VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

GRAFICKÉ UŽIVATELSKÉ ROZHRANÍ PRO SPRÁVU VÝKONNOSTNÍCH PROFILŮ

GRAPHICAL USER INTERFACE FOR PERFORMANCE CONTROL SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MARTINA GRZYBOWSKÁ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ FIEDOR**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Grzybowská Martina**
Obor: Informační technologie
Téma: **Grafické uživatelské rozhraní pro správu výkonnostních profilů**
Graphical User Interface for Performance Control System
Kategorie: Uživatelská rozhraní

Pokyny:

1. Seznamte se s projektem Perun---správcem výkonnostních profilů. Seznamte se se současnými přístupy pro tvorbu grafických a webových uživatelských rozhraní.
2. Navrhněte grafické uživatelské rozhraní pro nástroj Perun se zaměřením na globální statistiku projektů. Na základě návrhu vytvořte demonstrační testovací aplikaci.
3. Implementujte uživatelské rozhraní zaměřené na desktopové systémy podporované nástrojem Perun.
4. Řešení demonstrujte na alespoň třech projektech s netriviální historií.

Literatura:

- Pew, Stephen: Information Dashboard Design: The Effective Visual Communication of Data
- Domovská stránka projektu PerfRepo: https://github.com/PerfCake/PerfRepo
- Domovská stránka projektu Perun: https://github.com/tfiedor/perun

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Fiedor Tomáš, Ing.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Jedným z najčastejších spôsobov testovania výkonu systému je technika profilácie programu vykonávajúca zber údajov o spotrebe zdrojov a ich následné vyhodnotenie vedúce k detegovaniu potenciálnych výkonnostných zmien, ktorých existencia môže mať negatívny dopad na vyvíjaný systém. Pre realizáciu procesu profilácie a komplexnejšej správy výkonu aplikácie už existuje niekoľko zavedených riešení. *Perun* patrí medzi novších správcov výkonnosti, sprostredkováva ako automatizáciu vytvárania, tak aj správu výkonnostných profilov projektov. Súčasná verzia však poskytuje len terminálové užívateľské rozhranie a nie je tak vhodná pre nasadenie do napr. *cloudu*. Cieľom tejto práce je špecifikovať, navrhnúť a implementovať grafické užívateľské rozhranie pre nástroj *Perun*. Výsledné užívateľské rozhranie cieli ako na možnosť vykonávať základnú funkcionalitu nástroja, ako je zber profilovacích dát podľa zadanej konfigurácie, ich následné spracovanie či efektívnu vizualizáciu, tak aj na možnosť prehľadne informovať o stave degradácie výkonu medzi jednotlivými verziami zobrazovaných projektov. Riešenie je demonštrované na troch netriviálnych verziovacích systémoch anotovaných výkonnostnými profilmi.

Abstract

One of the most frequent ways to test system performance is the program profiling technique, which carries out a collection of resource consumption data and its subsequent evaluation leading to the detection of performance changes, whose existence may have a negative impact on the system in development. For the realization of the profiling process and more complex application performance management, there are several established solutions. Perun belongs among the newer performance managers, it provides automatization of creating as well as managing of the performance profiles. However, the current version only offers a console user interface, therefore it is not suitable for deployment to e.g. cloud. The main objective of this thesis is to specify, design and implement a graphical user interface for Perun. The resulting interface targets the core functionality such as profiling data collection based on the pre-defined configuration, its subsequent postprocessing or effective visualization, as well as the ability to clearly give information about the status of performance degradation among individual project versions. The solution is demonstrated on three nontrivial version control systems annotated by performance profiles.

Kľúčové slová

Užívateľské rozhranie, správa výkonu, správa verzií, jednostránková webová aplikácia, Vue.js

Keywords

User interface, performance control, version control, single-page web application, Vue.js

Citácia

GRZYBOWSKÁ, Martina. *Grafické uživatelské rozhraní pro správu výkonnostních profilů*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Fiedor

Grafické uživatelské rozhraní pro správu výkonnostních profilů

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracovala samostatne pod vedením pána Ing. Tomáša Fiedora. Uviedla som všetky literárne pramene a publikácie, z ktorých som čerpala.

.....
Martina Grzybowská
10. mája 2018

Poďakovanie

Rada by som touto cestou poďakovala vedúcemu práce Ing. Tomášovi Fiedorovi za konzultácie, rady, pripomienky a odbornú pomoc pri tvorbe práce. Ďalej ďakujem svojej rodine a priateľovi za veľkú podporu počas celého štúdia.

Obsah

Kapitola 1

Úvod

Hlavnou úlohou funkcionálneho testovania je zisťovať, či testovaný systém spĺňa zamýšľanú funkcionalitu, a zároveň odhaľovať prípadné chyby. Majorita súčasných firiem už má existujúce a dobre fungujúce funkcionálne testy integrované vo vývojovom procese, napr. v rámci tzv. priebežnej integrácie. Menej zavedenou oblasťou testovania, avšak v poslednej dobe dôležitou disciplínou, je výkonnostné testovanie — forma nefunkcionálneho testovania, ktorá si kladie za cieľ stanoviť výkonnosť systému pod určitou záťažou a identifikovať tak jeho kritické miesta. Napriek tomu, že v súčasnosti existuje viacero kvalitných nástrojov, málo z nich ponúka možnosť plnej automatizácie procesu profilácie či následnej komplexnej správy vytvorených výkonnostných profilov. Dôkladná správa výkonu je tak v praxi zväčša celkom opomínaná.

Pri správe výkonnostných profilov bez automatizácie je užívateľ nútený všetky zozbierané dáta manuálne anotovať a organizovať. Ručná manipulácia s veľkým množstvom dát je ale vysoko náchylná na chybovosť, hrozí tak strata presnej histórie sledovaných zmien vo vývoji. Pre správu zmien v projektoch existujú dedikované systémy pre správu verzií. Pre správu verziovania výkonnosti vznikol v rámci výskumnej skupiny VeriFit nástroj *Perun: Performance under control* [4]. *Perun* ponúka plnú funkcionalitu pre automatizáciu procesu profilácie aj pre správu profilovacích dát; umožňuje zozbierané dáta skladovať, ďalej spracovávať a výsledky efektívne zobrazovať pomocou sady vizualizačných techník. *Perun* však v momentálnej podobe disponuje len textovým vstupno-výstupným užívateľským rozhraním, ktoré značne obmedzuje jeho vizualizačné a interpretačné možnosti a znemožňuje tak možnosť širšieho nasadenia napr. do *cloudu*. Hlavným cieľom tejto práce je špecifikovať, navrhnúť a implementovať grafické užívateľské rozhranie pre nástroj *Perun* so zameraním na globálnu štatistiku projektov a rozšíriť tak použiteľnosť *Perunu*.

Štruktúra dokumentu. Kapitola 2 sa venuje uvedeniu do problematiky a vybraným riešeniam pre správu výkonnostných profilovacích dát. Kapitola 3 približuje architektúru, vnútornú štruktúru a funkcionalitu nástroja *Perun*, pre ktorý je v rámci práce vytvárané grafické užívateľské rozhranie. V kapitole 4 je priestor venovaný stručnej analýze dostupných technológií pre tvorbu webových aplikácií. Kapitola 5 popisuje zozbierané požiadavky pre rozhranie, na základe ktorých je popísaný návrh štruktúry komponentov rozhrania v kapitole 6. Kapitola 7 diskutuje vybrané zaujímavosti z implementácie a kapitola 8 experimentálne demonštruje a overuje správnosť funkcionality implementovaného rozhrania.

Kapitola 2

Princípy správy výkonnosti

Výkonnostné testovanie je jedným z prostriedkov pre zaistenie a overenie kvality software. Zvyčajne sa testovaním zisťuje rýchlosť, kapacita a stabilita systému. Cieľom výkonnostného testovania je vyhľadať v systéme slabé miesta, ktoré spôsobujú pokles výkonu či znižujú stabilitu celého systému [11].

Dôležitým aspektom profilovacích dát zozbieraných počas výkonnostej analýzy (združených do tzv. „výkonnostných profilov“) je verzia projektu, nad ktorou bola analýza vykonaná — tzn. konkrétny bod v histórii projektu pozostávajúci z množiny zmien. Pri aktuálnej verzii projektu je prítomnosť výkonnostých zmien testovaná voči profilom predošlých verzií (tzv. *baseline* profilom). Pre potreby presnej detekcie zmien výkonu a ich spätnej lokalizácie je však nutné vhodným spôsobom uchovávať celú históriu výsledkov výkonnostnej analýzy a neobmedzovať sa len na bezprostredné zmeny. Súčasným problémom je tzv. „postupná degradácia“, pri ktorej výsledky testovania voči *baseline* profilom tesne vyhovujú nastaveným prahovým hodnotám, avšak nie pri zohľadnení širšieho časového rozsahu.

Zachovanie tejto histórie je možné docieliť viacerými technológiami — jednou z možností je napríklad siahnuť po niektorej z moderných NoSQL[1] databáz (napr. MongoDB[2]). Aj napriek ich vysokej škálovateľnosti je však nevýhodou tohto riešenia fakt, že databázy sú príliš robustné, generické a hlavne nie sú prispôsobené pre prácu s výkonnostnými profilmi — vyžadujú totiž implementáciu špecifického rozhrania pre ich manipuláciu. Alternatívou sú systémy pre správu verzií, ktoré však vyžadujú manuálnu anotáciu a správu výkonnostných profilov. Poslednou možnosťou sú špecializované systémy pre správu výkonnostných profilov, ako je napríklad *PerfRepo* alebo *Perun*, ktoré sa snažia spomenuté nedostatky odstrániť a poskytnúť dedikované riešenie pre správu výkonu programov.

2.1 Systémy pre správu verzií

Systém pre správu verzií (Version Control System — VCS) je systém, ktorý slúži na zaznamenávanie zmien vykonaných v projekte počas doby jeho vývoja. Týmto umožňuje realizovať nad uloženým projektom operácie ako navrátenie vybraných súborov či celého projektu do určitého predošlého stavu, porovnávať zmeny vykonané v čase či v prípade zanesenia

[1] tzv. Not Only SQL — koncept označujúci databázy, ktoré využívajú k uskladneniu a spracovaniu dát iné prostriedky než klasické relačné databázy

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

chyby odhaliť zodpovednú osobu [10]. Systémy pre správu verzií vo všeobecnosti delíme na tri typy:

- **Lokálne verziovacie systémy** pozostávajú z jednoduchej databázy, do ktorej sú ukladané všetky zmeny vykonané na súboroch zaradených pod správu revízií. Jedným z takýchto systémov je *Revision Control System* (RCS), ktorý pracuje na princípe uchovávania sérií „záplat“ (tzv. *patches*) — rozdielov medzi súbormi. Za pomoci skladania „záplat“ potom dokáže zrekonštruovať podobu súboru z ľubovoľného časového okamihu. RCS však pracuje len s jednotlivými súbormi, nedokáže pracovať s celým projektom naraz.

- **Centralizované verziovacie systémy** vznikli ako riešenie pre umožnenie vývoja jedného projektu na viacerých fyzických systémoch. Pri týchto systémoch už existuje jedna centrálna kópia projektu, najčastejšie uložená na serveri, do ktorej prispieva každý zo zúčastnených vývojárov. Celá história verzií projektu je uložená len na serveri a na klientskej strane sa nachádza len aktuálna verzia. Príkladom takýchto systémov sú *Subversion*[3] a *Perforce*[4].

