mikroslužeb spočívá v rozdělení jednotlivých částí aplikace na menší, samostatně nezávislé jednotky. Každá jednotka má vlastní logiku a komunikují spolu pomocí daného rozhraní, například pomocí protokolu HTTP. Nezávislost jednotlivých služeb poskytuje při vývoji či nasazení mnoho výhod. Každý služba může být vyvíjena nezávisle na celém systému. Stejně tak chyba, která se projeví v jedné službě, nepostihne zbytek aplikace, jak by tomu bylo v monolitické architektuře. Při přidání nové služby či její migraci na jinou technologii nebo verzi opět není nutné zasahovat do implementace zbytku aplikace. V kontrastu těchto výhod, mikroslužby jsou mnohem komplexnější na návrh a testování.

Komunikace služeb

Jak je znázorněno v obrázku 4.2, služby frontend a backend mezi sebou komunikují pomocí protokolu HTTP. Tímto protokolem jsou přenášena data ve formátu JSON[2] (Javascript Object Notation). Následující sekvenční diagram na obrázku 4.3 zobrazuje průběh komunikace jednotlivých částí služeb při nahrání nových souborů. Části diagramu web interface a Nginx patří do kontejneru **frontend**, části uWSGI, flask_app a report_parser do **backendového** kontejneru.

[2] https://tools.ietf.org/html/rfc7159



Obrázek 4.3: Sekvenční diagram nahrání nových souborů

Bližší popis jednotlivých sekvencí diagramu:

1. Uživatel přes webové rozhraní nahraje patřičné soubory, tedy zdrojové kódy a report. Zde se pomocí javascriptu odešle asynchronní požadavek na server dle jeho API.

2. Tento požadavek zachytí služba Nginx, která rozpozná, že se jedná o požadavek na server, a přesměruje ho do backendového kontejneru na příslušný port.

3. Na backendu zachytí požadavek služba uWSGI, která požadavek přiřadí libovolnému neobsazenému procesu, kde běží Flask. S tímto procesem je drženo spojení pomocí *socket API*[3].

4. Flask požadavek dle URI přesměruje na patřičný koncový bod, a spustí vykonání aplikační logiky na základě HTTP metody. Ta zahrnuje zpracování reportu do unifikovaného formátu a následně uložení nahraných souborů a zpracovaného reportu do nově vytvořené složky v souborovém systému.

5. V odpovědi na dotaz Flask vrátí jméno složky, která slouží jako identifikátor nahraných souborů. Jméno složky se generuje dle aktuálního času.

6. Jakmile webové rozhraní, resp. javascript dostane kladnou odpověď na zaslaný dotaz, vytvoří se nový asynchronní dotaz za účelem získání nahraných souborů. V tomto dotazu se využije název složky získaný v předchozím dotazu.

7. Proces zpracování požadavku již proběhne stejně.

[3]http://web.mit.edu/macdev/Development/MITSupportLib/SocketsLib/Documentation/sockets.html

8. Flask dle URI zjistí, který koncový bod obslouží požadavek. Ten v aplikační logice načte nahrané soubory a vrátí je ve formátu JSON.

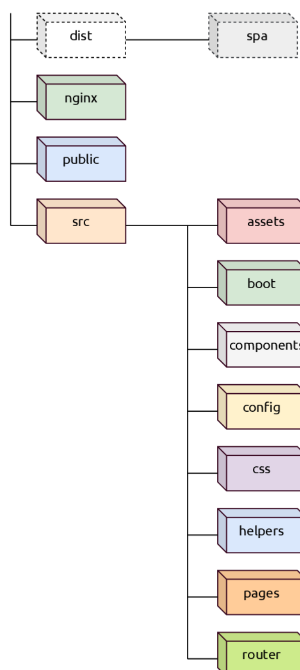
9. Javascript přijme odpověď a následně uživateli patřičně vykreslí syntakticky zvýrazněné zdrojové kódy společně se zvýrazněnými řádky dle reportu, se kterými je možné dále interagovat.

4.2 Návrh frontendu

Hlavním smyslem kontejneru frontend je poskytnou uživateli přívětivé webové rozhraní, které bude pokud možno snadno rozšiřitelné. Aby tento cíl mohl být splněn, je nutné vytvořit určitou projektovou strukturu na základě použitého frameworku Vue (viz 2.3). Tato kapitola se zabývá návrhem aplikační logiky frontendu, grafickému uživatelskému rozhraní se věnuje kapitola 6. Základní vývojová struktura služby frontend je znázorněna na obrázku 4.4. V produkční fázi je zde navíc přítomná složka *dist*, která obsahuje zkompilovanou a optimalizovanou aplikaci nástrojem `Webpack`.

Vývojová verze aplikace je přístupná přes jednoduchý zabudovaný server frameworku Quasar. Díky tomu je možné aplikaci jednoduše ladit například přes webový nástroj Chrome DevTools[4]. Vývojová verze tedy nepoužívá službu Nginx (viz 2.3).

Produkční verze aplikace obsahuje službu Nginx, která uživateli servíruje statická data. Těmi je myšlena hlavní HTML šablona, společně s javascriptovým kódem. Tento kód vznikne zabalením celé aplikace do optimalizovaných funkcí, které jsou vloženy do HTML šablony. Více o produkčním nasazení je uvedeno v kapitole 5.1. Dále má Nginx na starost přesměrovávání požadavků určených na server do backendového kontejneru díky reversní proxy.



Obrázek 4.4: Struktura souborového systému kontejneru frontend

[4]https://developers.google.com/web/tools/chrome-devtools

Následující výčet popisuje význam jednotlivých adresářů.

- **frontend** - kořenový adresář, obsahuje níže uvedené složky a konfigurační soubory frameworku Vue, Dockerfile a soubory obsahující závislosti projektu

---

- **dist** - Zkompilované zdrojové kódy do javascriptové funkce, která je následně vložena do HTML šablony. Pouze v produkční verzi aplikace. Pokud by aplikace byla nasazena v různých módech, tato složka bude obsahovat všechny její distribuce. RepView podporuje mód SPA.
- **nginx** - Konfigurační soubor webového serveru Nginx.
- **public** - Veřejně dostupné soubory.
- **src** - Níže uvedené složky včetně základní HTML šablony a základní komponenty, která tvoří kořenovou komponentu. Tato komponenta je následně dynamicky vložena do již zmíněné HTML šablony.

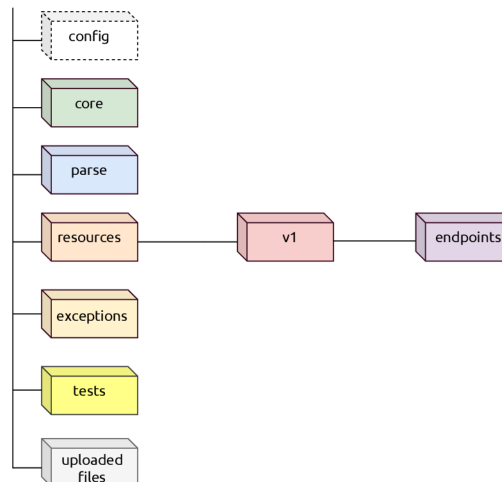
- **assets** - Dodatečné obrázky projektu.
- **boot** - Inicializace knihoven či stylů pro jejich následné využití.
- **components** - Všechny Vue komponenty.
- **config** - Konfigurační soubor, který slouží k načtení proměnných běhového prostředí ze služby `docker compose`
- **css** - Definice globálních stylů.
- **helpers** - Pomocné objekty usnadňující práci s lokálním úložištěm či zpracováním reportu.
- **pages** - Stránky, které jsou vykresleny dle URI adresy.
- **router** - Vue router, který obsahuje definici validních URI adres pro směrování v rámci frontendu.

4.3 Návrh backendu

Účelem serverové části aplikace je umožnit uložení a správu nahraných souborů od uživatele a zpracování zdrojového reportu do unifikovaného formátu. Přístup k těmto souborům je dále možný skrze REST API 4.3. Opět, podobně jako služba frontend, tak i backend je rozdělen na vývojovou a produkční verzi .

Vývojová verze využívá vestavěného webového serveru, kterým disponuje framework Flask. Tento server je primárně určen pouze pro vývojové účely.

Produkční verze disponuje navíc službou uWSGI, která představuje webové aplikační rozhraní mezi Flaskem a službou Nginx běžící ve frontendovém kontejneru. Díky tomuto rozhraní je možné spustit server ve více procesech či vláknech, což ve vývojové verzi serveru není možné. Lze tedy docílit rychlejší odpovědi ze stranu serveru v případě využívání aplikace více uživateli zároveň.



Obrázek 4.5: Struktura souborového systému kontejneru backend

Následující výčet popisuje jednotlivé složky a moduly, včetně jejich účelu.

- **Backend** - Kořenový adresář, obsahuje níže uvedené složky, konfigurační soubory a skripty, závislosti a Dockerfile.

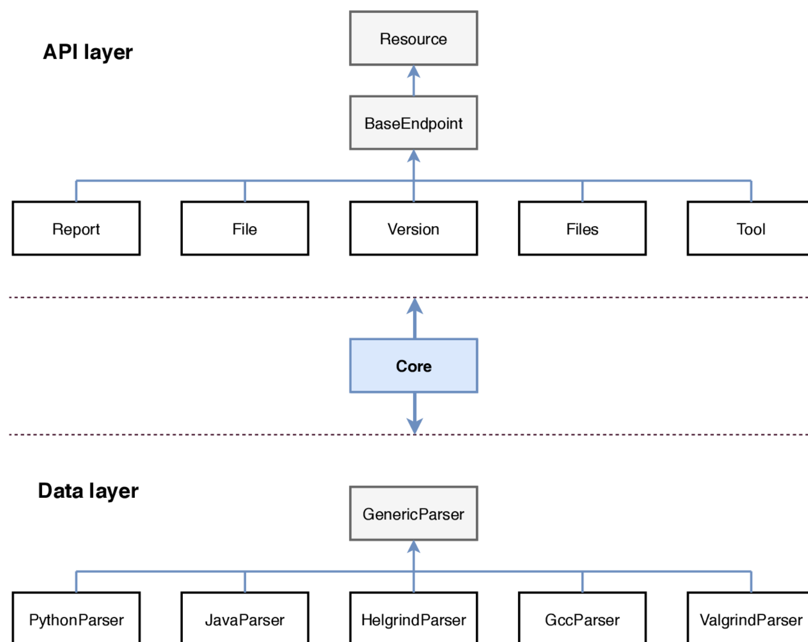
- **config** - Konfigurační soubor služby uWSGI. Pouze v produkční verzi.
- **core** - Hlavní modul pro serverovou činnost.
- **parse** - Jednotlivé parsery reportů.
- **exceptions** - Vlastní typy vyjímek.
- **tests** - Unit testy.
- **uploaded_files** - Složka obsahující nahrané zdrojové soubory.
- **resources** - Definice API.

- **v1** - Verze API.

- **endpoints** - Všechny aktuálně podporované koncové body.

Architektura backendu

Návrh serverové části aplikace se skládá ze dvou hlavních vrstev, které mezi sebou nezávisle komunikují. API vrstva definuje koncové body a povolené přístupové metody ke zdrojům. Datová vrstva obstarává samotnou práci se zdroji, jakožto uložení souborů na disku, jejich modifikaci či zpracování reportu. Nicméně je potřeba tyto dvě vrstvy patřičně propojit, aby mezi sebou mohli komunikovat a předávat si data. K tomu je určena třída `Core`. Tato třída slouží mimo jiné k provádění registrace parserů, výběru konkrétního parseru a jeho dynamickou instanciaci. Na vytvořené instanci třídy se následně invokují metody pro zpracování reportu.



Obrázek 4.6: Logický návrh backendu

Díky dynamickému vytváření objektů a rozhraní, které poskytuje třída `GenericParser`, je docíleno modulárního návrhu. Výše uvedený obrázek 4.6 zobrazuje pouze nejdůležitější třídy, které se v serverové části projektu nachází.

REST API

REST (Representational State Transfer) je architektura rozhraní pro jednotný přístup ke zdrojům přes protokol HTTP. Design REST je definován jako bezstavový a datově orientovaný [3]. Jednotlivé požadavky jsou tedy mezi sebou nezávislé a neukládají si svůj stav na server. Pro případné ověření kontextu uživatele je nutné použít autentizační mechanismus, např. `JSON Web Token`[5] či `OAuth`[6]. Samotný přístup k datům je zajištěn přes URI pomocí čtyř základních HTTP metod, díky kterým je možné provádět CRUD (Create, Retrieve, Update, Delete) operace.

- **GET** - Slouží pro získání dat.
- **POST** - Slouží k odeslání dat.
- **PUT** - Slouží k modifikaci existujících dat.
- **DELETE** - Slouží k mazání dat.

Přenášený formát dat není nijak striktně specifikován, nicméně nejpoužívanějším typem je formát JSON[7]. V případě přenosu jiné než serializované strukturované hodnoty je třeba

[5] https://tools.ietf.org/html/rfc7519
[6] https://tools.ietf.org/html/rfc6749
[7] https://tools.ietf.org/html/rfc8259

specifikovat `Content-Type`[8] v hlavičce dotazu. Design REST by dále měl splňovat šest níže uvedených omezení.

- **Klient-Server** - Architektura klient-server zajišťuje oddělení závislostí serverové a klientské strany, které lze tedy vyvíjet a škálovat nezávisle.
- **Bezstavovost** - Jak již bylo zmíněno v úvodu, REST API je bezstavové, jednotlivé požadavky jsou mezi sebou nezávislé a každý požadavek obsahuje vše potřebné pro jeho zpracování. Případný stav je uchováván na straně klienta.
- **Cache** - Díky bezstavovosti bohužel roste množství požadavků na server, které server musí zpracovat. Architektura REST by měla být schopna indikovat, která data je možná kešovat a která nikoliv.
- **Jednotné rozhraní** - Samotný základ architektury klient server. Pokud je třeba vyvíjet obě strany nezávisle na sobě, musí být jasně definováno rozhraní a způsob, jak budou obě části aplikace mezi sebou komunikovat.
- **Vrstevnatost** - Jak již z názvu plyne, princip vrstevnatosti spočívá v rozdělení aplikace do vrstev, kde každá vrstva má vlastní účel a funkcionalitu. Tyto vrstvy umožňují tvořit hierarchii aplikace.
- **Code on Demand** - Poslední, již spíše volitelná podmínka REST designu spočívá v rozšíření funkcionalit serverové části aplikace na základě přijatého kódu v požadavku z klientské strany. Díky této vlastnosti aplikace není výhradně závislá na vlastním kódu.

Zmíněný výčet vlastností tvoří základní aplikační programové rozhraní, které je široce využíváno.

Report

Jednou z dvou zpracovávaných částí uživatelem nahraných souborů je krom zdrojových kódů také report. Report zde představuje výstup nástroje **analýzy** zdrojových kódů či **traceback**. Nástrojem analýzy zdrojových kódu se rozumí výstup z nástrojů GCC, Helgrind či Valgrind. Traceback představuje výsledek běhové chyby jazyků Python nebo Java[9]. Tato chyba typicky způsobí vyhození výjimky. Pokud daná výjimka není programátorem zachycena, je na výstup programu vypsán právě onen traceback. Samotný traceback obsahuje výpis volání funkcí či metod v zdrojovém kódu v takovém pořadí, v jakém byly invokovány. Pro správnou interpretaci tracebacku je důležité pořadí. Vrchol tracebacku představuje nejvíce abstraktní funkci, tedy tu, která byla zavolána jako první. Dno tracebacku naopak představuje konkrétní funkci, ve které nastala chyba.

Každý report se skládá z **0 - n** částí, přičemž každá část reportu se vztahuje na konkrétní segment zvaný dílčí část kódu. V případě tracebacku se část reportu vztahuje na jednu konkrétní část tracebacku.

[8] https://www.w3.org/Protocols/rfc1341/4_Content-Type.html

[9] Traceback se vyskytuje samozřejmě i v jiných jazycích, nicméně zde jsou zmíněny jazyky, jejichž výstupní report je aktuálně podporován.

Dílčí část kódu

Každý report či jeho část se vždy vztahují ke konkrétnímu úseku zdrojového kódu. Tento úsek se nazývá dílčí část kódu. Dílčích částí kódu může být více, konkrétně přesně tolik, kolik report obsahuje částí.

Jednotný formát reportu

Zvýraznění specifických řádků či zobrazení jejich detailních informací v uživatelském rozhraní se provádí na základě informací obsažených v reportu. Aby však bylo možné s daným reportem jednoduše pracovat, je nutné zajistit jeho zpracování do unifikovaného formátu. Tento formát musí být navržen tak, aby mohl obsáhnout různé formáty zdrojového reportu. Je tedy nezbytné provést analýzu výstupních formátů reportů analýzy zdrojových kódů, protože reporty se mezi sebou liší. Výstup z nástroje GCC je kupříkladu poměrně odlišný od tracebacku jazyka Python. Je tedy nutné se zaměřit na části, které jsou obsaženy v každém zdrojovém reportu, a ty následně zpracovat do předem dané struktury. Tyto části unifikovaného reportu se dále nazývají artefakty. Samotný formát unifikovaného reportu má stromovou strukturu, určité artefakty lze dále vnořovat.

- **File** - Soubor, na který se report vztahuje.
- **Line** - Řádek, na který se report vztahuje.
- **From** - Pozice začátku reportu na daném řádku.
- **Meaning** - Význam daného reportu. Může být funkce, proměnná či třída.
- **Message** - Zpráva, která se vztahuje k danému reportu.
- **Thread** - Vlákno, ke kterému se report vztahuje.
- **Backtrace** - Traceback reportu. Pokud je přítomen, obsahuje libovolný počet zde uvedených artefaktů.

Kapitola 5

Implementace prohlížeče reportů

Následující kapitola se zaměřuje na popis implementačních částí práce. Podobně jako v kapitole 4, i zde je popsána zvlášť implementace funkcionality backendového a frontendového kontejneru.

5.1 Frontend

Cílem frontend kontejneru je poskytnou uživateli možnost si interaktivně procházet zdrojovými kódy na webové stránce dle reportu. K tomu slouží framework Vue v kombinaci s frameworkem Quasar a knihovnou CodeMirror. Další úlohou frontendového kontejneru je směrovat požadavky na backendový kontejner, k čemuž slouží Nginx. Tato kapitola se primárně věnuje implementaci na logické úrovni, popis uživatelského rozhraní je více popsán v kapitole 6.

Asynchronní komunikace

Veškeré požadavky na server jsou odesílány asynchronně, uživatel nikdy nečeká na načtení stránky. K asynchronní komunikaci je využita knihovna axios[1], která poskytuje jednoduché rozhraní nad XMLHttpRequest API[2]. Tohoto rozhraní je docíleno využitím `Promise`[3] objektů, které slouží k řízení asynchronních operací. Samotný objekt se může nacházet ve třech stavech.

- **pending** - Počáteční stav objektu.
- **fullfied** - Operace byla úspěšně dokončena .
- **rejected** - Operace skončila neúspěchem.

Nad vytvořeným objektem lze volat primárně tři metody, pomocí kterých lze přistoupit k jeho výsledku.

- **then** - Slouží k přijetí odpovědi v případě, že operace proběhla úspěšně.
- **catch** - Slouží k zachycení odpovědi, pokud operace neproběhla v pořádku
- **finally** - Provede se vždy, nehledě na výsledný stav objektu.

[1] https://github.com/axios/axios
[2] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Logika uživatelského rozhraní

Logická část frameworku Vue je rozdělena na celkem pět dílčích komponent a jednu stránku. Kompozice jednotlivých komponent je zobrazena na obrázku 5.1. Komponenta `Review` představuje hlavní část aplikace, která má na starost přijímání a posílání dat mezi dceřinými komponentami. Těmi jsou `History`, `Report`, `Upload` a `SourceCodeWrapper`.

- Jedním z požadavků na klientskou část aplikace je možnost uchovávat již nahrané zdrojové soubory. Právě tuto vlastnost zajišťuje komponenta `History`. Pokud uživatel již nějaké soubory nahrál, jeho URI obsahuje náhodně vygenerovaný řetězec, který jednoznačně identifikuje jeho nahrané soubory. K těm je umožněn přístup právě přes onen vygenerovaný řetězec. Pokud uživatel tento řetězec odešle jinému klientovi, který ho následně vloží do své URI, komponenta `History` se postará, aby druhý uživatel viděl všechny nahrané soubory prvního uživatele. Úlohou této komponenty ovšem není jenom zobrazovat historii uživatele, ale i reagovat na její změny. Tyto změny zahrnují přejmenování složek, ve kterých se nachází nahrané soubory, či jejich mazání. Veškeré operace zajišťuje komponenta asynchronně skrze serverové REST API.

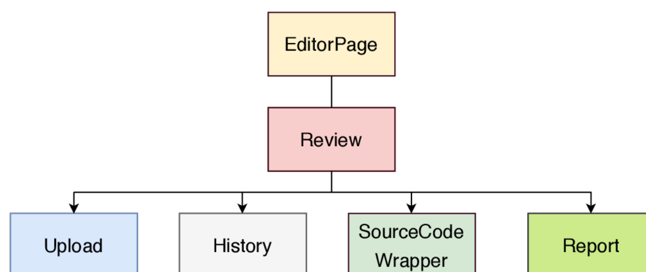
- Dalším požadavkem na aplikaci je zobrazení uživateli zdrojový report, což obstarává komponenta `Report`. Tento report je možné v případě potřeby editovat. Každá změna reportu je opět asynchronně uložena na server. V případě, že v komponentě `SourceCodeWrapper` je zobrazen detail označeného řádku, v zdrojovém reportu je tento řádek patřičně zvýrazněn.

- Úlohou komponenty `SourceCodeWrapper` je přehledné zobrazení zdrojových kódů na základě informací v reportu. Zdrojové kódy jsou tříděny do záložek, kde každá záložka nese název dle jména souboru, ve kterém se zdrojové kódy nachází. Komponenta `SourceCodeWrapper` v sobě obsahuje mimo jiné CodeMirror. Tato knihovna zajišťuje syntaktické zvýraznění zdrojového kódu, skrývání/odkrývání bloků kódu[4] a v neposlední řadě i zvýraznění řádků kódu, kterých se report týká. U těchto řádků je možné si dále zobrazit detailní informace dle reportu, např. kterého vlákna se řádek týká či jaký je jeho kontext/pozice v rámci tracebacku. Mezi jednotlivými částmi tracebacku je možné se interaktivně posouvat kliknutím, a to v obou směrech. Detailní informace o patřičných řádcích je samozřejmě možné libovolně skrývat, odkrývat či zkopírovat do schránky.

- Poslední dílčí komponentou je `Upload`, která zajišťuje asynchronní nahrávání zdrojových souborů na server.