- **Distribuované verziovacie systémy** pracujú na podobnom princípe ako centralizované verziovacie systémy, avšak pri týchto systémoch klient disponuje presnou kópiou (tzv. *clone*) repozitára a celou históriou projektu. Medzi distribuované verziovacie systémy patrí systém *Git*, ktorému sa bližšie venuje sekcia 2.1.1.

Pre realizáciu správy výkonnostných profilov jednotlivých verzií projektu cielenú na otvorenú komunitu sú spomedzi spomínaných troch typov vhodnou voľbou distribuované verziovacie systémy — súčasťou správy profilov je potom aj manipulácia so samotnými verziami projektu. Základy správy verzií projektu budú predstavené na systéme *Git*.

Verziovacie systémy sami o sebe neposkytujú podporu automatizácie výkonnostného testovania a správy jeho výsledkov. Nakoľko je potrebné uchovávať výkonnostné profily pre každú verziu projektu, manuálna správa môže byť pri väčších projektoch náročná a náchylná na chybovosť, čo môže spôsobiť stratu presnosti histórie výkonu projektu. Distribuované systémy však poskytujú dobrý základ pre realizáciu správcu výkonnostných profilov verzií.

2.1.1 Systém *Git*

Git je rýchly, škálovateľný, distribuovaný systém pre správu verzií s neobvykle bohatou sadou príkazov, ktorá poskytuje vysokoúrovňové operácie, rovnako ako plný prístup k vnútornej funkcionalite [10].

Systém *Git* bol pôvodne vyvinutý Linusom Torvaldsom ako otvorený systém pre správu verzií pri vývoji jadra operačného systému Linux; v súčasnosti je jedným z najpoužívanejších VCS. Keďže sa jedná sa o distribuovaný verziovací systém, na každom fyzickom systéme je lokálna kópia plnohodnotným repozitárom s celou históriou vývoja projektu a kompletnou správou verzií. Konkrétne zmeny v stave projektu sú najprv pridávané do lokálnej kópie, a následne je lokálny repozitár synchronizovaný s hlavným repozitárom (zvyčajne sa jedná o vzdialený repozitár nachádzajúci sa na serveri). V prípade, že v upravovaných súboroch

[3] https://subversion.apache.org/
[4] https://www.perforce.com/

v skutočnosti nie sú detegované žiadne zmeny, súbory nie sú ukladané nanovo; *Git* sa len odkáže na predchádzajúci identický súbor [8].

Git sa od ostatných verziovacích systémov líši hlavne v spôsobe, akým zaobchádza s dátami. Väčšina verziovacích systémov si zachováva súbor s pôvodným obsahom a iba udržiava zoznam zmien obsahu vykonaných v čase; systém *Git* naproti tomu uvažuje o uložených dátach ako o sérii snímok (*snapshots*). Pri ukladaní aktuálneho stavu projektu do lokálneho repozitára (*commit*) je zachytená snímka všetkých súborov v repozitári v danom časovom okamihu, následne je uložená do repozitára ako objekt typu *blob*[5] a odkaz na ňu je uložený do vytváraného objektu verzie (*commit object*). Tento *commit* objekt obsahuje taktiež dodatočné informácie ako je identifikácia autora zmien či textová správa (*commit message*). Po konkrétnom *commite* sú teda do repozitára projektu pridané celkovo tri typy objektov: *bloby* dát; strom, ktorý spisuje obsah adresára a určuje, ku ktorému súboru prináležia vytvorené *bloby*; *commit* s metadátami a referenciami. V kontexte správy profilov potom jeden *commit* môžme považovať za tzv. „malú“ (ďalej len *minor*) verziu projektu, ktorej prináleží množina nameraných výkonnostných profilov. Týmto je možné získať úzky vzťah medzi funkcionalitou a výkonnostnými zmenami.

Pred uložením nového stavu projektu je najprv z obsahu súboru alebo z adresárovej štruktúry vypočítaný kontrolný súčet, ktorý následne slúži ako odkaz na danú verziu. Jedná sa o reťazec zložený zo štyridsiatich hexadecimálnych znakov vytvorený za pomoci mechanizmu SHA-1[6]. Touto funkcionalitou *Git* súčasne zaisťuje integritu dát; vzhľadom na to, že je kontrolný súčet vytvorený ešte pred uložením dát, nie je možné čokoľvek zmeniť bez toho, aby táto zmena ostala nepovšimnutá [10].

Keďže všetky zmeny vykonané na základe aktuálnej verzie sú najprv ukladané do lokálneho repozitára, môže na rôznych fyzických systémoch zároveň existovať viacero rôznych zmien založených na rovnakej pôvodnej verzii. Z tohto dôvodu systém *Git* ponúka možnosť vetvenia (*branching*) — odklonenia od hlavnej vetvy (*master*) vývoja, ktoré slúži k štrukturovaniu vývoja projektu izoláciou zmien [8]. Na rozdiel od ostatných verziovacích systémov, ktoré mnohokrát pri vetvení kopírujú celý obsah zdrojového adresára, je v systéme *Git* proces vetvenia odľahčený a mimoriadne efektívny. Objekt verzie, ktorý je vytváraný pri ukladaní aktuálneho stavu projektu do lokálneho repozitára, obsahuje okrem dodatočných informácií aj odkaz na jeden alebo viac objektov, ktoré tomuto objektu predchádzali — objekty rodičovských verzií. Počet odkazov záleží od druhu *commitu*; inicializačný *commit* nemá žiadne odkazy, obyčajný *commit* má práve jeden odkaz a *commit*, ktorý je výsledkom spojenia (*merge*) dvoch a viacerých vetiev, má toľko odkazov, koľko mu predchádzalo rodičovských objektov. Vetva je potom v skutočnosti len referenciou na špecifický objekt verzie, pri ktorej sa oddelila od hlavnej vetvy [10]. V kontexte správy verzií je možné vetvy považovať za tzv. „veľké“ (ďalej len *major*) verzie projektu — ich existencia však nie je tak kľúčová pre zachovanie histórie výkonu.

Git poskytuje vstavané nástroje grafického užívateľského rozhrania (Graphical User Interface — GUI), ktoré sú prispôsobené len pre jeden účel a nezaoberajú sa funkcionalitou, ktorá pre nich nie je potrebná. Sú nimi `git-gui` (nástroj pre vytváranie *commitov*) a `gitk` (grafický prehliadač histórie repozitárov). Existuje taktiež mnoho grafických rozhraní a nástrojov tretích strán; od špecifických pre isté úkony až po plnohodnotné rozhrania, ktoré pokrývajú všetky možnosti práce so systémom *Git* [10]. Príkladom je *GitHub*, ktorého grafické rozhranie je dobrou inšpiráciou pre rozhranie pre správu výkonnostných profilov, a preto je podrobené bližšej analýze.

[5] binary large object — kolekcia binárnych dát uložených ako jedna entita

[6] Secure Hash Algorithm 1 — jedna zo skupiny kryptografických transformačných funkcií

2.1.2 *GitHub*

GitHub je jednou z najrozšírenejších internetových hostiteľských služieb pre správu verzií otvorených projektov založených na systéme *Git*. Umožňuje užívateľom zapojiť sa do spolupráce na projektoch pomocou tzv. *fork* funkcionality — jedná sa o vytvorenie vlastnej vetvy projektu, čím je užívateľovi umožnené vlastné experimentovanie s projektom. Zdrojový kód pochádzajúci z originálnej vetvy môže poslúžiť ako stavebný kameň pre novú aplikáciu, ale aj pre vytvorenie novej funkcionality pre pôvodnú aplikáciu. Po dokončení práce na odklonenej vetve môže užívateľ vytvoriť žiadosť o pridanie do originálnej vetvy (tzv. *pull request*), ktorá obsahuje prehľad všetkých zmien oproti pôvodnému zdrojovému kódu — v pôvodnom systéme *Git* je ekvivalentné správanie možné dosiahnuť pomocou príkazu `git request pull`. Vývojári spolupracujúci na projekte tak majú možnosť preskúmať kód a vyjadriť sa k prípadným zmenám (*code-review*). Po schválení administrátorom repozitára sú vytvorené zmeny pričlenené do vetvy *master* [6]. Vývojári spolupracujúci na projekte majú taktiež možnosť kedykoľvek vytvoriť pripomienku (*issue*) a poukázať na prípadé chyby či nezrovnalosti, ktoré mohli v kóde spozorovať.

Jednou z veľkých výhod služby *GitHub* je solídne webové užívateľské rozhranie, ktoré sprehľadňuje prácu s verziami projektov. Umožňuje zviditeľniť zaujímavé repozitáre za pomoci hlasovacieho systému, ktorý funguje na základe pridávania hviezd (*stars*), a taktiež sledovať (*watch*) zmeny v repozitároch, o ktoré užívateľ prejavil záujem. V roku 2017 bola taktiež predstavená desktopová aplikácia *GitHub Desktop*[7], ktorá je dostupná pre platformy Windows a macOS. Jedná sa o otvorený projekt napísaný v jazyku TypeScript, založený na aplikačných rámcoch (tzv. *framework*) Electron a React.

2.1.3 *Codacy*

Codacy je webový nástroj určený pre statickú analýzu a automatickú kontrolu kvality zdrojového kódu obsiahnutého v sledovaných repozitároch systému *Git*. Nejedná sa teda priamo o verziovací systém, ale o jeho nadstavbu (tzv. *wrapper*). V súčasnej dobe sú podporované takmer všetky najpoužívanejšie programovacie jazyky; kompletný zoznam sa nachádza na oficiálnych stránkach nástroja[8]. Projekt je do *Codacy* možné automaticky importovať z podporovaných hostiteľských služieb ako *GitHub*, *BitBucket* či *GitLab*.

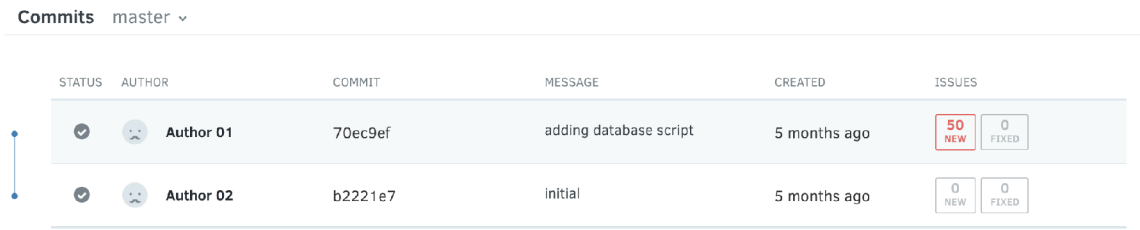
Jednou z hlavných výhod tohto nástroja je jeho intuitívne užívateľské rozhranie, ktoré efektívne zobrazuje výsledky statickej analýzy — z pohľadu tejto práce je však zaujímavé z toho dôvodu, že jeho celková architektúra vyhovuje požiadavkam, ktoré logicky plynú z prvotnej predstavy o grafickom užívateľskom rozhraní pre správu výkonnostných profilov. Rozhranie v prvom rade umožňuje jednoducho zobrazovať sledované repozitáre a ich základné údaje a po výbere konkrétneho repozitára ponúka zobrazenie ďalších informácií a možností.

Zobrazenie fyzických súborov projektu. V rámci konkrétneho fyzického súboru *Codacy* monitoruje zmeny v pokrytí zdrojového kódu, jeho zložitosti a prípadných výskytoch duplicitných častí. Každá informácia je zreteľne znázornená a každý súbor je ohodnotený známkou v závislosti od jeho štatistík. Z obrázku 2.1 je napr. možné vyčítať, že súbor `index.php` je kandidátom na refaktor kódu kvôli vysokému počtu nájdených chýb a s tým spojenej nízkej klasifikácii kvality.