Tento výpis obsahuje pouze nejdůležitější části frontendové služby. Ve skutečnosti se zde nachází ještě Vue Router[5], který zajišťuje směrování v rámci frontendu. Na základě URI adresy rozhodne, která z hlavních komponent bude vykreslena (EditorPage, ErrorPage). Tato komponenta je následně vložena do hlavní, kořenové komponenty, která je připojena k HTML šabloně.

[4] Blokem kódu je myšlen konkrétní rámec, tj. funkce, komentář či třída
[5] https://router.vuejs.org/guide/#html



Obrázek 5.1: Skladba Vue komponent

Veškerá komunikace mezi komponentami probíhá skrze komponentu `Review` tak, aby u dat byla zajištěna jejich správná reaktivita a byla tudíž vždy korektně aktualizovaná.

Kontext uživatele

Cílem aplikace je poskytnout uživateli nástroj pro rychlou revizi zdrojových kódů. Z tohoto důvodu zde není implementováno vytváření uživatelských účtů. Aplikace si nicméně musí umět držet kontext uživatele. To je zajištěno již zmíněným **náhodně** vygenerovaným řetězcem. Tento řetězec se vytvoří na backendu při prvním nahrání souborů uživatelem a následně je vložen do jeho URI. Řetězec následně slouží jako jednoznačný identifikátor nahraných souborů konkrétního uživatele. Veškeré operace následující operace uživatele (nahrání dalších souborů, přejmenování složek, modifikace reportu) jsou prováděny na základě onoho řetězce.

Proces načtení reportu

Zpracování reportu v unifikovaném formátu zajišťuje komponenta `SoruceCodeWrapper`. Report je přijat jako **kolekce struktur**. Každá struktura představuje jednu konkrétní část reportu a obsahuje o ní všechny potřebné informace. Struktury můžou být i vnořené, dle specifikace formátu reportu v kapitole 4.3. Při zobrazení či otevření záložky v uživatelském rozhraní, která obsahuje konkrétní soubor se zdrojovými kódy se spustí algoritmus, který provede provede průchod kolekcí, viz obrázek 8. Pro každý prvek této kolekce, který koresponduje s aktuálně otevřeným zdrojovým souborem provede dvě základní činnosti. Za prvé je nutné označit řádek, ke kterému se daný prvek kolekce, tedy část reportu, vztahuje. Dále je potřeba k tomuto řádku připojit dodatečné informace, které se k němu na základě reportu vztahují.

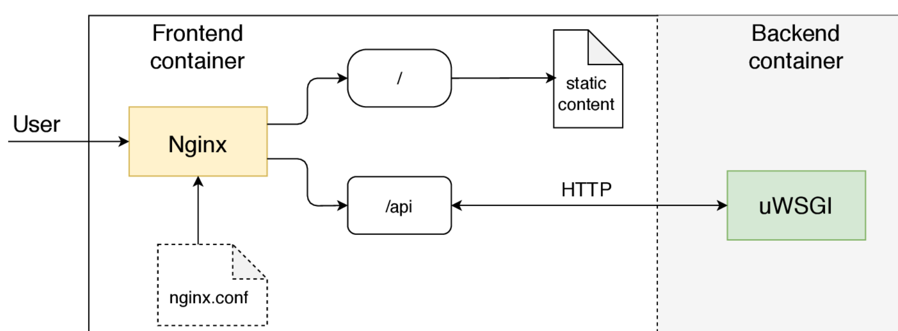
Algorithm 1 Highlight lines

```
1: procedure SearchLines(reportsCollection)            ▷ Get collection of reports
2:     for each report in reportCollection do
3:         if report contains traceback then
4:             searchLines(report[traceback])            ▷ Search again if object is nested
5:         else
6:             highlightLine(report)                     ▷ Process line highlight
7:             addLineWidget(report)                     ▷ Attach additional detail to current line
8:
9:
```

Výše uvedený algoritmus se spustí pouze při prvním zobrazení daného souboru se zdrojovými kódy. Při vícenásobných otevřeních stejného souboru v rámci jedné složky je stav záložky, která obsahuje soubor se zdrojovými kódy, kešován.

Směrování

Směrování v klientské části zajišťuje služba Nginx, která zároveň představuje vstupní bod do celé aplikace, viz obrázek 2.3. Služba dle URI požadavku zajistí jeho správné přesměrování. Pokud má požadavek tvar `<BASE_SERVER_URI>/api/<params>`, je pomocí reversní proxy přesměrován na backendový kontejner, který zajistí jeho zpracování. Detailnějšímu popisu formátu požadavků na REST API se věnuje kapitola 5.2. V případě, že má požadavek tvar , tak je zobrazeno uživatelské rozhraní. Konfigurace služby Nginx se nachází v souboru `nginx.conf`.



Obrázek 5.2: Směrování frontend kontejneru

V případě požadavku ve tvaru `<BASE_SERVER_URI>/<hash?>` je zajištěno ještě směrování v rámci uživatelského rozhraní pomocí modulu Vue Router, nicméně to obsahuje pouze dvě stránky. Z toho důvodu není na diagramu zakresleno.

Produkční nasazení

Produkční verze se od té vývojové poměrně liší. Hlavní změny spočívají v **optimalizaci** zdrojových kódu a **kompresí** odpovědí.

První vlastnost zajišťuje framework `Vue` v kombinace s nástroji Webpack[6] a Babel[7], které při spuštění v módu produkčního nasazení zabalí všechny zdrojové kódy a moduly dle diagramu závislostí. Při tomto procesu proběhnou optimalizace zdrojových kódů společně s jeho `transpilací` [10] do čistého javascriptu za účelem zpětné kompatibility. Transpilací rozumíme `souce-to-source` kompilaci, tedy proces, kdy se převede jeden typ zdrojového kódu na jiný typ. Díky tomu je možné využívat nejnovějších vlastností jazyka, případně jejich nadstaveb ve formě CoffeSscriptu[8] či TypeScriptu[9]. Transpilací zůstane zachována kompatibilita zdrojového kódu napříč prohlížeči.

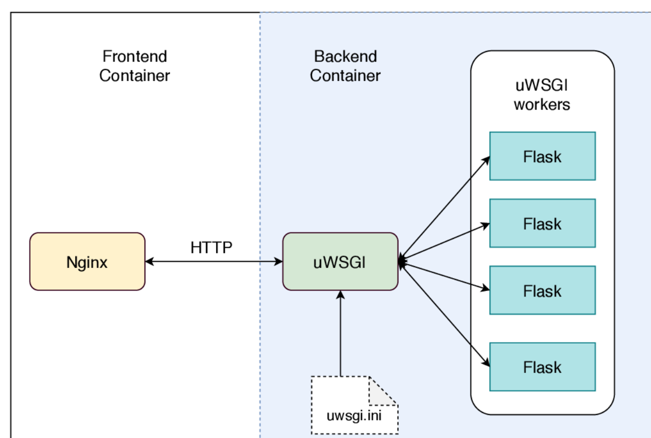
Kompresi odpovědí zajišťuje program Ngnix metodou GZIP. Díky kompresi je možné zmenšit velikost přenášených dat. Další metodou, kterou je možné zvýšit rychlost načtení

[6]https://webpack.js.org/
[7]https://babeljs.io/
[8]https://coffeescript.org/
[9]https://www.typescriptlang.org/

aplikace je využití **lazy loadingu**, neboli dynamického importování komponent. Dynamické importování je zajištěno pomocí **tovární metody**, které asynchronně vrací definici komponenty pomocí objektu `Promise` 5.1. Jinými slovy, pokud je u komponenty nastaveno její dynamické importování, načte se pouze tehdy, pokud je potřeba. Rozdělení aplikace pomocí dynamického importování se nazývá **code spliting**. Tento přístup může hrát velkou roli z hlediska rychlosti prvního načtení stránky, a to zejména u velkých aplikací, které se skládají z desítek komponent.

5.2 Backend

Primárním účelem backendového kontejneru je zajistit nahrání zdrojových souborů na server a zpracování reportu do jednotného formátu popsaného v kapitole 4.3. Vzhledem k tomu, že každý nahraný soubor má svůj životní cyklus, je nutné zajistit jednoduchou správu těchto souborů. K tomuto účelu je vybudováno REST API, které poskytuje jednotné rozhraní nad službami serveru. K tvorbě REST API byl použit framework Flask s rozšířením FlaskRESTful[10], které usnadňuje jeho tvorbu. Webový server, který je implicitně součástí frameworku Flask ovšem není vhodný na produkční nasazení, proto je využit aplikační webový framework uWSGI, který představuje mezivrstvu mezi serverovou aplikací a službou Nginx. uWSGI umožňuje spustit serverovou aplikaci, tedy Flask, ve více vláknech či procesech, tudíž je zlepšena dostupnost serveru při případném větším množství požadavků. Konfigurovat uWSGI server je možné přes argumenty příkazové řádky při jeho spuštění, nicméně při větším počtu parametrů je tento způsob značně nepřehledný. Z tohoto důvodu se zde nachází soubor `uwsgi.ini`, ze kterého je konfigurace načtena.



Obrázek 5.3: Zpracování požadavků backendovým kontejnerem

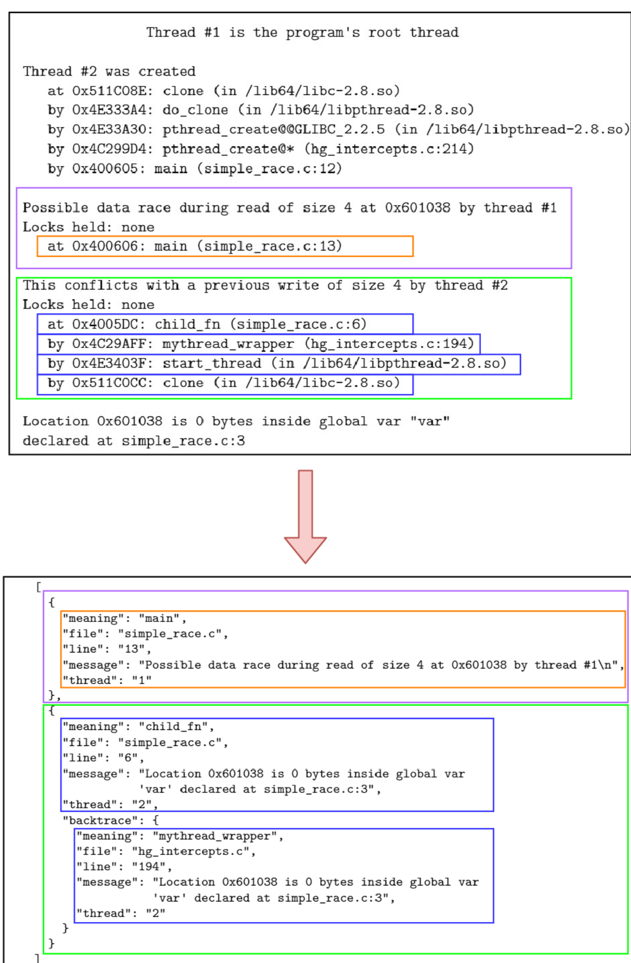
Implementaci backendu lze tedy rozdělit do dvou kategorií. První kategorie popisuje zpracování reportu do jednotného formátu, druhá kategorie popisuje implementaci REST API.

Proces zpracování reportu

Ihned poté, co server provede uložení nahraných souborů, začne zpracovávat zdrojový report. Samotné zpracování se skládá z několika kroků popsaných v následujících bodech.

[10] https://flask-restful.readthedocs.io/en/latest/

- **Výběr parseru** - V první řadě server vybere nástroj, tzv. parser, který bude zpracovávat report. Každý parser je reprezentován jednou třídou, která dědí z bázové třídy `GenericParser`, dle návrhu 4.3. Nástroj, který má být vybrán je specifikován v přijatém požadavku. Jestliže specifikován není, server vybere nástroj sám. Následně se dynamicky vytvoří instance třídy daného parseru, u které se invokuje metoda **parse**.
- **Zpracování** - Další krok spočívá v samotném zpracování reportu. Toto zpracování je provedeno pomocí sady regulárních výrazů. Každá část reportu je uložena do nového objektu `Report`, jejichž výsledná sada je uložena do listu. Vzhledem k tomu, že každý objekt má pevně dané atributy, do kterých je možné uložit informace z reportu, je zajištěn vždy stejný formát reportu. Tyto atributy korespondují s artefakty reportu specifikované v kapitole 4.3. Ukázkový převod reportu je možné vidět na obrázku 5.4.
- **Export/uložení** - Posledním krok zpracování reportu se skládá z převedení kolekcí objektů do datového typu slovník, který je jednoduše serializovatelný do formátu JSON. V tomto formátu je také již zpracovaný report uložen do souborového systému. Obrázek 5.4 zobrazuje ukázkové zpracování zdrojového reportu z nástroje Helgrind.



Obrázek 5.4: Ukázkový převod reportu do unifikovaného formátu.

Na výše uvedeném obrázku lze vidět demonstrační převod reportu produkovaném nástrojem Helgrind do unifikovaného formátu. Vzhledem k tomu, že na základě zpracovaného

reportu jsou v grafickém uživatelském rozhraní vyznačeny konkrétní řádky zdrojového kódu, poslední dvě části tracebacku do tohoto formátu zahrnuty nejsou, protože obsahují odkaz na knihovny.

Rozšiřitelnost

Aplikace je navržena tak, aby byla dále jednoduše rozšiřitelná. V případě implementace dalšího parseru je nutné dodržet pouze určitá pravidla. **Každý** parser musí dědit z bázové třídy `GenericParser`, ze které zároveň musí implementovat dvě metody. První z nich, `register` slouží k zaregistrování parseru. Po implementaci této metody bude nabídka podporovaných nástrojů v uživatelském rozhraní obsahovat tento parser. Zároveň tato metoda obsahuje definici výrazu, na jehož základě server vybere onen nástroj v případě, že nebyl specifikován v požadavku. Druhá metoda zvaná `parse` slouží k provedení samotného zpracování reportu. I zde je nutné dodržet určitá pravidla. Jednotlivé části reportu je nutné uložit do objektu `Report`. Objekt se vytváří pomocí **tovární metody**, uživatel tedy sám nemusí vytvářet žádné nové objekty. `Report` objekty musí být následně uloženy do již připraveného seznamu, ze kterého se následně vyexportují do souboru ve formátu JSON.

Serverové rozhraní

Aby bylo možné k službám serveru jednotně a jednoduše přistupovat, je nutné implementovat určitě rozhraní, které bude definovat formát komunikace. Toto rozhraní je vytvořeno pomocí REST API, které již bylo uvedeno v kapitole 4.3. K samotné implementaci byl využit framework Flask s rozšířením Flask-RESTful.

K přístupům ke zdrojům serveru slouží koncové body. Každý koncový je implementován samostatnou třídou. Tato třída může obsahovat metody `get`, `post`, `update` a `delete`, které odpovídají stejnojmenným HTTP metodám. Validní HTTP metody nad daným koncovým bodem jsou pouze ty, které jsou implementovány v patřičné třídě popisující konkrétní koncový bod. Následující tabulka vypisuje seznam koncových bodů serverového API včetně jejich URI a popisu.

metoda	**koncový bod**	**URI**	**popis**
GET	File	api/v1/file/<id>/<folder>	Získá všechny soubory dle identifikátoru <id>a složky <folder>
GET	Files	api/v1/files/<id>	Získá všechny složky včetně souborů dle identifikátoru <id>
POST	Files	api/v1/files/	Uloží na server nahrané soubory, zpracuje report a vrátí název složky
DELETE	Files	api/v1/files/<id>	Smaže obsah složky dle identifikátoru <id>
DELETE	Files	api/v1/files/<id>/<folder>	Smaže složku <folder>obsaženou ve složce dané identifikátorem <id>
PUT	Files	api/v1/files/<id>/<folder>	Změní název složky <folder>dle identifikátoru <id>
PUT	Report	api/v1/report/<id>/<folder>	Změna obsah zdrojového reportu obsaženého ve složce <folder>dle identifikátoru <id>
GET	Tools	api/v1/tools/	Získá seznam všech podporovaných nástrojů na zpracování reportu

Tabulka 5.1: Popis koncových bodů podporovaných serverem

V rámci přehlednosti je každý koncový bod, tedy třída, v separátním souboru. Tyto soubory se nachází ve složce endpoints, dle fyzického návrhu aplikace popsaného v kapitole 4.3. V případě rozšíření serveru o další koncové body je nutné vytvořit třídu, která bude dědit z bázové třídy `BaseEndpoint`. Dále už je třeba pouze daný koncový bod zaregistrovat v třídě `RepViewInterface`, která mu předá potřebné závislosti.

5.3 Spuštění aplikace

Jak již bylo avizováno v kapitole 4.1, celá aplikace je spustitelná pomocí služby Docker Compose. Prerekvizitou k úspěšnému spuštění aplikace je tedy služba Docker. K příkazům Docker Compose lze přistoupit přímo pomocí CLI[11], či přes program Make[12]. Využití utility Make spočívá především v automatizaci spuštění služeb. Aplikace má celkem tři možné cíle spuštění, přičemž každá z nich obsahuje jinou konfiguraci (docker-compose.yml, Dockerfile, porty, apod.).

První možností je spuštění aplikace ve vývojovém módu. Při tomto typu spuštění aplikace využívá standardních webových serverů pro Flask a Quasar, které jsou vhodné pro vývoj. Při této konfiguraci je umožněno tzv. **hot reloading**, tedy při editaci kódu se změny automaticky projeví v aplikaci uvnitř kontejneru. Dále je možno využít tzv. **remote interpret**, který slouží k ladění aplikace uvnitř kontejneru. Aplikaci je možné v této konfiguraci spustit pomocí příkazu 5.1.

```
> make dev
```

Zdrojový kód 5.1: Spuštění aplikace v konfiguraci pro vývoj.

Patrně nejdůležitější konfigurací spuštění je produkční verze. Ta obsahuje již produkční webové servery (Nginx, uWSGI), zabalený a minifikovaný javascriptový kód. Tato verze samozřejmě neumožňuje použití vývojářských nástrojů. Produkční verzi aplikace lze spustit příkazem 5.2.

```
> make prod
```

Zdrojový kód 5.2: Spuštění aplikace v produkční konfiguraci.

Poslední konfigurací spuštění je testovací mód. Ten v sobě navíc obsahuje kontejnery pro realizaci systémových a akceptačních testů. Bližšímu popisu testů včetně schématu zapojení kontejnerů se věnuje kapitola 7.1. Kontejnery `backend` a `frontend` jsou zde spuštěny v produkční verzi. Testy lze spustit pomocí příkazu 5.3.

[11] https://docs.docker.com/compose/reference/overview/
[12] https://www.gnu.org/software/make/

```
> make test
```

Zdrojový kód 5.3: Spuštění testovací konfigurace.

Příkazy jednotlivých konfigurací je samozřejmě možné změnit v souboru Makefile, konfiguraci samotných kontejnerů v souborech docker.compose.[dev | prod | test].yml a patřičných souborech Dockerfile. V případě nutnosti změnit porty jednotlivých služeb je nutno editovat soubor `.env` nacházející se podobně jako Makefile v kořenovém adresáři projektu. Tento soubor obsahuje proměnné běhového prostředí, které jsou vkládány do jednotlivých kontejnerů. V případě změny portu backendového kontejneru je nutné změnit jeho port i v konfiguraci služby Nginx, která neumožňuje načítat proměnné běhového prostředí.

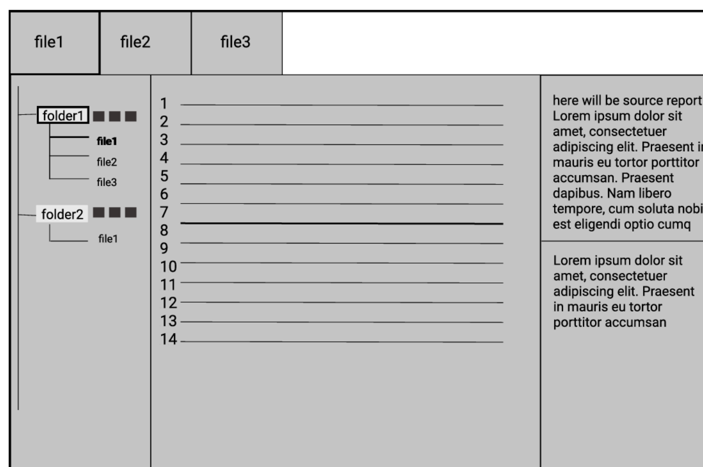
Kapitola 6

Uživatelské rozhraní webové aplikace

Grafické uživatelské rozhraní tvoří důležitou část projektu, protože skrze něj uživatel přistupuje k celé aplikaci. Je tedy nutné k této části vývoje přistupovat s náležitou váhou. Podobně jako při návrhu aplikační logiky, i zde je potřebné nejdříve provést důkladnou analýzu požadavků, aby bylo zřejmé, co vše uživatel bude chtít zobrazit, viz. kapitola 3. S analýzou požadavků se pojí tvorba počátečního návrhu grafického uživatelského rozhraní, který se nazývá **wireframe** neboli drátový model.

6.1 Wireframe

Wireframe slouží v oblasti vývoje webových stránek či aplikací k prezentaci a náhledu nového řešení. Smyslem wireframu je ujasnění rozložení jednotlivých částí rozhraní, případně jejich funkcionalit. Zde není nutné, aby grafické rozhraní obsahovalo designové prvky, naopak je vhodné držet se neutrálního stylu a barev. K Vytvoření wireframu byl použit nástroj Figma[1].



Obrázek 6.1: Prvotní rozložení prvků webového rozhraní

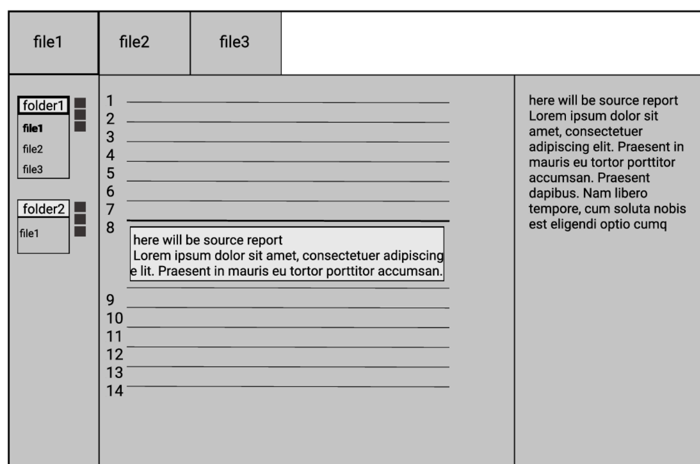
[1] https://www.figma.com/

Obrázek 6.1 ukazuje první verzi rozložení prvků grafického uživatelského rozhraní. Struktura vychází z analýzy požadavků na prohlížení zdrojových kódů 3. Horní část obsahuje záložky označující nahrané soubory v rámci složky, v tomto konkrétním případě `folder1`. V levé části jsou všechny uživatelem nahrané soubory tříděné do složek. Prostřední část se skládá z editoru, který obsahuje zdrojový kód vybraného souboru, zde konkrétně `file1`. Poslední důležitá část se týká reportu, který se k uvedenému zdrojovému kódu vztahuje. Jeho horní polovina obsahuje zdrojový report v podobě, v jaké ho uživatel nahrál. Spodní polovina zahrnuje jeho konkrétní, již zpracovanou část, která se týká zvýrazněného řádku.