[7] https://desktop.github.com/
[8] https://support.codacy.com/

Obrázok 2.1: Ukážka zobrazenia súborov projektu v nástroji *Codacy*. Pri každom súbore je uvedená jeho celková zložitosť, počet nájdených problémov či duplicít a výsledné hodnotenie kvality kódu.

Zobrazenie *commitov* projektu. Jednou z hlavných výhod *Codacy* je možnosť monitorovať kvalitu každého *commitu* a vyhodnocovať jeho dopad na projekt z viacerých rôznorodých hľadísk (napr. bezpečnostného či štýlového hľadiska). Táto funkcionalita tak umožňuje jednoducho pozorovať vývin kvality projektu počas celej doby jeho vývoja. Jednotlivé *commity* sú zobrazované v rámci vetvy vývoja, ku ktorej prináležia — *Codacy* umožňuje natívne pracovať s rôznymi vetvami, implicitne je ale analýza vykonávaná len nad vetvou *master*. Ďalšie vetvy je však možné pridať v lokálnych nastaveniach. Ukážka zobrazenia jednotlivých *commitov* a prepínania vetiev vývoja sa nachádza na obrázku 2.2, kde je napr. možné vidieť, že pridanie databázového skriptu zaviedlo päťdesiat nových chýb a tým malo vysoký dopad na kvalitu kódu.



Obrázok 2.2: Ukážka zobrazenia jednotlivých *commitov* v nástroji *Codacy*. Pri každom *commite* sú uvedené základné informácie a nové/opravené problémy.

2.2 Špecializované systémy

Dedikované systémy pre správu výkonnostných profilov ponúkajú oproti verziovacím systémom mimo iné automatizáciu správy a ukladania výsledkov výkonnostných testov alebo odstránenie potreby manuálnej anotácie a ďalších úkonov náchylných na chybovosť. Často taktiež disponujú technikami pre efektívnu interpretáciu alebo porovnávanie výsledkov testov. Jedným z takýchto systémov je aplikácia *PerfRepo*, ktorá bola vytvorená pre otvorenú komunitu. Ďalším zástupcom je potom systém *Perun*, ktorý je vyvíjaný v rámci výskumnej skupiny VeriFIT a je taktiež hlavným predmetom tejto práce.

2.2.1 *PerfRepo*

PerfRepo (*Performance result repository*) je webová aplikácia od spoločnosti RedHat, slúžiaca pre zachytávanie a archiváciu výsledkov výkonnostných testov, zjednodušenie ich porovnávania a základné zautomatizovanie detekcie výkonnostnej regresie medzi zostavami (*build*) [3]. Táto aplikácia je napísaná na platforme Java EE, pričom klientska časť je vytvorená pomocou technológie JSF 2.2. Pre samotný beh aplikácie sa používa aplikačný server WildFly a pre uloženie perzistentných dát je využívaná relačná databáza PostgreSQL [11].

PerfRepo je pridruženým nástrojom aplikácie *PerfCake*, ktorá slúži pre automatizované generovanie pracovnej záťaže (tzv. *workload*). Výsledky testovania z nástroja *PerfCake* sú do aplikácie *PerfRepo* nahrávané pomocou komponentu *destination plugin* alebo pomocou REST API. V súčasnosti je ale priamy vývoj *PerfRepo* pozastavený a dochádza len k občasným úpravám a opravám chýb, ktoré je možné nahlásiť na webovej stránke[9].

Veľkou nevýhodou tejto aplikácie je jej pôvodné grafické užívateľské rozhranie, ktoré je značne neintuitívne a práca s ním je nekomfortná. Medzi jeho hlavné nedostatky patria absencia našepkávania a nedostatočné overovanie vkladaných hodnôt, čo spôsobuje neočakávané chyby. Samotné chybové hlášky informujúce o týchto chybách v mnohých prípadoch nenesú potrebnú informáciu, z ktorej by užívateľ dokázal určiť, aká chyba vlastne nastala.

Odstráneniu týchto nedostatkov sa venoval vo svojej diplomovej práci Jiří Grunwald [11]. Pre implementáciu nového rozhrania zvolil CSS framework PatternFly (nadstavba frameworku Bootstrap) a JavaScriptový framework AngularJS.

[9] JBoss Issue Tracker PERFREPO — https://issues.jboss.org/projects/PERFREPO

Kapitola 3

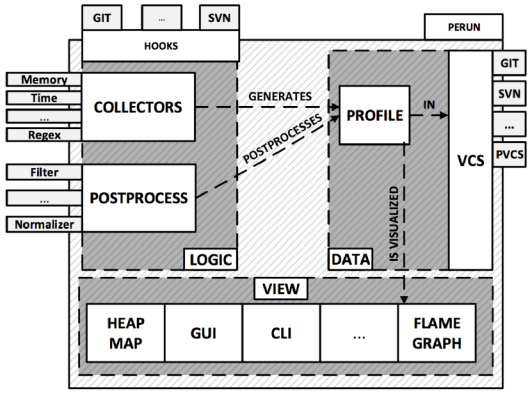
Perun: Systém pre správu výkonnostných profilov verzií

Perun (*Performance Under Control*) je otvorený projekt, ktorý vznikol v rámci výskumnej skupiny VeriFIT s cieľom automatizácie vytvárania a správy výkonnostných profilov programov. Súčasne obsahuje sadu nástrojov umožňujúcich automatizáciu behu regresných výkonnostných testov, spracovanie už existujúcich profilov a efektívnu interpretáciu výsledkov [4]. Jeho funkcionalita rozhrania a spôsobu uloženia dát je inšpirovaná systémami pre správu verzií (konkrétne systémom *Git*). Instancia *Perunu* je začlenená do používaného verziovacieho systému, čím umožňuje automaticky uchovávať a spravovať výkonnostné profily pre jednotlivé verzie projektu. Každý profil tak prináleží jednej konkrétnej verzii, ktorá obsahuje okrem iného aj informácie o pozmenených častiach kódu, čase, kedy boli tieto zmeny vykonané a ich autorovi. Obsah samotného výkonnostného profilu je potom tvorený zozbieranými dátami a dodatočnými informáciami ako je konfigurácia aplikácie pri zbere dát.

3.1 Architektúra

Architektúru nástroja *Perun* je možné rozdeliť na tri hlavné časti — dátovú časť, pohľad a logickú časť. Toto rozdelenie je ilustrované na obrázku 3.1.

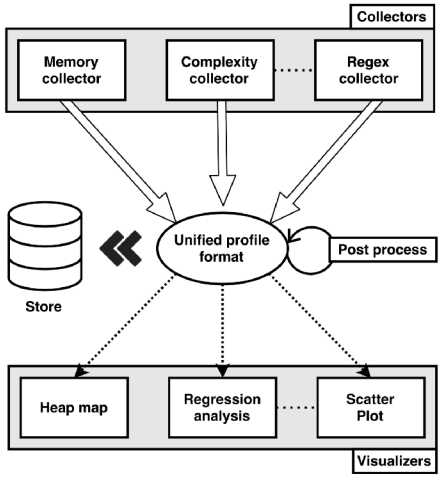
V *dátovej časti* sa nachádzajú moduly pre správu profilov a napojenie na podporované verzovacie systémy. *Logická časť* sa stará o automatizáciu, vytváranie profilov a manipuláciu s nimi. Táto časť zahŕňa taktiež napojenia na podporované verziovacie systémy za pomoci tzv. *hooks*, čím je umožnená automatizácia vytvárania profilov. *Pohľad* je potom nezávislý balíček modulov, ktorý obsahuje vstupno-výstupné užívateľské rozhranie a interpretácie profilov. Jedná sa o textové rozhranie (CLI) doplnené o vizualizáciu výstupov za pomoci rôznych vizualizačných techník.



Obrázok 3.1: Schéma rozdelenia architektúry nástroja *Perun* na jednotlivé časti. Zdroj: [4].

3.2 Životný cyklus výkonnostného profilu

Vytváranie výkonnostných profilov zaisťujú zberače profilovacích dát (tzv. *collectors*), pričom ich výstup môže byť následne spracovaný pomocou jednotiek nazvaných postprocesory — *postprocessors* (napr. normalizácie alebo filtrovania hodnôt). Spracované profily sú potom spolu s dodatočnými informáciami komprimované za pomoci metódy *zlib*[1], uložené do adresárovej štruktúry nástroja a priradené konkrétnej *minor* verzii projektu. Profily môžu ďalej byť interpretované za pomoci podporovaných vizualizačných techník, ako napríklad graf toku dát alebo stĺpcový graf. Životný cyklus profilu je zobrazený na obrázku 3.2.



Obrázok 3.2: Schéma životného cyklu výkonnostného profilu. Zdroj: [4].

3.3 Formát výkonnostného profilu

Interný formát výkonnostných profilov využívaný v nástroji *Perun* je založený na dátovom formáte *JSON*[2]. Hlavná schéma formátu profilu je zobrazená na obrázku 3.3, plná špecifikácia sa nachádza v dokumentácii nástroja *Perun* [4].

Okrem profilovacích dát zachytených zbernými jednotkami, profil taktiež obsahuje všeobecné informácie o priebehu procesu profilovania, ktoré sú rozdelené do šiestich hlavných regiónov:

Origin. Tento región obsahuje identifikáciu *minor* verzie projektu, na základe ktorej daný výkonnostný profil vznikol. Je prítomný len v profiloch, ktoré ešte neboli priradené príslušnej verzii — slúži ako prevencia pred chybným priradením profilu nesprávnej verzii. Spravidla sa jedná o profily nachádzajúce sa v adresári `./perun/jobs`, ktorý bude bližšie popísaný v sekcii 3.5. Pred uložením profilu do perzistentného úložiska je tento región odstránený.

Header. Obsahuje základné špecifikácie profilu; konkrétne kľúče regiónu sú vyznačené na obrázku 3.4. Kľúč *type* definuje hrubý typ profilu (momentálne podporované *mixed* (zložené), *time* (časové) a *memory* (pamätové) profily) a *units* mapuje využitú metrickú jednotku k danému typu zdroja. Kľúč *cmd* určuje, pre aký príkaz/skript/binárny

[1] knižnica pre bezstratovú kompresiu dát — https://zlib.net/
[2] JavaScript Object Notation — https://www.json.org/

súbor tento profil vznikol. Kľúče *params* a *workload* sú voliteľné a obsahujú dodatočné špecifikácie príkazu uvedeného v *cmd*. Pri automatizovanom generovaní sú profily vytvárané pre každú kombináciu *cmd*, *params* a *workload* špecifikovanú v konfigurácii projektu.

Collector-info. Uchováva informácie o zberači dát, ktorý daný profil vytvoril. Obsahuje identifikáciu zberača v kľúči *name* a jeho bližšie nastavenia v kľúči *params*.

Postprocessors. V tomto regióne sa nachádza zoradený zoznam postprocesorových jednotiek, ktoré boli pri spracovaní profilu využité. Poradie, v ktorom profil spracovávali, je v tomto zozname zachované. Zoznam môže byť v priebehu ďalšej fázy spracovania aktualizovaný o nové postprocesory.