Ačkoliv tato verze drátového modelu splňuje téměř všechny požadavky na webové rozhraní, obsahuje určité nedostatky. Strukturální nedostatky pomohl odhalit až funkční prototyp aplikace.

Zhodnocení výsledků

Zhotovení prototypu z wireframu na obrázku 6.1 vedlo k jistým strukturálním změnám, které vedly ke vzniku nového, finálního wireframu. Podstatné změny oproti první verzi je možnost zobrazení detailu u více řádků zároveň, což u první verze nebylo možné, protože detail byl zobrazen v pravé spodní části obrazovky, nikoliv však pod daným řádkem. Dále, sloupec obsahující historii nahraných souborů byl zjednodušen odebráním přebytečných linií ke každému souboru v rámci složky.



Obrázek 6.2: Upravená verze drátového modelu dle zjištěných poznatků

Z výše uvedených poznatků již lze přistoupit k návrhu a tvorbě grafického uživatelského rozhraní, jehož výstupem je výsledný grafický návrh aplikace.

6.2 Návrh grafického rozhraní

Při návrhu grafického rozhraní je nutné vzít v potaz nejen samotný vzhled aplikace, ale i správné rozmístění prvků rozhraní, vhodný výběr barev či jejich odstínů, logické souvislosti jednotlivých prvků, počet nutných kliknutí, které uživatel musí vykonat k určité činnosti, intuitivnost, doba odezvy aplikace, apod. Tyto vlastnosti se souhrnně nazývají **user experience**, neboli UX. Cílem návrhu grafického rozhraní je tedy dosáhnout co největší uživatelské přívětivosti aplikováním výše uvedených vlastností.

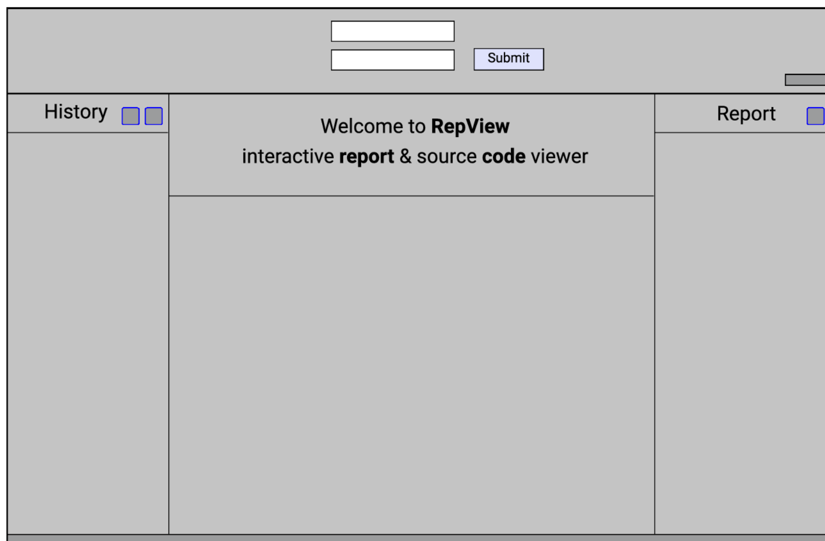
Návrh grafického rozhraní, potažmo UX je postupem orientovaný proces, který je vhodný rozdělit do několika logických celků, které poskytnou vývoji určitou strukturu [7]. Následující seznam detailněji popisuje jednotlivé fáze vývoje.

- **Analýza** - Pochopení uživatelských potřeb je klíčové pro dobrý návrh. Z toho důvodu cyklus začíná touto fází.
- **Design** - V této části iterace probíhá návrh a vytvoření základních konceptů vzhledu.
- **Prototyp** - Samotný vývoj prototypu lze rozdělit do několika dílčích částí dle úrovně věrnosti, jako jsou statické drátové modely, které můžou být následně rozšířeny o funkční prvky, dále základní prototypy, do kterých se implementují jednotlivé designové vlastnosti a posléze i funkcionalita.
- **Vyhodnocení** - Poslední část cyklu se zaměřuje na vyhodnocení a verifikaci vzhledu, jestli splňuje stanovené požadavky.

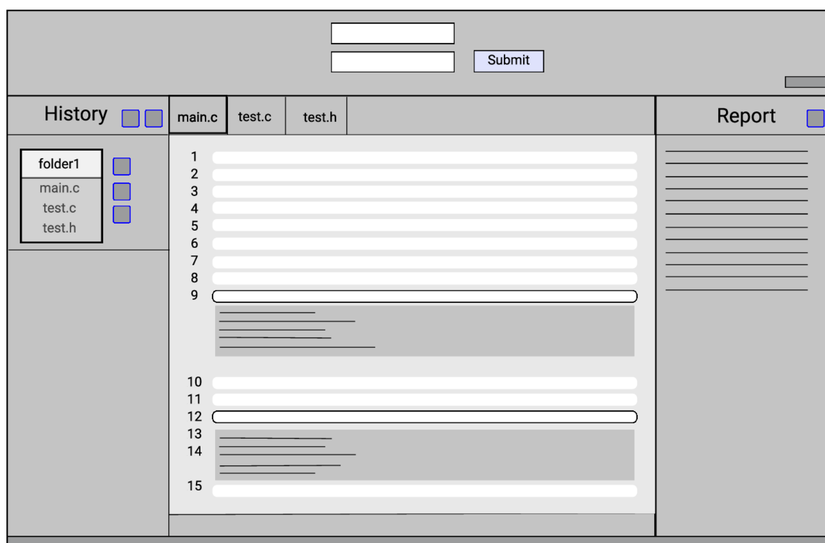
Techniky vyhodnocování UX

UX techniky jsou použity při při návrhu vzhledu za účelem zajištění lepší uživatelské přívětivosti. Technik lze ve skutečnosti najít celé množství [6], nicméně zde jsou uvedeny pouze ty, které byly při vývoji využity. Tyto techniky byly využity na již fungujícím prototypu aplikace dostupným přes webové rozhraní. Roli zákazníků zde představovali spolužáci VUT FIT.

- **Přemýšlení nahlas** - Kolekce technik, která spočívá v uživatelské interakci s aplikací s tím, že každou myšlenku uživatel vysloví nahlas. Smýšlení vývojáře či designéra je často odlišené od toho, jak uvažuje potencionální zákazník. Účelem techniky přemýšlení nahlas je tento rozdíl eliminovat.
- **Sledování očí** - Vzhledem k tomu, že jednostránková aplikace poskytuje všechen svůj obsah na jedné stránce, je důležité vhodné umístění jednotlivých prvků či jejich částí. Pozorováním pohybu očí lze zjistit, nad kterými částmi grafického rozhraní uživatel tráví větší dobu, jestli jeho pohyb očí není příliš nevyzpytatelný z důvodu nepřehlednosti či jaké pohyby jsou nejčastější.
- **Fly-on-the-Wall pozorování** - Technika pozorování Fly-on-the-Wall je podobná testování `drátového modelu`. Uživatel pracuje s aplikací dle zadání, a vývojář sleduje jeho činnosti. Důležitým faktorem je zde záměrné odstranění přímého pozorovatele z průběhu testu. Cílem toho pozorování je minimalizace zaujatost či vlivu, který by jinak mohl vzniknout při přímém zapojení pozorovatele do testu.



Obrázek 6.3: Výsledný prototyp stránky při prvním navštívení



Obrázek 6.4: Výsledný prototyp stránky při nahraných souborech a zobrazení detailu dvou řádků

Prototyp na obrázku 6.4, resp. 6.3 zobrazuje již finální verzi grafického uživatelského rozhraní. Na prototypu je také vidět korespondence hlavních prvků rozhraní s příslušnými komponentami dle návrhu v kapitole 4.2.

6.3 Výsledné řešení

Výsledné řešení grafického uživatelského rozhraní silně vychází z prototypů uvedených na obrázcích 6.3 a 6.4. Na implementaci byl kromě technologií HTML/CSS/Javascript využit také frontendový framework Quasar, který poskytuje velké množství komponent usnadňující vývoj.

Jak již bylo zmíněno, grafické uživatelské rozhraní se skládá se čtyřech základních komponent, které zde budou podrobněji popsány. Nejprve bude popsána část, která se skládá ze tří sloupců zobrazující historii, zdrojové kódy a report. Poměr velikostí jednotlivých sloupců lze libovolně upravovat tak, aby uživatel vždy viděl vše potřebné. Následující text blíže popisuje jednotlivé sloupce. Veškeré tlačítka či ikony, které budou dále popsány budou nazývány jménem, které se zobrazí při najetí myší na jejich obsah.

Mód zobrazení

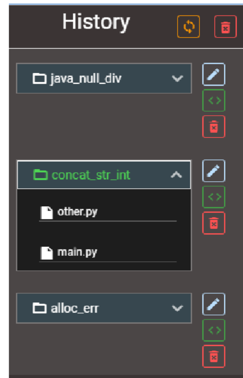
Ačkoliv se volba zobrazovacího módu může zdát jako nedůležitý faktor, stále více společností umožňuje uživatelům jejich aplikací je zobrazit v tmavém, kontrastním módu. S rostoucí poptávkou se přítomnost tmavého módu již netýká pouze zábavního průmyslu (chatovací aplikace, audio/video streaming), ale zasahuje do používání každodenních aplikací (tmavý mód v operačním systému iOS či Windows). Aplikace podporuje dva barevné módy zobrazení. Základní, neboli tmavý mód a světlý mód (viz příloha C). Přechod mezi těmito módy lze provést pomocí přepínače v pravé horní části uživatelského rozhraní.

Historie

V sloupci historie, který zobrazuje historii nahraných souborů konkrétního uživatele se nachází dvě hlavní tlačítka, **Synchronize** a **DeleteAll**. V případě, že nad stejnými zdrojovými soubory pracuje více uživatelů, tlačítko Synchronize slouží k synchronizaci souborů. DeleteAll dle názvu slouží k vymazání všech nahraných souborů daného uživatele.

Dále sloupce obsahuje již samotné nahrané soubory. Obrázek 6.5 zobrazuje konkrétně tři složky obsahující nahrané soubory, přičemž druhá složka je aktivní, tedy její obsah je nahrán v editoru, který bude popsán v následující podkapitole. Kliknutím na název složky je možné zobrazit, resp. skrýt její obsah. S každou složkou je možné provádět tři různé akce symbolizované patřičnými tlačítky.

- **Edit** - Změní název složky. Název složky nesmí obsahovat speciální znaky jako mezera či znak konce řádku. Pokud je operace úspěšná, daný adresář je příslušně přejmenován v souborovém systému.
- **Load** - Nahraje obsahu složky do editoru. V jednu chvíli může být samozřejmě aktivní pouze jedna složka.
- **Delete** - Smaže danou složku a veškerý její obsah.