Chunks. Jedná sa o nepovinný región, ktorý je momentálne vo vývoji. Má predstavovať vyhľadávaciu tabuľku, ktorá bude mapovať jednoznačné identifikátory na väčšie časti *JSON* regiónov. V profiloch sa totiž často nachádzajú duplicitné informácie — nahradením duplicitného regiónu odkazom do vyhľadávacej tabuľky sa potom značne zníži veľkosť profilu.

```
{
      "origin": "",
      "header": {},
      "collector_info": {},
      "postprocessors": [],
      "snapshots": [],
      "chunks": {}
}
```

Obrázok 3.3: Hlavná schéma regiónov jednotného formátu pre ukladanie profilovacích dát a informácií o priebehu procesu profilácie. Zdroj: [4].

```
"header": {
      "type": "time",
      "units": {
            "time": "s"
      },
      "cmd": "perun",
      "params": "status",
      "workload": "--short",
},
```

Obrázok 3.4: Schéma kľúčov obsiahnutých v regióne *header* pre profil príkazu `perun status --short`. Zdroj: [4].

Snapshots. Obsahuje zberačom zachytené profilovacie dáta, ktoré sú usporiadané do zoznamu snímok[3] zaznamenaných zdrojov. Každý záznam v *snapshots* obsahuje nasledujúce infomácie:

- **time** — špecifikuje časovú pečiatku zachytenia daného záznamu.
- **resources** — obsahuje zoznam zachytených profilovacích dát, pričom položky a jednotky zachytených hodnôt sa menia v závislosti od zbernej jednotky, ktorá daný výkonnostný profil vytvorila. Kľúč *type*, prípadne *subtype*, informuje o jednotke konkrétnych zachytených dát a kľúč *amount* udáva ich množstvo.
- **models** — obsahuje zoznam modelov získaných pomocou regresnej analýzy [14]. Modely slúžia pre funkčné vyjadrenie veľkosti premennej v závislosti na ostatných premenných, napríklad model doby behu v závislosti na veľkosti vstupu.

3.4 Automatizácia zberu profilovacích dát

Zber profilovacích dát je za pomoci nástroja *Perun* možné realizovať manuálne alebo automatizovane. Pri manuálnej profilácii užívateľ konfiguruje proces pomocou parametrov

[3] Snímkou rozumieme pevný časový interval vykonanej profilácie

(*cmd*, *args*, *workload*, *params*) príkazu `perun collect`, ktorý je volaný pre konkrétny zberač dát. V tomto prípade nie je výsledný profil už nijak spracovávaný — spracovanie je však možné vykonať ručne pomocou príkazu `perun postprocessby`. *Perun* ďalej umožňuje automatický zber profilovacích dát na základe vopred uloženej lokálnej alebo zdieľanej konfigurácie — tento spôsob je využívaný najmä pri pravidelnej výkonnostnej analýze verzií projektu a môže tak byť súčasťou napr. *Git hooks* alebo priebežnej integrácie. Jadrom automatizácie v nástroji *Perun* sú matice úloh (tzv. *job matrices*), ktoré sú určené množinou jednotiek *collectorov*, *postprocessorov* a vstupných dát [14]. Užívateľ má možnosť definovať vlastné matice v súbore lokálnych nastavení (`local.yml`, viď sekcia 3.5), a tak celý proces profilácie zhrnúť to jedného príkazu — `perun run matrix`. Formát zápisu matice je zobrazený na obrázku 3.5. V prípade potreby nepravidelného alebo špecifického vytvorenia výkonnostných profilov *Perun* ponúka možnosť definovať špecifikáciu pre samostatnú úlohu (tzv. *single job specification*) pomocou príkazu `perun run job`.

```
cmds:
   - perun

args:
   - log
   - log --short

workloads:
   - HEAD
   - HEAD~1

collectors:
   - name: time

postprocessors:
   - name: normalizer
   - name: regression_analysis
   - params:
        - method: full
        - steps: 10
```

Obrázok 3.5: Príklad formátu pre definíciu matice úloh. Zdroj: [4].

3.5 Spôsob uloženia dát

Adresárová štruktúra instancií nástroja *Perun* a vnútorné úložisko dát sú založené na vnútornej organizácii systému *Git*. Základnú schému adresárovej štruktúry ilustruje obrázok 3.6.

```
.perun/
    |-- /jobs
    |-- /objects
    |-- local.yml
```

Obrázok 3.6: Schéma adresárovej štruktúry nástroja Perun. Každá instancia obsahuje adresár nepriradených profilov, adresár objektov a lokálnu konfiguráciu. Zdroj: [4].

jobs. Adresár obsahujúci výkonnostné profily, ktoré ešte neboli priradené ku konkrétnym verziám projektu. Tieto profily obsahujú región *origin* opísaný v kapitole 3.3.

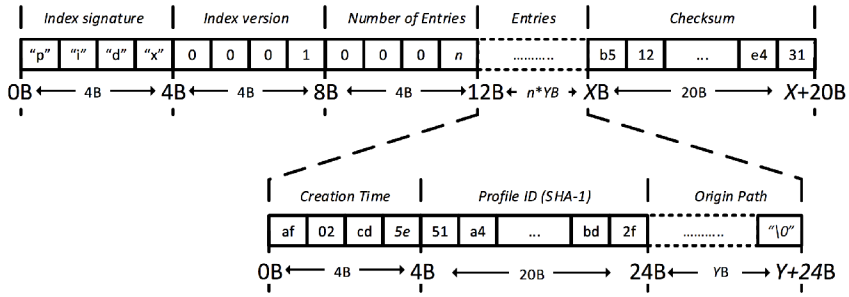
objects. Je hlavným dátovým úložiskom, obsahuje objektové primitíva. Každý objekt je reprezentovaný jednoznačným identifikátorom (vo forme SHA-1) a môže byť jedným z troch typov:

- **objekt typu *blob*** — obsahuje komprimované profilovacie dáta. Špecifikácia tohto typu bude bližšie popísaná v podkapitole 3.5.2.
- **objekt typu index** — prináleží jednej konkrétnej verzii projektu. Tento typ bude bližšie opísaný v podkapitole 3.5.1.
- **objekt typu changes** — obsahuje zoznam nájdených zmien výkonu.

local.yml. Konfiguračný súbor v *Yaml*[4] formáte uložený v adresári `.perun`. Obsahuje lokálne nastavenia, tzn. nastavenia pre danú konkrétnu instanciu *Perunu*, napr. špecifikáciu obaleného verziovacieho systému alebo maticu úloh (viď sekcia 3.4), pre jeden konkrétny projekt.

3.5.1 Špecifikácia objektu typu index

Pre každú verziu projektu (napríklad pre každý *commit* verziovacieho systému *Git*), ku ktorej bol priradený aspoň jeden profil, sa v zložke `./perun/objects` nachádza indexový súbor v binárnom formáte. Na obrázku 3.7 je zobrazený význam jednotlivých bajtov, z ktorých sa súbor skladá.



Obrázok 3.7: Schéma významu bajtov indexového súboru. Zdroj: [4].

Index signature. Obsahuje reťazec „pidx“ (perun index), ktorý slúži na rýchlu identifikáciu, že sa jedná o objekt indexového typu.

Index version. Špecifikácia verzie indexu zavedená kvôli možnosti spätnej kompatibility v prípade zmeny špecifikácie indexu.

Number of entries. Celočíselný počet záznamov nachádzajúcich sa v indexe.

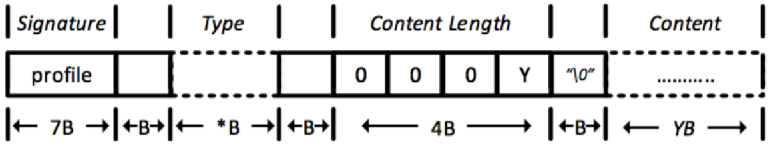
Entries. Každý záznam v indexe je premennej dĺžky a prináleží práve jednému profilu priradenému k danej verzii projektu. Záznam obsahuje časovú pečiatku (*Creation time*) určujúcu čas vytvorenia, identifikáciu konkrétneho komprimovaného objektu s profilovacími dátami (*Profile ID*) a cestu (*Origin path*) k danému objektu reprezentovanú ASCII reťazcom premennej dĺžky, ktorý je zakončený nulovým znakom `\0`.

Checksum. Kontrolný súčet indexu slúžiaci na detekciu chýb.

[4]YAML Ain't Markup Language — http://yaml.org/

3.5.2 Špecifikácia objektu typu *blob*

Každý objekt typu *blob* pozostáva z tela profilovacích dát a krátkej hlavičky zakončenej nulovým znakom `\0`. Z obsahu hlavičky a profilovacích dát je najprv vypočítaný kontrolný súčet (SHA-1), ktorý slúži ako *Profil ID*, teda identifikácia objektu, ktorej využitie bolo opísané v kapitole 3.5.1. Objekt je následne komprimovaný pomocou metódy *zlib* a uložený v `.perun/objects`.



Obrázok 3.8: Schéma významu bajtov objektu typu *blob*. Zdroj: [4].

Signature. Jedná sa o prefix obsahujúci ASCII reťazec „profile“, slúži na rýchlu identifikáciu, že sa jedná o objekt typu *blob* obsahujúci profilovacie dáta.

Type. ASCII reťazec premennej dĺžky špecifikujúci typ profilu, slúži pre zjednodušenie spracovania profilov.

Content Length. Veľkosť obsahu dát nachádzajúcich sa za nulovým znakom `\0` uvedená v bajtoch.

Content. Nespracovaný obsah výkonnostného profilu vo formáte bližšie opísanom v kapitole 3.3.

Kapitola 4

Dostupné technológie pre tvorbu webových aplikácií

Výsledné rozhranie pre správu výkonnostných profilov je požadované vo forme webovej aplikácie s dôrazom na prehľadnosť a jednoduchosť použitia. Pre splnenie týchto požiadaviek bolo preštudovaných niekoľko voľne dostupných JavaScriptvých frameworkov a knižníc určených pre vytváranie webových aplikácií a rozhraní. Konkrétne sú to AngularJS, React a Vue.js. Následuje stručný popis ich architektúr, implementačných možností a vlastností predstavujúcich výhody či nevýhody, ktoré by prinášal výber danej technológie.

4.1 Jednostránkové aplikácie

Jednou z hlavných špecialít frameworkov a knižníc opísaných v tejto kapitole je tvorba jednostránkových aplikácií. Jednostránková aplikácia (*single page application* — SPA) je webová aplikácia, ktorá pozostáva z jedinej HTML stránky, ktorej obsah je dynamicky aktualizovaný na základe interakcie s užívateľom. Stránka je týmto načítaná len raz (pokiaľ opätovné načítanie nie je vynútená manuálne) [9].

Navigácia v rámci aplikácie je realizovaná za pomoci tzv. „hash-based“ smerovania (*hash-based routing*), pri ktorom je smerovanie vykonávané za pomoci časti adresy URL nachádzajúcej sa za znakom `#` — časť, ktorá nie je odosielaná na server a slúži na identifikáciu aktuálneho umiestnenia v rámci webovej aplikácie. Pri jednostránkovej aplikácii je miesto načítavania novej stránky zo serveru pri každej užívateľskej akcii využívaná asynchrónna komunikácia so serverom — pre tento účel sa najčastejšie využíva technológia Ajax[1].

Hlavným cieľom jednostránkových aplikácií je poskynúť užívateľovi skúsenosť neprerušovanú neustálym načítaním nových stránok zo serverovej strany — pokúšajú sa teda priblížiť plynulosti desktopových aplikácií a zároveň sa vyhnúť ich závislosti na operačnom systéme.

4.2 AngularJS

AngularJS je otvorený JavaScriptový framework prvýkrát predstavený spoločnosťou Google v roku 2012, zameriava sa na tvorbu jednostránkových dynamických webových aplikácií. Je založený na softvérovej architektúre Model-View-Controller (MVC), ktorá delí rôzne

[1] Asynchronous JavaScript and XML

aspekty aplikácie do vrstiev model, pohľad a kontroler s cieľom rozdelenia zodpovednosti a sprehľadnenia kódu [13].

AngularJS umožňuje pridávať do HTML značiek špeciálne formátovacie atribúty, ktoré sa nazývajú direktívy. Tieto atribúty diktujú aplikácii spôsob väzby dát medzi modelom a vstupnými-výstupnými časťami pohľadu, pričom model je reprezentovaný štandardnými JavaScriptovými premennými [2]. Jedná sa o deklaratívny spôsob implementácie bez nutnosti popisovať, ako majú byť dáta synchronizované. Vzťahy stačí deklarovať a synchronizácia je automaticky zaistená. V prípade AngularJS ide o dvojsmerný tok dát, teda pokiaľ nastane zmena v dátach modelu, pohľad je patrične upravený, a naopak. Výhodou takejto väzby je zabezpečenie automatickej aktualizácie obsahu, naopak nevýhodou sú problémy s výkonnosťou a väčšie nároky na výpočtové prostriedky pri rozhraní s veľkým počtom prvkov.

AngularJS podporuje koncept oddelenia zodpovedností[2] využívaním špeciálnych nahraditeľných objektov nazývaných služby, ktoré sú navzájom poprepájané pomocou vkladania závislostí[3]. Popri tých službách, ktoré AngularJS obsahuje implicitne, je taktiež možné si službu užívateľsky nadefinovať. Keďže každá služba vykonáva len jednu špecifickú úlohu, je jednoduché ju udržiavať a testovať jej správnu činnosť. Za pomoci služieb AngularJS ponúka možnosť ponechať kontroleru len základnú logiku potrebnú pre daný pohľad, teda zvyšná funkcionalita je presunutá do služieb, ktoré sú kontroleru odovzdané ako závislosti.

V roku 2016 bol predstavený framework Angular [1], ktorý sa od AngularJS odlišuje v mnohých aspektoch a nie je s ním spätne kompatibliný. Vývoj týchto dvoch frameworkov prebieha samostatne.

Výhody:

- prehľadný a štrukturovaný kód,
- obojsmerná väzba dát zaisťujúca automatickú aktualizáciu obsahu,
- vstavaný systém vkladania závislostí,
- možnosť definície vlastných služieb a direktív.

Nevýhody:

- robustnosť a komplikovanosť,
- pokročilá funkcionalita je náročená na osvojenie.

Napriek výhodám, ktoré AngularJS ponúka, je v prípade menších projektov odradzujúca práve jeho celková robustnosť. Cieľom tejto práce je vytvoriť odľahčené rozhranie pre jednoduché budúce rozširovanie, a preto sa AngularJS nejaví byť vhodný pre tento projekt.

4.3 React

React je otvorená JavaScriptová knižnica pôvodne vytvorená a vyvíjaná pre interné účely spoločnosti Facebook. Na rozdiel od AngularJS zastupuje len pohľadovú vrstvu (*view*) softvérovej architektúry MVC, je určená výlučne pre vytváranie responzívnych užívateľských

[2] *Separation of Concerns* označuje rozdelenie programu tak, aby sa jeho časti z hľadiska funkcionality čo najmenej prekrývali

[3] *Dependency Injection* je návrhový vzor, pri ktorom trieda nevytvára závislosti, ale nadobúda ich od externých zdrojov

rozhraní. Poskytuje možnosť definície vlastných, znovu využitených komponentov, ktoré sú ľahko manipulovateľné, pričom je možné ich navzájom spájať a zanárať. Tento deklaratívny charakter Reactu umožňuje jednoducho popísať, ako by mal komponent vo výsledku vyzerať, miesto manuálneho vyhľadávania a upravovania uzlov objektového modelu dokumentu (tzv. Document Object Model — DOM) [5].

React interne pracuje s virtuálnou reprezentáciou pôvodného DOM, tzv. virtuálnym DOM. Jedná sa o abstrakciu odľahčenú od detailov konkrétnej implementácie internetového prehliadača, kde pri zmene stavu komponentu je daná zmena aplikovaná vo virtuálnom DOM, miesto toho pôvodného. Následne sú oba modely porovnávané a po nájdení rozdielu sa vyhodnocuje najmenšia množina zmien potrebná pre udržanie aktuality pôvodného DOM. Touto cestou sa React dokáže vyhnúť výkonnostne náročným operáciam s DOM a zrýchliť tak beh samotnej aplikácie.

Okrem štandardnej JavaScriptovej syntaxe React využíva špeciálnu nadstavbu, podobnú XML, nazývanú JavaScript XML (JSX), ktorá umožňuje vkladanie HTML do JavaScriptového kódu. Tento spôsob tvorenia kódu je preferovaný oproti klasickému JavaScriptovému formátu aj napriek tomu, že je potrebné daný kód pred spustením preložiť za pomoci špeciálneho prekladača, keďže sa nejedná o valídnu JavaScriptovú syntax. Prekladač tohoto typu sa nazýva *transpiler* a zaisťuje preklad vstupného zdrojového kódu v jednom jazyku na zdrojový kód iného jazyka. Pôvodne bol pre tento účel využívaný *JSTransform*[4], v roku 2015 bol nahradený[5] prekladačom *Babel*[6].

Výhody:

- virtuálny DOM a rýchlosť aktualizácií obsahu,
- komponenty a ich znovuvyužiteľnosť,
- možnosť kombinácie JavaScriptu s HTML pomocou JSX,
- celková jednoduchosť a odľahčenosť.

Nevýhody:

- čerstvé zmeny v licencovaní a neistota v smere vývinu situácie,
- nutnosť zahŕňať ďalšie knižnice pre pokrytie ostatných častí aplikácie.

React bol do októbra 2017 licencovaný pod BSD+Patents licenciou nevhodnou pre otvorené projekty pod záštitou spoločností. Neskôr došlo k zmene licencie na licenciu MIT, ktorá poskytuje väčšiu voľnosť použitia. Z dôvodu zmien a nejasnej budúcnosti ohľadom licencovania tejto knižnice bolo upustené od jej použitia v projekte, vzhľadom na to, že nástroj *Perun* je cielený na úplne otvorenú komunitu.

4.4 Vue.js

Vue je nezávislý otvorený projekt licencovaný pod MIT licenciou. Jedná sa o progresívny Model-View-View-Model (MVVM) JavaScriptový framework určený pre budovanie užívateľských rozhraní jednostránkových webových aplikácií. Vznikol v roku 2014 a momentálne

[4] https://github.com/facebookarchive/jstransform
[5] https://reactjs.org/blog/2015/06/12/deprecating-jstransform-and-react-tools.html
[6] http://babeljs.io/

je jedným z najrýchlejšie sa vyvíjajúcich JavaScriptových frameworkov, kombinuje v sebe technológie pôvodne predstavené v AngularJS a Reacte.

Vue, rovnako ako React, pracuje s konceptom virtuálneho DOM, a taktiež je založený na komponentoch, ktoré umožňujú ročleniť rozhranie do menších jednotiek. Vo Vue má každá instancia komponentu svoj vlastný uzavretý rámec; obsahuje šablónu (*template*) špecifikujúcu pohľad, má vlastný vnútorný stav určený dátami obsiahnutými vo funkcií `data` a v objekte `methods` obsahuje definície operácií, ktoré určujú spôsoby, akými môže byť vnútorný stav menený na základe užívateľského vstupu. Komponent môže taktiež byť doplnený o vlastné definície CSS pravidiel upravujúce štýl.

Ako väčšina JavaScriptových frameworkov aj Vue využíva renderovanie obsahu na strane klienta (*client-side rendering* — CSR), čo kladie na klientsky prehliadač značnú záťaž. Nevýhodou tohto spôsobu je nutnosť počkať na dokončenie sťahovania a vykonanie JavaScriptového kódu, aby mohol byť obsah zobrazený. Podobne ako React, Vue okrem CSR podporuje renderovanie na strane servera (*server-side rendering* — SSR), čo značne skracuje dobu načítavania stránky a znižuje záťaž kladenú na prehliadač. K tomuto účelu je využívaná knižnica Nuxt.js.

Vue využíva direktívy, ktorých syntax je podobná syntaxi direktív AngularJS (napríklad direktíva v-if vo Vue je ekvivalentná ng-if v AngularJS). Direktívy vo Vue začínajú sprefixom „v“ a slúžia striktne na zapúzdrenie manipulácie s DOM. Podobne ako AngularJS aj Vue dokáže pracovať s konceptom obojsmernej väzby dát (za pomoci direktívy v-model), a tým zabezpečovať automatickú aktualizáciu obsahu. Medzi jednotlivými komponentmi Vue presadzuje jednosmerný tok dát [7].

Keďže je Vue nezávislým projektom, na rozdiel od AngularJS a Reactu nemá podporu veľkej spoločnosti s pevným zázemím. Vývojárska komunita Vue je v porovnaní so staršími frameworkami malá, ale veľmi rýchlo sa rozrastá.

Výhody:

- virtuálny DOM a rýchlosť aktualizácií obsahu,
- možnosť obojsmernej väzby dát zaisťujúcej automatickú aktualizáciu obsahu,
- komponenty a ich znovuvyužiteľnosť,
- direktívy zapuzdrujúce prácu s DOM,
- nenáročný na zvládnutie za krátky čas,
- detailná a zrozumiteľná dokumentácia.

Nevýhody:

- nezávislý projekt bez podpory veľkej spoločnosti s pevným zázemím.

Vue je framework, ktorý z AngularJS a Reactu preberá tie najlepšie vlastnosti, a zároveň sa vo veľkej miere úspešne vyhýba nedostatkom, ktoré tieto dve technológie obsahujú. Po zvážení všetkých výhod a nevýhod bol framework Vue zvolený pre implementáciu rozhrania pre správu výkonnostných profilov.

Kapitola 5

Analýza a špecifikácia GUI pre správu výkonnostných profilov

Analýza požiadaviek na systém (*requirements engineering*) je proces vytvárania špecifikácie software na základe požiadaviek zadávateľa. Tieto požiadavky odrážajú potreby zadávateľa, vymedzujú funkcionalitu systému a zároveň určujú služby, ktoré by výsledný systém mal poskytovať. V závislosti od cieleného čitateľa a informácií, ktoré majú požiadavky obsahovať, je nutné vytvoriť ich popis v rôznych úrovniach detailnosti — podľa využitej úrovne abstrakcie teda požiadavky delíme na užívateľské[1] a systémové[2] [16]. Požiadavky na systém sa podľa vecnosti môžu ďalej deliť na:

- **Funkčné požiadavky** — sú popisom služieb, ktoré by mal systém poskytovať, reakcií systému na určité vstupy a správania systému v konkrétnych situáciách. Popisujú „čo“ má systém umožnovať.
- **Mimofunkčné požiadavky** — obvykle určujú či obmedzujú vlastnosti systému ako celku, nesústredujú sa na individuálne funkcie a služby. Vytvorenie jednej mimofunkčnej požiadavky môže zapríčiniť vznik niekoľkých funkčných požiadaviek. Popisujú „ako“ sa systém má správať.