Obrázek 6.5: Sloupec zobrazující historii

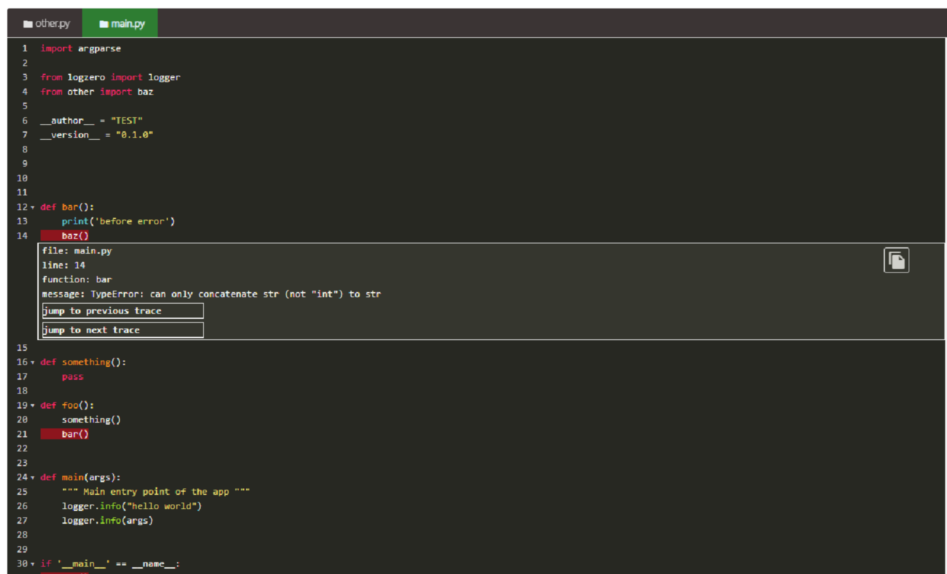
Pokud je libovolná akce není povolená, kupříkladu změna názvu složky na již existující název v rámci kontextu daného uživatele, je zobrazena patřičná chybová hláška informující uživatele o jeho chybě.

Editor

Sloupec Editor zobrazuje obsah aktivní složky, tedy soubory se zdrojovými kódy. Zdrojový kód je možné do editoru nahrát dvěma způsoby, buď nahráním nových souborů nebo kliknutím na tlačítko **Load** u příslušné složky v historii. V případě nahráním nových souborů se zdrojové kódy automaticky nahrají do editoru.

Samotný editor se skládá ze záložek, které reprezentují jednotlivé soubory a ze zdrojového kódu příslušící konkrétnímu souboru. Zdrojové kódy obsahují zvýrazněné řádky, které jsou označené v reportu a jsou syntakticky obarvené, dle specifikace požadavků. Při kliknutí na libovolný označený řádek je zobrazen jeho detail na základě informací v reportu. V případě, že se jedná o **traceback** je zobrazen i aktuální kontext dané části reportu. Tato situace je zobrazena na obrázku 6.6. Kontext se skládá ze dvou tlačítek, které symbolizují přechod mezi jednotlivými částmi tracebacku. Kliknutím na tlačítko se v editoru otevře soubor, ve kterém se následující, resp. předchozí část tracebacku nachází, pohled editoru se posune na daný řádek a automaticky se zobrazí detail onoho řádku.

Editor dále podporuje **code folding**, tedy vlastnost, kdy lze určité bloky kódu skrývat či zobrazovat. Tyto bloky kódu jsou označeny šipkou na daném řádku v postranním panelu. Typické části kódu, nad kterými lze provádět tuto operaci jsou funkce, třídy či komentáře. Poslední vlastností editoru je možnost zkopírovat si obsah dané části reportu do schránky kliknutím na tlačítko **CopyToClipboard** v pravé části detailu řádku.

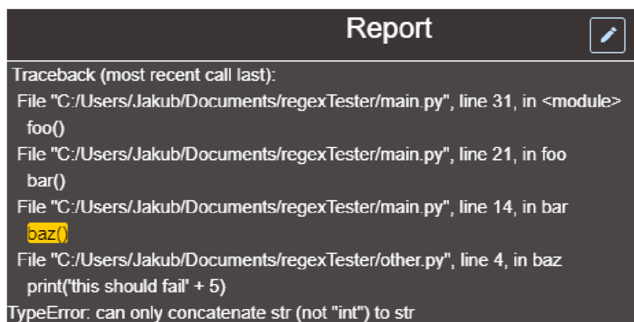


Obrázek 6.6: Sloupec zobrazující editor kódu

Vzhled editoru se samozřejmě, jako ostatní části uživatelského rozhraní, přizpůsobuje módu zobrazení, tedy tmavému či světlému režimu.

Report

Posledním sloupcem je komponenta obsahující informace o reportu, který se váže k aktivní složce, resp. zdrojovým souborům nahraným v editoru. Zde je možné provádět pouze jedinou akci, a tou je modifikace reportu vyvolaná tlačítkem **Update**. Zároveň, pokud je v editoru otevřený detail specifického řádku, je tento řádek patřičně vyznačen i v této části.



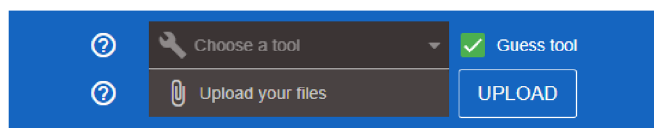
Obrázek 6.7: Sloupec zobrazující report zdrojových kódů

Barevné zabarvení aktuálně zobrazeného detailu uživateli pomůže ujasnit souvislost reportu a zdrojových kódů.

Nahrání zdrojových souborů

Poslední hlavní částí grafického uživatelského rozhraní je komponenta Upload, která zajišťuje nahrání nových zdrojových souborů. Krom tlačítka **Upload**, které obstarává samotné nahrání vybraných souborů, se zde nachází dvojice položek pro uživatelský vstup. Horní

slouží k výběru parsovacího nástroje. V základním zobrazení je tento výběr neaktivní, server vybere nástroj sám. Jestliže tato akce nebyla z nějakého důvodu úspěšná či uživatel chce nástroj vybrat sám, lze odškrtnout checkbox, který zpřístupní výběr nástroje. Dolní položka slouží pro výběr souborů ze souborového systému, které chce uživatel nahrát. Nahrané soubory musí obsahovat zdrojové kódy a korespondující report.



Obrázek 6.8: Komponenta zobrazující nahrání nových souborů

K oběma položkám uživatelského vstupu navíc náleží ikony, které při najetí myší zobrazí ke každému z nich nápovědu.

Kapitola 7

Testování

Testování je důležitou části fáze vývoje software. Testování lze popsat jako podmnožinu zajištění kvality produktu. Obecně lze testování rozdělit do několika úrovní, kterými jsou jednotkové testování, testování modulů. integrační, systémové a akceptační testování [14]. Ačkoliv má každá úroveň jiné účely, jednotlivé úrovně se vzájemně doplňují, nejsou tedy vůči sobě výlučné. Jednotlivé kategorie testů se provádějí v určitých fázích vývoje software, přičemž na konci každé iterace vývoje software by měly následovat ještě **regresivní** testy. Regresivní testy slouží k ověření, zda nově přidaná funkcionalita, refaktoring či změna kódů neovlivnily již fungující kód.

7.1 Automatizované testování

Testování aplikace probíhalo na celkově třech, níže popsaných rovinách. Jednotlivé typy testování jsou seřazeny vzestupně dle zařazení ve V-modelu[1].

Jednotkové testování

Jednotkové testování testuje správnost konkrétních, izolovaných fragmentů kódu, např. funkcí či tříd. Jeho cílem je odhalit chyby na nejnižší úrovni, což může mít za následek zmenšení rozsahu testování na vyšších úrovních. Jednotkové testy spadají společně s testy modulů a integračními testy do kategorie `white-box` testování. Při tomto způsobu tester vidí zdrojový kód. Testeři jsou většinou zároveň i programátoři dané aplikace.

Jednotkové testování bylo tomto projektu použito na testování jednotlivých parserů na serverové straně. Tyto testy pokrývají nejdůležitější funkce parseru, tj. ověřují správnost metod `parse` a `register`. Implementace testů je v jazyce Python s využitím testovacího frameworku `unittest`[2]. Toto testování je také součástí CI pipeline v repozitáři na Gitlabu[3], kde je projekt verzován. S každou změnou, která je vložena do repozitáře pomocí verzovacího programu Git[4] se pipeline spustí a provede jednotkové testy v rámci Continuous Integration.

[1] http://tryqa.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/
[2] https://docs.python.org/3/library/unittest.html
[3] https://pajda.fit.vutbr.cz/testos/repview
[4] https://git-scm.com/

Systémové testování

Systémové testy představují důležitou část testování software. Zaměřují se již na funkčnost aplikace jako na celek, zda splňuje formální specifikaci požadavků. Testy zahrnují komunikaci mezi službami, správné směrování požadavků uvnitř docker sítě a testování aplikačního rozhraní serveru. Tento způsob spadá společně s akceptačním testováním do kategorie `black-box` testování, kdy tester je odstíněn od konkrétní implementace testovaného software a zná pouze rozhraní aplikace.

Implementace testů je napsána v jazyce Python s využitím frameworku `pytest`[5]. Vzhledem k tomu, že aplikace implementuje otevřené API, tak je díky tomu možné využívat služeb serveru mimo uživatelské rozhraní. Tuto vlastnost zahrnují právě systémové testy, které přes běžné HTTP dotazování testují funkčnost aplikace.

Každý testovací případ se skládá z parametrů HTTP dotazu, kterými jsou URL adresa, metoda, hlavička či formulářová data. Nad výsledkem dotazu, tj. odpovědí, je následně provedena verifikace [18]. Ta se skládá z ověření návratového kódu, formátu odpovědi, kontroly hlavičky či případně samotné ověření získaných dat.

Systémové testy jsou spustitelné uvnitř vlastního dockerového kontejneru. Testy je možné je spustit buď samostatně či společně s akceptačními testy. Diagram kompozice testovacích kontejnerů je zobrazen na obrázku 7.1.

Akceptační testování

Akceptační testy slouží k výslednému ověření, zda software odpovídá požadavkům zadavatele. V reálném prostředí testování může probíhat společně se zadavatelem ve stanoveném prostředí. Tuto činnost lze částečně simulovat automatizovanými testy grafického uživatelského rozhraní, které provádí akce, které by jinak prováděl uživatel.

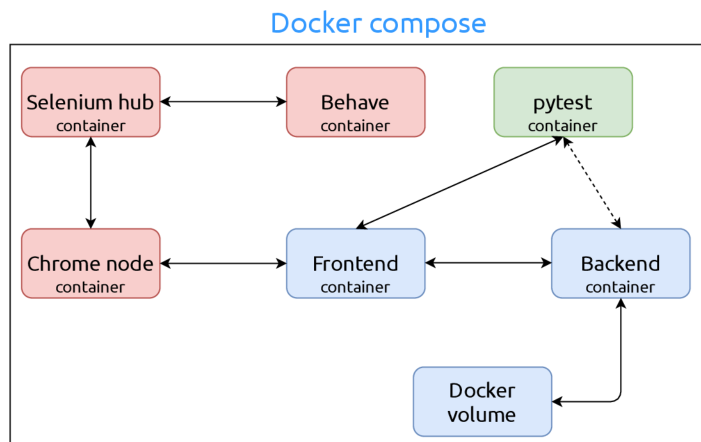
Testy jsou implementovány v jazyce Python, společně s knihovnami Behave[6] a Selenium[7].