5.1 Funkčné požiadavky

Hlavným zdrojom funkčných požiadaviek boli konzultácie so zadávateľom a autorom projektu *Perun*. Niektoré požiadavky vyplynuli z preštudovania a analýzy textového rozhrania (CLI), ktorým súčasná verzia *Perunu* disponuje — boli vybrané najmä úkony, pri ktorých môže práca s grafickým rozhraním byť pohodlnejšia a prehľadnejšia, než pri klasickom textovom rozhraní. Hlavným zdrojom informácií o možnostiach textového rozhrania nástroja *Perun* bola jeho dokumentácia [4]. Ako ďalšia inšpirácia pre niektoré požiadavky taktiež poslúžili grafické rozhrania existujúcich nástrojov bližšie analyzované v sekcii 2.1.1. Z funkcionálneho hľadiska by výsledné rozhranie pre správu výkonnostných profilov malo spĺňať nasledujúce požiadavky:

[1] Výroky v prirodzenom jazyku alebo diagramoch, opisujú úkony, ktoré má užívateľ byť schopný so systémom vykonávať.

[2] Obsahujú detailnejší popis systémovej funkcionality, služieb a obmedzení, ktoré majú byť implementované.

Sec.1 Správa systému *Perun*. Pre umožnenie automatizácie vytvárania výkonnostných profilov a ich správy je nutné nástroj *Perun* integrovať do používaného systému pre správu verzií (viď sekcia 3.4). Medzi požiadavky na správu integrácie v rámci grafického rozhrania patria tieto možnosti:

P.1 Rozhranie zobrazuje zoznam všetkých systémov pre správu verzií, ktoré sú dostupné v užívateľom zadanom adresári a jeho podadresároch, a zároveň sú podporované nástrojom *Perun*.

P.2 Pri každom dostupnom verziovacom systéme je zobrazený údaj o stave integrácie nástroja *Perun*.

P.3 Rozhranie umožňuje integrovať nástroj *Perun* do vybraného verziovacieho systému, nad ktorým ešte nebola vytvorená obaľujúca *Perun* instancia. Inicializácia zahŕňa:

- obalenie verziovacieho systému instanciou *Perunu*,
- možnosť inicializácie konfigurácie pre automatický zber dát (tzv. matice úloh obsiahnutej v lokálnych nastaveniach),
- možnosť vloženia *hooks* — úloh, ktoré sú automaticky spúšťané po ukončení určitej udalosti (napr. vytvorenie novej verzie projektu). Táto funkcionalita momentálne v nástroji nie je podporovaná, ale jej doplnenie sa chystá v blízkej budúcnosti.

Sec.2 Správa automatizácie procesu profilácie. Automatizácia procesu profilácie je v nástroji *Perun* realizovaná za pomoci matíc úloh (viď sekcia 3.4). V textovom rozhraní sú definícia a editovanie matíc zapísaných v lokálnej konfigurácii vykonávané pomocou predvoleného textového editora v termináli; jedná sa teda práve o činnosť, ktorá by mohla byť pohodlnejšia pri použití grafického rozhrania. Medzi požadované možnosti patria:

P.1 Rozhranie zobrazuje aktuálne nastavenia jednotlivých regiónov matice úloh.

P.2 Pri každom zobrazenom regióne je možné pridať/nadefinovať/odobrať položku.

P.3 Rozhranie umožňuje automaticky spustiť proces profilovania podľa zadanej matice úloh.

Sec.3 Správa výkonnostných profilov verzií. Hlavným cieľom grafického rozhrania je sprehľadnenie správy výkonnostných profilov. Jednotlivé profily je potrebné zobrazovať v kontexte konkrétnej verzie projektu, ku ktorej boli priradené, a zároveň je danú verziu potrebné zobrazovať v kontexte projektu, v rámci ktorého vznikla — ako dôsledok boli definované samostatné požiadavky na zobrazovanie informácií o projektoch a ich verziách, ktoré budú popísané neskôr v kapitole. Medzi požiadavky zamerané priamo na správu výkonnostných profilov v rámci grafického rozhrania patria tieto možnosti:

P.1 Rozhranie zobrazuje zoznam všetkých výkonnostných profilov — ako priradených ku vybranej verzii projektu, tak aj doposiaľ nepriradených (spoločne s ich pôvodnou špecifikáciou verzie, ku ktorej prináležia).

P.2 Pri každom výkonnostnom profile sú zobrazené údaje o type (*type*), príkaze (*cmd*), záťaži (*workload*), argumentoch (*args*), využitom zberači (*collector*) a čase vytvorenia (*time*). Význam týchto údajov bol bližie popísaný v sekcii 3.3.

P.3 Pri zozname výkonnostných profilov je možné dynamicky vyberať, ktoré z údajov majú byť zobrazené a ktoré majú byť skryté.

P.4 Pre konkrétny výkonnostný profil umožňuje rozhranie zobraziť zachytené profilovacie dáta a ďalšie dodatočné informácie.

P.5 Rozhranie umožňuje vytvorené výkonnostné profily spracovať pomocou jednotiek *postprocessorov*, ktoré si užívateľ sám vyberie.

P.6 Rozhranie umožňuje priradiť doposiaľ nepriradený výkonnostný profil k zodpovedajúcej verzii, a taktiež priradený profil od verzie odobrať.

Sec.4 Zobrazovanie informácií o projektoch. Požiadavka vyplýva z potreby zobrazovať jednotlivé verzie v kontexte projektu, v rámci ktorého vznikli — v grafickom rozhraní je preto potrebné okrem zoznamu projektov dostupných v systéme zobrazovať aj základné informácie o nich. Rozhranie má spĺňať nasledovné:

P.1 Pri každom projekte je zobrazený údaj o type verziovacieho systému a stave integrácie nástroja *Perun*.

P.2 V zobrazených projektoch je možné vyhľadávať podľa základných pravidiel.

P.3 Rozhranie umožňuje meniť poradie v stĺpcoch jednotlivých údajov o projekte zostupne alebo vzostupne.

Sec.5 Zobrazovanie informácií o verziách projektu. Požiadavka vyplýva z potreby zobrazovať jednotlivé výkonnostné profily v kontexte konkrétnej verzie projektu, ku ktorej prináležia. Medzi možnosti, ktoré je potrebné v grafickom rozhraní podporovať, patria:

P.1 Rozhranie zobrazuje celú históriu verzií konkrétneho projektu. V prípade existencie viacerých vetiev vývoja (*major* verzií) umožňuje prepínanie kontextu medzi vetvami.

P.2 Pri každej *minor* verzii projektu sa nachádzajú údaje o textovom popise zmien od autora, dátume pridania verzie a stave degradácie výkonu oproti predošlým verziám.

P.3 Rozhranie umožňuje meniť poradie údajov verzií obsiahnutých v jednotlivých stĺpcoch zostupne alebo vzostupne.

P.4 Rozhranie obsahuje možnosť filtrovania obsahu samostatne pre každý údaj.

P.5 Rozhranie zobrazuje informácie o stave výkonu danej verzie programu vo forme počtov nových degradácií a optimalizácií.

Sec.6 Interpretácia výkonnostných profilov Výkonnostné profily je potrebné vhodne interpretovať za pomoci rôznych vizualizačných techník. Medzi možnosti, ktoré má grafické užívateľské prostredie podporovať, patria:

P.1 Základné vizualizácie pomocou stĺpcových grafov podľa zavedenej konfigurácie.

P.2 Základné vizualizácie pomocou čiarových grafov podľa zavedenej konfigurácie.

P.3 Základné vizualizácie pomocou korelačných grafov podľa zavedenej konfigurácie.

Sec.7 Správa konfigurácie. Nástroj *Perun* využíva dva typy konfigurácie — lokálnu a globálnu. V prípade, že lokálna konfigurácia aktuálnej instancie *Perun* neobsahuje vyhľadávaný kľúč, je tento kľúč postupne vyhľadávaný v konfiguráciach nadradených instancií, pričom vyhľadávanie môže skončiť až v globálnej konfigurácii. Medzi požiadavky zamerané na konfiguráciu nastavení v rámci grafického rozhrania patria nasledujúce možnosti:

P.1 Rozhranie umožňuje zobrazovať a upravovať základné lokálne nastavenia týkajúce sa konkrétneho adresára `.perun`.

P.2 Rozhranie zobrazuje v prípade absencie vyhľadávaného kľúča v lokálnych nastaveniach najbližší nadradený korešpondujúci kľúč.

P.3 Rozhranie umožňuje zobrazovať a upravovať globálne nastavenia, ktoré môžu ovplyvniť viacero instacnií *Perunu*.

5.2 Mimofunkčné požiadavky

Hlavným zdrojom mimofunkčných požiadaviek na grafické užívateľské rozhranie boli rovnako ako pri funkčných požiadavkách konzultácie s udržovateľom projektu *Perun*/zadávateľom práce. Niektoré požiadavky vyplynuli priamo zo zadania práce. Výsledný zoznam mimofunkčných požiadaviek na grafické rozhranie pre správu výkonnostných profilov je nasledovný:

- **Webové užívateľské rozhranie**. Rozhranie má byť vytvorené za pomoci webových technológií, s možnosťou jeho nasadenia na serverový systém.
- **Jednostránková aplikácia**. Rozhranie má byť vo forme jednostránkovej aplikácie pre dosiahnutie plynulej užívateľskej skúsenosti.
- **Multiplatformné rozhranie zamerané na desktopové systémy**. Výsledné grafické rozhranie bude pracovať nezávisle na operačnom systéme.
- **Jednoduché nasadenie**. Výsledné grafické rozhranie by malo byť ľahko distribuovateľné a pripravené k nasadeniu do priestoru *cloud*.

Kapitola 6

Návrh grafického rozhrania

Pod vizualizáciou možno rozumieť grafickú reprezentáciu dát; jej hlavným účelom je vziať komplexnú informáciu a interpretovať ju čo najjednoduchším a najpútavejším spôsobom — prinajlepšom tak, aby pre porozumenie stačil jediný pohľad. Pri grafických užívateľských rozhraniach nestačí len vytvoriť vhodnú vizualizáciu, je taktiež potrebné tieto vizualizácie vhodne prepojiť. Jedným z najčastejšie sa opakujúcich problémov pri užívateľských rozhraniach je množstvo akcií, ktoré je potrebné vykonať pre dosiahnutie cieľovej lokality.

Weboví dizajnéri a profesionáli zaoberajúci sa využiteľnosťou strávili veľa času polemizovaním o tom, koľkokrát sú používatelia ochotní „kliknúť“ na ceste za cieľom bez toho, aby boli príliš frustrovaní [12].

Dobre navrhnuté užívateľské rozhranie teda nie len zrozumiteľne interpretuje komplexné dáta, ale taktiež umožňuje užívateľovi vykonávať zamýšľané akcie efektívne a intuitívne. Takýmto rozhraním oplýva napr. nástroj *Codacy*, ktorý bol bližšie popísaný v sekcií 2.1.3 — rozhranie tohto nástroja poslúžilo ako hlavná inšpirácia pri vytváraní návrhu.

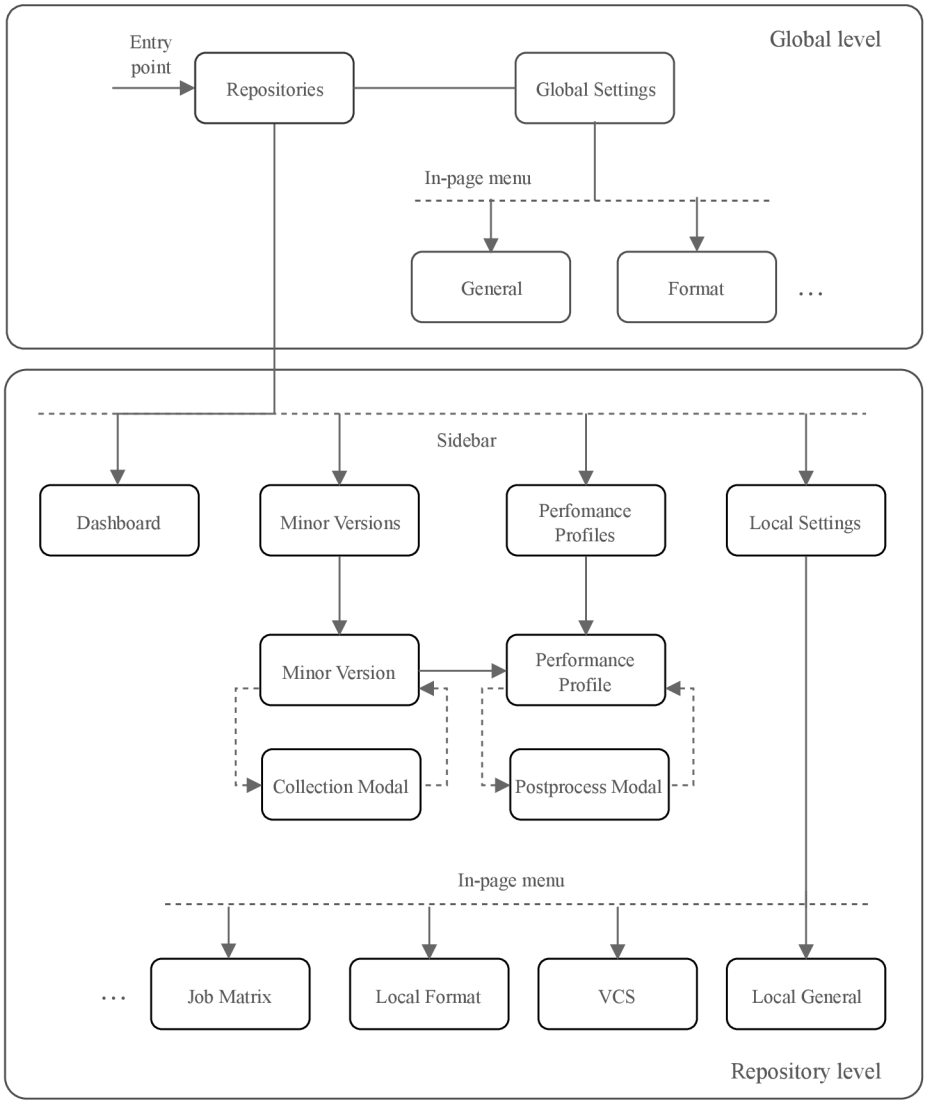
6.1 Architektúra klientskej časti

V kapitole 5 boli popísané základné požiadavky, ktoré má vytvárané grafické užívateľské rozhranie spĺňať. Špecifikované požiadavky na funkcionalitu boli na základe spoločných čŕt rozdelené do niekoľkých skupín zodpovedajúcich komponentom a ich podčastiam, z ktorých sa vo výsledku skladá architektúra návrhu rozhrania. Ilustrácia návrhu hlavnej logiky medzi jednotlivými komponentmi sa nachádza na obrázku 6.1 (pre jednoduchosť je možnosť pohybu medzi jednotlivými komponentmi znázornená len v smere od najvyššej úrovne k najhlbšiemu zanoreniu). Výsledný návrh obsahuje nasledujúce komponenty:

Repozitáre (*Repositories*). Jedná sa o hlavný vstupný bod rozhrania (*entry point*) — časť repozitárov, ktorá obsahuje zoznam vyhľadaných existujúcich verziovacích systémov a informáciu o stave ich integrácie. Spĺňa požiadavky obsiahnuté v sekciách **Sec.1** a **Sec.4**.

Globálne nastavenia (*Global Settings*). Na rovnakej úrovni ako komponent repozitárov sa taktiež nachádza časť globálnych (zdieľaných) nastavení, ktorá obsahuje podsekcie — jednu podsekciu pre každú časť (napr. *general* pre všeobecné nastavenia, *format* pre nastavenie výstupu logov či generovaných súborov) nachádzajúcu sa v nastaveniach zdie-

ľaných všetkými instanciami *Perunu* v súbore `shared.yml`. Spĺňa požiadavku **P.3** sekcie **Sec.7**.



Obrázok 6.1: Schéma logiky jednotlivých komponentov architektúry a ich prepojenia. Schéma je rozdelená do dvoch častí — globálnej (nezávislej na konkrétnej instancii repozitára) a repozitárovej. Plná šípka s vyznačením smeru vyjadruje možnosť prechodu z kontextu jedného kompotentu do druhého, prerušovaná čiara značí menu umožňujúce prechod do viacerých komponentov.

Nástenka (*Dashboard*). Po výbere jedného z ponúkaných verziovacích systémov v časti repozitárov je pohľad prenesený do kontextu vybraného systému — vstupným bodom je nástenka obsahujúca základné informácie o stave systému. Kontext je potom medzi ďalšími podčasťami repozitára možné prepínať za pomoci postranného menu. Jedná s o rozširujúcu funkcionalitu, ktorá nie je obsiahnutá v požiadavkach.

***Minor* verzie (*Minor Versions*).** Časť tzv. *minor* verzií (spomínaných v sekcii 2.1.1) predstavuje zoznam všetkých verzií (napr. *commitov*) prináležiacich *major* verziám da-

ného projektu (tzn. všetkým vetvám v kontexte verziovacieho systému). Obsahuje taktiež základné informácie o *minor* verziách, o vykonaných zmenách a o prináležiacich výkonnostných profiloch. Spĺňa požiadavky sekcie **Sec.5**.

Výkonnostné profily (*Performance Profiles*). Komponent výkonnostných profilov predstavujúci zoznam všetkých profilov prináležiacich danému projektu, roztriedených na už registrované a doposiaľ nepriradené profily a ich základný popis. Spĺňa požiadavky **P.1**, **P.2**, **P.3** a **P.6** sekcie **Sec.3**.

Lokálne nastavenia (*Local Settings*). Lokálne nastavenia repozitára, podobne ako globálne, obsahujú niekoľko podsekcií (napr. *general*, *format*, *vcs*) zodpovedajúcich obsahu súboru `local.yml` (viď sekcia 3.5), pričom podsekcie *cmd*, *args*, *workloads*, *collectors* a *postprocessors*, slúžiace pre špecifikáciu matice úloh (viď sekcia 3.4), sú združené do spoločnej časti *Job Matrix*. Tento komponent spĺňa všetky požiadavky sekcie **Sec.2** a požiadavky **P.1** a **P.2** sekcie **Sec.7**.

***Minor* verzia (*Minor Version*).** Informácie o konkrétnej *minor* verzii a jej prináležiacich (priradených aj doposiaľ nepriradených) výkonnostných profiloch, spĺňa požiadavky **P.1**, **P.2**, **P.3** a **P.6** sekcie **Sec.3**. Táto časť taktiež umožňuje prepnutie kontextu do modálneho okna, za pomoci ktorého bude realizovaná špecifikácia pre samostatnú profilovaciu úlohu (viď sekcia 3.4) a jej spustenie, pričom jej výsledkom bude potom sada novo vytvorených výkonnostných profilov. Jedná sa o prípravu pre budúce rozšírenie funkcionality, ktoré nie je obsiahnuté v požiadavkach.

Výkonnostný profil (*Performance Profile*). Informácie o jednom konkrétnom výkonnostnom profile. K tomuto komponentu je možné pristúpiť buď zo zoznamu profilov jednej *minor* verzie, alebo priamo zo zoznamu všetkých profilov prináležiacich projektu. Umožňuje taktiež zobrazenie modálneho okna pre jednorazovú špecifikáciu procesu neskoršieho spracovania profilu (*profile postprocessing*) a jeho vykonanie. Pohľad taktiež zobrazuje základnú interpretáciu profilu vo forme grafov rôznych typov. Spĺňa požiadavky **P.2**, **P.4**, **P.5** a **P.6** sekcie **Sec.3** a všetky požiadavky sekcie **Sec.6**.

6.1.1 Rozloženie prvkov na stránkach

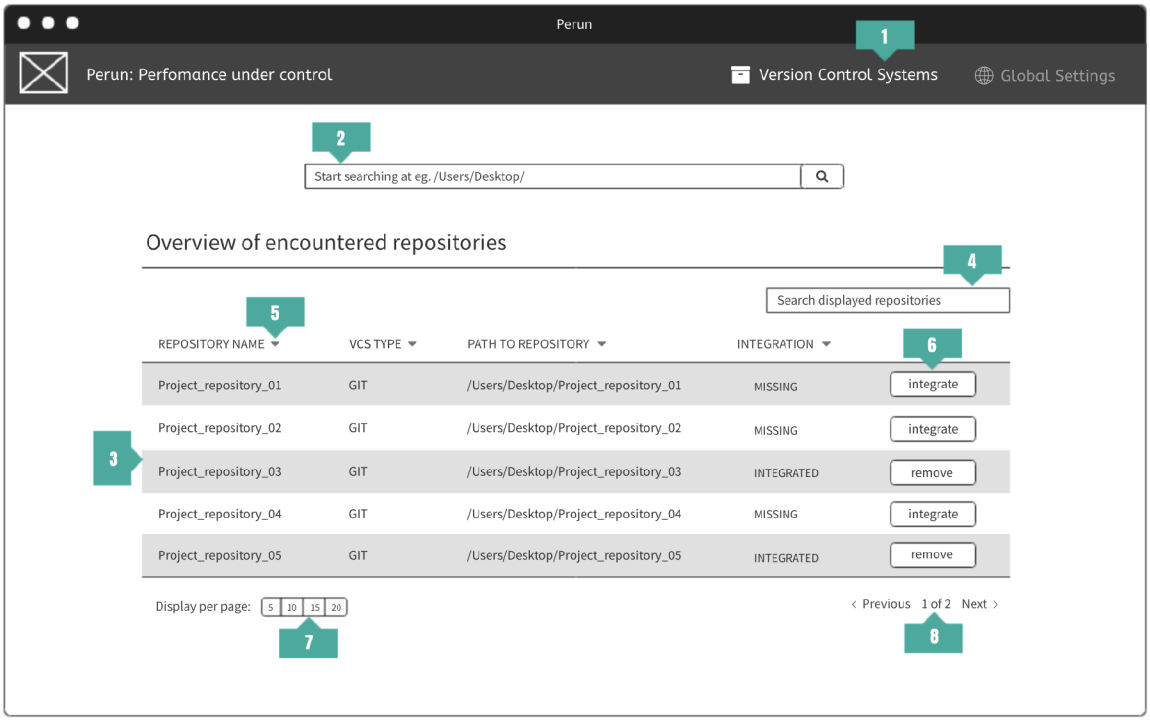
Stránka na najvrchnejšej úrovni návrhu (komponenty repozitárov (*Repositories*) a globálnych nastavení (*Global Settings*), spomínané v sekcii 6.1) obsahuje telo stránky a horizontálne menu — umožňujúce prepínať kontext medzi stránkou globálnych nastavení a zoznamom verziovacích systémov. Názov práve aktívnej sekcie je vždy vyznačený bielou farbou. Pre návrh stránok bol využitý webový nástroj Mockflow[1]. Následne budú bližšie popísané návrhy vybraných komponentov a ich kľúčových častí.