- **Behave** - slouží pro vývoj řízený chováním (Behavior-driven development), což je technika agilního vývoje software. Základem jsou scénáře, které popsují akce jednotlivých případů. Scénáře jsou popsány v jazyce `Gherkin`[8], což je speciální jazyk navržený tak, aby byl čitelný běžným uživatelem. Každý scénář se skládá se série základních kroků, jejichž pořadí je nutné dodržet. Danými kroky jsou **Given** →**When** →**Then**. Ke každému scénáři koresponduje již napsaný test v jazyce Python, který implementuje jednotlivé kroky popsané v scénáři.

- **Selenium** - slouží k automatizaci testování uživatelského rozhraní [17]. K využití Selenia je nutný tzv. **WebDriver**, který provede dané úkony ve webovém prohlížeči. Při spuštění testů klientská aplikace, která obsahuje Selenium testy zasílá požadavky přes protokol HTTP právě na WebDriver, který funguje jako HTTP server. Přijímá požadavky z klientské aplikace, provádí konkrétní akce v prohlížeči a následně zasílá zpět odpověď.

Testy tedy obsahují implementované scénáře, kde každý scénář provádí úkony s webovým rozhraním. Testy jsou kontejnerizované a spustitelné přes službu Docker Compose dle diagramu 7.1.

[5] https://docs.pytest.org/en/latest/
[6] https://github.com/behave/behave
[7] https://github.com/SeleniumHQ/selenium/
[8] https://behave.readthedocs.io/en/latest/gherkin.html



Obrázek 7.1: Struktura testovacích kontejnerů

Z výše uvedeného diagramu je vidět, že systémové testy prováděné kontejnerem **pytest** lze provádět primárně přes kontejner **frontend**, který zajistí přesměrování požadavků na backendový kontejner, ale lze i požadavky zasílat rovnou na **backend**.

7.2 Manuální testování

Nedílnou součástí vývoje je samozřejmě i manuální testování, kdy programátor ručně testuje funkcionalitu stávajících či nově přidaných prvků. Manuální testování se týkalo převážně webového rozhraní, které se prolínalo s jeho vývojem.

Postman

Postman[9] slouží k API testování. Aplikace umožní vytvořit jednoduchého API klienta, který umožňuje zasílat a přijímat dotazy. Postman uživateli standardně dovoluje nastavit URL, metodu, hlavičky či formát dotazu. Dotazy je možné shlukovat do kolekcí a následně jednotně spouštět. Z libovolného dotazu je možné také vyexportovat jeho implementaci v podporovaném jazyce, např. Java.

Aplikace díky jejímu jednoduchému rozhraní umožnila snadnější vývoj a testování REST API. Testována byla zejména serverová strana aplikace pomocí HTTP dotazovaní, které ověřovalo správnost komunikace (formát, získaná data, hlavičky).

Testované prohlížeče

Aplikace byla testována na celkem třech noteboocích, přičemž každý obsahoval jeden z následujících operačních systémů. Konkrétně to byly systémy Windows 10 Pro, Ubuntu 18.04 a MacOS Catalina. Celkem byly otestovány prohlížeče Google Chrome, Mozilla Firefox, Microsoft Edge a Safari.

Krom drobných změn barev či okrajů aplikace běží ve všech prohlížečích kromě Safari korektně. Safari má problém s vykreslením světlého a tmavého módu. Z tohoto důsledku je aplikace při použití prohlížeče Safari nepoužitelná.

[9] https://www.postman.com/

7.3 Vývojové nástroje

Následující kapitola popisuje výčet nejdůležitějších nástrojů, které byly při vývoji využity.

PyCharm

Pycharm[10] je populární integrované vývojové prostředí společnosti `JetBrains`. Nástroj obsahuje intelliSense, který usnadňuje našeptávání při psaní kódu, integrované nástroje jako Git či Docker, živou kontrolu syntaxe, podporu ladění či refaktorování kódu. Ladění lze spustit lokálně či vzdáleně. Při vývoji projektu, konkrétně jeho backendové části, bylo využito primárně vzdálené ladění, které umožňuje připojit se k dockerovému kontejneru a využít jeho interpret (zde konkrétně pro jazyk Python). Dále PyCharm obsahuje možnost instalace dodatečných balíků či pluginů, čehož bylo využito pro tvorbu frontendové části, konkrétně podpora jazyka Javascript.

Docker Desktop

Docker Desktop[11] poskytuje minimalistické grafické uživatelské rozhraní pro správu dockerových kontejnerů. To je obzvláště užitečné při nutnosti kontejnery v rámci služby `Docker Compose` nezávisle na sobě vypínat či zapínat. U každého kontejneru je dále možné zobrazit si jeho informace ze standardního výstupu, konfigurační detaily či logy. U každého kontejneru je také možné spustit interaktivní shell a vykonávat příkazy uvnitř kontejneru.

Aplikace poskytuje rozhraní pro základní práci s kontejnery, v případě nutnosti využití pokročilejších funkcí je nutné použít standardně příkazovou řádku.

Chrome DevTools

Chrome DevTools[12] je sada vývojářských nástrojů dostupná přes webové prohlížeč Google Chrome. Tyto nástroje poskytují základní prostředí pro ladění a diagnostiku webových stránek a aplikací. Je zde možné zobrazit si strom HTML elementů, kterým je možné dynamicky měnit vlastnosti. Z pohledu samotné logiky webové aplikace toho nástroje umíce ovšem více. Je možné zde ladit javascriptový kód, přičemž jsou zde dostupné standardní ladicí informace. Nejdůležitějšími jsou aktuální stav hodnot proměnných, stav zásobníku volání, aktuální rámec či zachytávané události. Z hlediska síťové komunikace je možné zobrazit si průběh všech dotazů se serverem, společně s jejich hlavičkami a návratovými kódy. Poslední důležitou věcí je možnost zobrazení dat držených v prohlížeči, například stav `local storage`. Veškeré kontrolní výpisy či informace o chybě v aplikace jsou dostupné právě přes tuto sadu nástrojů.

Dále bylo využito rozšíření `Vue.js DevTools`[13], které poskytuje rozšířené ladící prostředí pro framework Vue.js. Lze zde vidět stromovou strukturu komponent, podobně jako v standardních nástrojích pro vývojáře HTML elementy. U každé komponenty je zobrazen její aktuální stav, tj. hodnoty všech atributů dané komponenty. Dále rozšíření nabízí nástroje pro správu směrování na klientské straně, diagnostiku výkonosti či zabezpečení.

[10] https://www.jetbrains.com/pycharm/
[11] https://www.docker.com/products/docker-desktop
[12] https://developers.google.com/web/tools/chrome-devtools
[13] https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd

Vue CLI

Framework Vue je možné využívat několika způsoby, v tomto projektu byl konkrétně využit Vue CLI[14]. Tato verze umožňuje mimo jiné přistoupit ke konfiguraci frameworku také přes webové uživatelské rozhraní. Zde je možná správa všech závislostí a balíků projektu, které obsahuje. Závislosti je možné přehledně mazat, aktualizovat či stahovat nové. Nástroj dále nabízí možnost sestavení projektu jak ve vývojové, tak v produkční verzi. Důležité je využití analyzujících služeb, které programátorovi poskytnou přehled o velikosti zabalené aplikace. K tomu byl využit dodatečný modul `Webpack Bundle Analyzer`[15]. Výsledkem analýzy tohoto modulu je zobrazení velikostí jednotlivých závislostí projektu ve výsledné sestavené aplikaci. Programátor na základě těchto informací může uvážit o code splittingu za účelem zmenšení velikosti sestavené aplikace.

[14]https://github.com/vuejs/vue-cli
[15]https://github.com/webpack-contrib/webpack-bundle-analyzer

Kapitola 8

Závěr

Výstupem interpretu, překladače či nástroje sloužícího k analýze zdrojových kódů je výstupní report. Tento report v sobě mnohdy obsahuje nejen informaci o konkrétní chybě či varování, nýbrž i spoustu dodatečných informací. Tyto informace mohou, ale nemusí být důležité k pochopení významu reportu. Ujasnění souvislosti mezi reportem a zdrojovými kódy je tedy klíčové pro jeho správnou interpretaci.

Výsledkem této práce je webový nástroj RepView, jehož cílem je umožnit uživateli provést přehlednou a interaktivní revizi zdrojového kódu na základě informací v reportu. Nástroj je implementován jako jednostránková aplikace bez nutnosti autentizace z důvodu co možná nejsnazšího použití aplikace. Samotná aplikace je rozdělena do dvou dockerových kontejnerů, které mezi sebou nezávisle komunikují, dle architektury mikroslužeb. Ve frontendovém kontejneru běží primárně služba Nginx, která servíruje statická data a poskytuje reversní proxy na backendový kontejner. Statická data tvoří grafické uživatelské rozhraní, které bylo vyvinuto nad frameworky Vuejs a Quasar. Backendový kontejner obsahuje webový aplikační server uWSGI, který poskytuje rozhraní nad frameworkem Flask. Pomocí tohoto frameworku bylo vytvořeno REST API, které umožňuje jednotný přístup k službám backendového kontejneru. Mezi základní služby backendového kontejneru patří zpracování zdrojového reportu do unifikovaného formátu a uložení či získání uživatelem nahraných souborů. Unifikovaný formát reportu je typu JSON a a skládá se z kolekcí struktur.

Samotná aplikace umožňuje nejen nahrát zdrojové soubory, na základě nichž je provedena revize, ale i si tyto soubory uchovávat v rámci kontextu uživatele. Každý uživatel má svoji vlastní historii nahraných souborů, nad kterou může provádět patřičné úkony. Uživatel je identifikován náhodně vygenerovaným řetězcem v jeho URI adrese při prvním nahrání souborů.

Ačkoliv aplikace umožňuje provedení rychlé revize zdrojových kódů na základě reportu, do budoucna by bylo vhodné využít potenciál aplikace a neomezovat její rozvoj pouze na práci s reporty. Jinými slovy by aplikace mohla být součástí větší platformy zaměřující se na edukaci či podporu výuky programování, která by pomáhala ujasnit jednotlivé souvislosti v kontextu programování. Mohlo by se jednat kupříkladu o vizualizaci práce s dynamickou alokací paměti, práce s vlákny či procesy.

Literatura

[1] Clark, L. *A crash course in just-in-time (JIT) compilers – Mozilla Hacks - the Web developer blog*. Feb 2017. [Online; accessed 15-May-2020]. Dostupné z: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/.

[2] contributors, M. *Concurrency model and the event loop*. Mozilla. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop.

[3] Fielding, R. T. a Taylor, R. N. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, 2000. [Online; accessed 10-June-2020]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[4] Fotis, A. *Vue Virtual Dom*. Medium, 2020. [Online; accessed 3-March-2020]. Dostupné z: https://medium.com/js-dojo/vue-virtual-dom-13af62d2be41.

[5] Gross, S. *FuzzIL: Coverage guided fuzzing for JavaScript engines*. 2018. Disertační práce. Master's thesis, Karlsruhe Institute of Technology, 2018. https://saelo.github.io/papers/thesis.pdf,.

[6] Hanington, B. a Martin, B. *Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers, 2012. [Online; accessed 10-May-2020].

[7] Hartson, R. a Pyla, P. Chapter 2 - The Wheel: UX Processes, Lifecycles, Methods, and Techniques. In: Hartson, R. a Pyla, P., ed. *The UX Book (Second Edition)*. Second Edition. Boston: Morgan Kaufmann, 2019, s. 27 – 48. DOI: https://doi.org/10.1016/B978-0-12-805342-3.00002-3. ISBN 978-0-12-805342-3. [Online; accessed 5-May-2020]. Dostupné z: http://www.sciencedirect.com/science/article/pii/B9780128053423000023.

[8] Jaramillo, D., Nguyen, D. V. a Smart, R. Leveraging microservices architecture by using Docker technology. In: *SoutheastCon 2016*. 2016, s. 1–5. [Online; accessed 05-March-2020]. Dostupné z: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7506647.

[9] Kočí, R. a Křena, B. *Úvod do softwarového inženýrství*. 2019. [Online; accessed 6-february-2020]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FIUS-IT%2Flectures%2FIUS1.pdf&cid=12208.

[10] Kyriakou, K. D., Chaniotis, I. K. a Tselikas, N. D. The GPM meta-transcompiler: Harmonizing JavaScript-oriented Web development with the

upcoming ECMAScript 6 "Harmony" specification. In: *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*. 2015, s. 176–181.

[11] Modak, A., Chaudhary, S. D., Paygude, P. S. a Ldate, S. R. Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes? In: *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. 2018, s. 7–12.

[12] Mouat, A. *Understanding Volumes in Docker*. 2017. [Online; accessed 19-March-2020]. Dostupné z: https://blog.container-solutions.com/understanding-volumes-docker.

[13] Nath, P. *Is Python an interpreted language?* Medium, Jul 2018. [Online; accessed 19-April-2020]. Dostupné z: https://medium.com/@prithajnath/is-python-an-interpreted-language-2906e38f6e36.

[14] Smrčka, A. *Testování a dynamická analýza - Úvod a pojmy testování*. 2018. [Online; accessed 13-June-2020]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FITS-IT%2Flectures%2F1-uvod.pdf&cid=13359.

[15] Solovei, V., Olshevska, O. a Bortsova, Y. The difference between developing single page application and traditional web application based on mechatronics robot laboratory onaft application. *Automation of technological and business processes*. 2018, sv. 10, č. 1. [Online; accessed 17-February-2020]. Dostupné z: https://journals.onaft.edu.ua/index.php/atbp/article/view/874/950.

[16] Uday, H. *Anatomy of Docker*. Medium, 2018. [Online; accessed 9-March-2020]. Dostupné z: https://itnext.io/getting-started-with-docker-1-b4dc83e64389.

[17] Vila, E., Novakova, G. a Todorova, D. Automation Testing Framework for Web Applications with Selenium WebDriver: Opportunities and Threats. In: *Proceedings of the International Conference on Advances in Image Processing*. New York, NY, USA: Association for Computing Machinery, 2017, s. 144–150. ICAIP 2017. DOI: 10.1145/3133264.3133300. ISBN 9781450352956. [Online; accessed 25-June-2020]. Dostupné z: https://doi.org/10.1145/3133264.3133300.

[18] Wenhui, H., Yu, H., Xueyang, L. a Chen, X. Study on REST API Test Model Supporting Web Service Integration. In: *2017 ieee 3rd international conference on big data security on cloud (bigdatasecurity), ieee international conference on high performance and smart computing (hpsc), and ieee international conference on intelligent data and security (ids)*. May 2017, s. 133–138. DOI: 10.1109/BigDataSecurity.2017.35.

[19] Wikipedia contributors. *List of ECMAScript engines — Wikipedia, The Free Encyclopedia* [https://en.wikipedia.org/w/index.php?title=List_of_ECMAScript_engines&oldid=950347026]. 2020. [Online; accessed 23-May-2020].

Příloha A

Obsah přiloženého média

Přiložené médium obsahuje následující soubory a složky:

- **/backend** - Zdrojový kódy pro backendový kontejner.
- **/doc** - Dokumentace aplikace. Obsahuje README soubor a přiloženou textovou část bakalářské práce ve formátu PDF.
- **/examples** - Ukázková sada zdrojových kódů a reportů sloužící k demonstraci aplikace.
- **/frontend** - Zdrojové kódy pro frontendový kontejner.
- **/tests** - Testovací sada aplikace.
- **.env** - Soubor obsahující definici proměnných ěbhového prostředí.
- **.gitlab-ci.yml** - Soubor obsahující konfiguraci Gitlab pipeline.
- **docker-compose.dev.yml** - Konfigurační soubor služby docker compose pro spuštění aplikace ve vývojovém režimu.
- **docker-compose.prod.yml** - Konfigurační soubor služby docker compose pro spuštění produkční verze aplikace.
- **docker-compose.test.yml** - Konfigurační soubor služby docker compose pro spuštění testů aplikace.
- **Makefile** - Makefile soubor pro automatizace spuštění služeb.
- **README.md** - README soubor obsahující základní informace o aplikaci.

Příloha B

Ukázková sada reportů

```
1  Traceback (most recent call last):
2    File "C:/Users/Jakub/Documents/regexTester/main.py", line 31, in <module>
3      foo()
4    File "C:/Users/Jakub/Documents/regexTester/main.py", line 21, in foo
5      bar()
6    File "C:/Users/Jakub/Documents/regexTester/main.py", line 14, in bar
7      baz()
8    File "C:/Users/Jakub/Documents/regexTester/other.py", line 4, in baz
9      print('this should fail' + 5)
10 TypeError: can only concatenate str (not "int") to str
```

Zdrojový kód B.1: Zdrojový report zobrazující traceback z Pythonu

```
1  [
2    {
3      "backtrace": {
4        "backtrace": {
5          "backtrace": {
6            "file": "main.py",
7            "line": "10",
8            "meaning": "baz",
9            "message": "TypeError: can only concatenate str (not 'int') to str"
10         },
11         "file": "main.py",
12         "line": "15",
13         "meaning": "bar",
14         "message": "TypeError: can only concatenate str (not 'int') to str"
15       },
16       "file": "main.py",
17       "line": "22",
18       "meaning": "foo",
19       "message": "TypeError: can only concatenate str (not 'int') to str"
20     },
21     "file": "main.py",
22     "line": "32",
23     "meaning": "<module>",
24     "message": "TypeError: can only concatenate str (not 'int') to str"
25   }
26 ]
```

Zdrojový kód B.2: Zpracovaný report do unifikovaného formátu

```
main.c: In function 'main':
main.c:17:5: error: expected ';' before 'return'
     return 0;
     ^~~~~~
test.c: In function 'invokeError':
test.c:5:1: error: expected ';' before '}' token
 }
 ^
```

Zdrojový kód B.3: Zdrojový report představující výstup z nástroje GCC.

```
    [
      {
        "file": "main.c",
        "line": "17",
        "meaning": "main"
        "message": "expected ';' before 'return'",
        "to": "5"
      },
      {
        "file": "test.c",
        "line": "5",
        "meaning": "invokeError",
        "message": "expected ';' before '}' token",
        "to": "1"
      }
    ]
```

Zdrojový kód B.4: Zpracovaný report do unifikovaného formátu

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Example.baz(Example.java:18)
        at Example.bar(Example.java:13)
        at Example.foo(Example.java:9)
        at Example.main(Example.java:5)
```

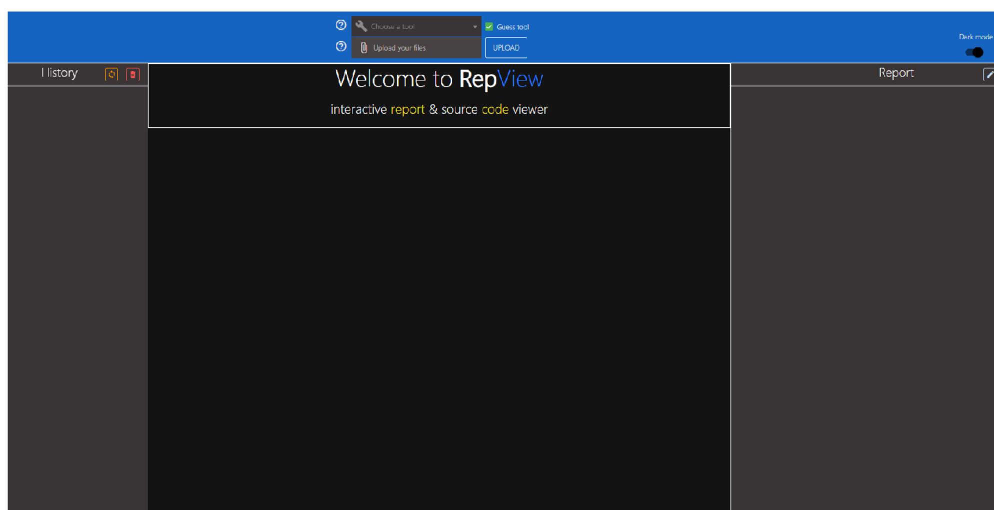
Zdrojový kód B.5: Zdrojový report přestavující výstup z Javy.


```
[
  {
    "backtrace": {
      "backtrace": {
        "backtrace": {
          "file": "Example.java",
          "meaning": "Example.baz",
          "line": "18",
          "message": "java.lang.ArithmeticException: / by zero",
          "thread": "main"
        },
        "file": "Example.java",
        "meaning": "Example.bar",
        "line": "13",
        "message": "java.lang.ArithmeticException: / by zero",
        "thread": "main"
      },
      "file": "Example.java",
      "meaning": "Example.foo",
      "line": "9",
      "message": "java.lang.ArithmeticException: / by zero",
      "thread": "main"
    },
    "file": "Example.java",
    "meaning": "Example.main",
    "line": "5",
    "message": "java.lang.ArithmeticException: / by zero",
    "thread": "main"
  }
]
```

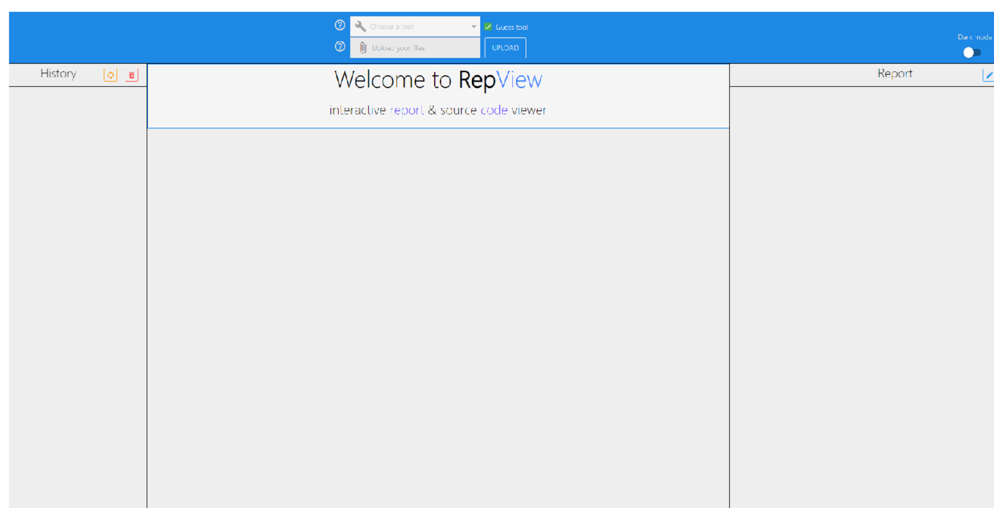

Zdrojový kód B.6: Zpracovaný report do unifikovaného formátu

Příloha C

Módy zobrazení GUI



Obrázek C.1: Úvodní obrazovka v tmavém módu



Obrázek C.2: Úvodní obrazovka v světlém módu