Návrh komponentu repozitárov (*Repositories*):

1. Práve aktívna sekcia vyznačená bielou farbou v hlavnom horizontálnom menu.
2. Vyhľadávací panel slúžiaci pre určenie cesty v súborovom systéme, od ktorej má byť započaté vyhľadávanie dostupných verziovacích systémov.

[1] https://wireframepro.mockflow.com

3. Tabuľka obsahujúca zoznam nájdených verziovacích systémov.
4. Vyhľadávací panel slúžiaci pre filtrovanie zobrazených verziovacích systémov.
5. Tlačidlo pre zmenu abecedného poradia zobrazovaných údajov v danom stĺpci.
6. Tlačidlo pre vytvorenie/odobratie obaľujúcej instancie Perunu nad daným repozitárom.
7. Panel stránkovania umožňujúci meniť počet položiek zobrazených v tabuľke.
8. Panel navigácie medzi jednotlivými stránkami tabuľky repozitárov.



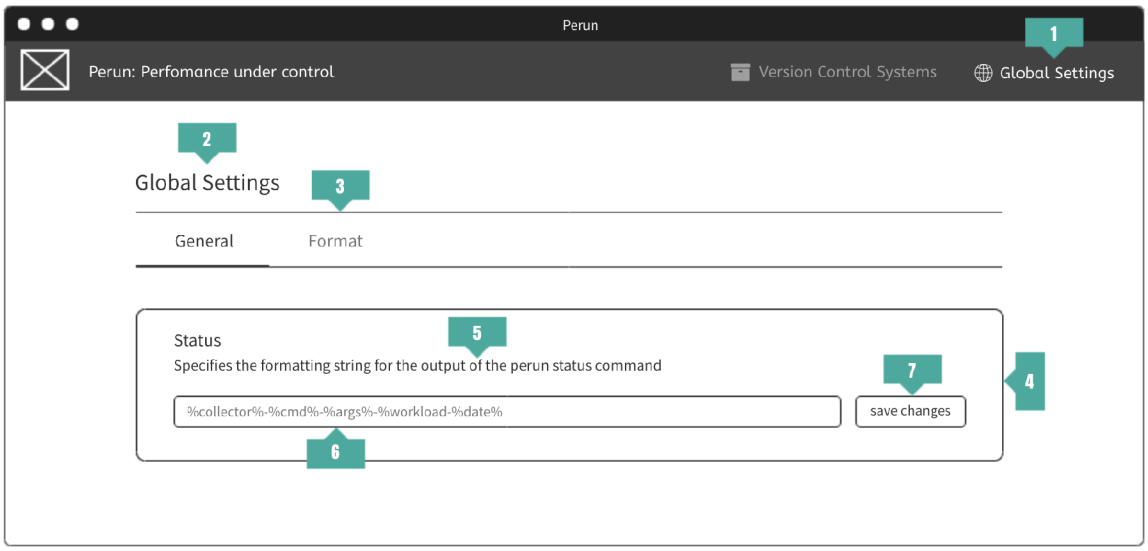
Obrázok 6.2: *Wireframe* pre nárvh časti *Repositories*. Kľúčové časti sú zvýraznené číslom.

Návrh komponenty globálnych nastavení (*Global Settings*):

1. Práve aktívna sekcia vyznačená v hlavnom horizontálnom menu.
2. Hlavička obsahovej stránky; prvý nadpis indikuje názov aktuálnej stránky.
3. Stránkové menu s jednotlivými sekciami globálnej konfigurácie; momentálne aktívnu sekciu indikuje prúžok pod jej názvom.
4. Podsekcia prislúchajúca jednej konkrétnej sekcií v `shared.yml`.
5. Stručné informácie o význame špecifikácie, ktorú podsekcia predstavuje, slúžiace ako nápoveda pre užívateľa.

6. Panel informujúci o momentálnom nastavení podsekcie. Tento panel zároveň slúži aj pre modifikáciu nastavenia. Podľa charakteru podsekcie sa môže jednať o obyčajný panel pre vpisovanie textu či panel s možnosťou výberu jednej/viacerých ponúkaných možností.

7. Tlačidlo pre uloženie prípadných modifikácií podsekcie.



Obrázok 6.3: *Wireframe* pre nárvh časti *Global Settings*. Kľúčové časti sú zvýraznené číslom.

Podstránky časti repozitárov (*Repositories*) obsahujú okrem tela stránky a hlavného horizontálneho menu taktiež postranný navigačný panel, ktorý slúži na prepínanie kontextu podstránok v rámci repozitára. Na obrázku 6.4 je zobrazený návrh jednej z podstránok — časti *minor* verzií.

Návrh komponentu *minor* verzií *Minor Versions*:

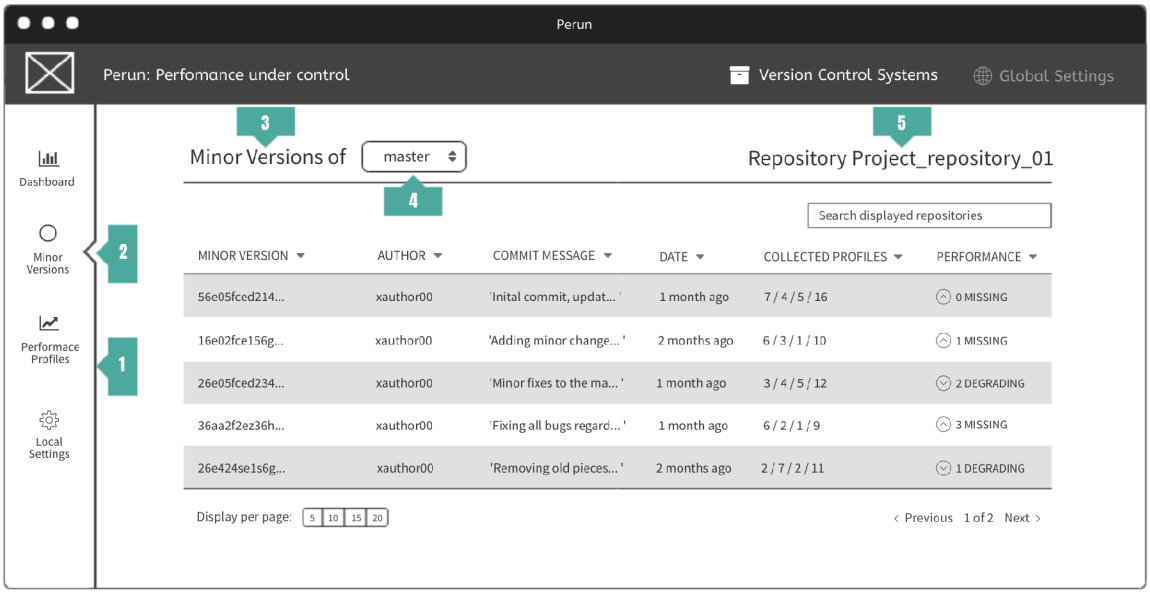
1. Postranný navigačný panel.

2. Práve aktívna sekcia je vyznačená postranným trojuholníkom „zarezávajúcim“ sa do pravej strany položky.

3. Názov aktuálnej podsekcie.

4. Rozbalovacie menu špecifické pre túto podsekciu — slúži pre určenie vetvy vývoja, ktorej *minor* verzie majú byť zobrazené v tabuľke.

5. Názov repozitára, v ktorom sa momentálne nachádzame.



Obrázok 6.4: *Wireframe* pre nárvh časti *Minor Versions*. Kľúčové časti sú zvýraznené číslom.

6.2 Architektúra serverovej časti

V pôvodnom návrhu sa predpokladalo, že vytvárané grafické rozhranie bude môcť pracovať priamo nad nástrojom *Perun* a serverová časť nebude potrebná. Avšak na základe podrobnej analýzy a špecifikácie požiadaviek vyplynula potreba vytvoriť HTTP server, ktorý bude rozhranie, teda klientsku časť, obsluhovať. Server je tak existujúcou vrstvou medzi rozhraním a nástrojom *Perun*.

Obrázok 6.5: Ilustrácia komunikácie klientskej a serverovej časti pri jednostránkových aplikáciach.

Výsledné rozhranie je požadované vo forme jednostránkovej webovej aplikácie. Celá navigácia, resp. smerovanie (*routing*), v rámci jednostránkovej aplikácie je vykonávaná na klientskej strane (viď sekcia 4.1) — teda v jednoduchých aplikáciach, ktoré nevyžadujú načítanie dát zo strany servera na základe užívateľského vstupu, je jedinou úlohou servera

prijímať a spracovávať požiadavky na statické súbory ako obrázky, súbory obsahujúce CSS kód či JavaScript. Server na všetky požiadavky, ktoré nezodpovedajú jednému zo statických súborov, odpovedá odoslaním súboru obsahujúceho jedinú stránku aplikácie (tento súbor sa zvyčajne nazýva `index.html`). Pri jednostránkových aplikáciách, v ktorých je potrebné načítavať dodatočné dáta zo serverovej časti, ako je aj vytvárané grafické rozhranie, server okrem vyššie spomenutej funkcionality taktiež prijíma a odpovedá na HTTP požiadavky, ktoré sú z klientskej časti odosielané pomocou technológie Ajax (príklad tejto komunikácie zobrazuje obrázok 6.5). Pre tento účel bol navrhnutý formát pre správy, ktorými budú časti medzi sebou komunikovať. Klientska časť bude odosielať HTTP požiadavky, ktorých účelom bude buď získať dáta alebo spustiť určitú operáciu nad špecifikovaným zdrojom. Každá požiadavka sa bude skladať z nasledujúcich objektov:

- **HTTP „sloveso“**, špecifikujúce typ operácie, ktorá má byť vykonaná (napr. `GET` či `PATCH`),
- **Hlavička** (*header*), ktorá obsahuje presnejšie informácie o požiadavke,
- **Cesta** ku zdroju, nad ktorým má byť operácia vykonaná, poprípade bližšie určenie akcie,
- **Telo správy** obsahujúce dáta (*payload*) vo formáte *JSON*.

Na základe zhrnutých požiadaviek na vytvárané grafické rozhranie bola navrhnutá nasledujúca HTTP sada pre komunikáciu so serverom:

`GET /repos/{path}`

Získanie repozitárov — požiadavka pre získanie zoznamu repozitárov nachádzajúcich sa v adresári špecifikovanom cestou `path` a jeho podadresároch. Spĺňa požiadavky **P.1** a **P.2** sekcie **Sec.1** a požiadavku **P.1** sekcie **Sec.4**.

`PATCH /repos/{path}/integrate`

Vytvorenie instance *Perun* — požiadavka pre vytvorenie instancie *Perunu*, tzn. obalu nad repozitárom špecifikovaným cestou `path`. Spĺňa požiadavku **P.3** v sekcii **Sec.1**.

`PATCH /repos/{path}/remove`

Odstránanie instance *Perun* — požiadavka pre odstránenie obalu nad repozitárom, pričom repozitár v systéme ostáva nedotknutý — jedná sa teda len o zmazanie zložky `.perun` obsahujúcej internú infraštruktúru instancie *Perunu*.

`GET /repos/{path}/branches`

Získanie vetiev vývoja — požiadavka pre získanie zoznamu vetiev vývoja (*major* verzií), ktoré sú obsiahnuté v repozitári danom cestou `path`. Čiastočne spĺňa požiadavku **P.1** v sekcii **Sec.5**.

`GET /repos/{path}/{branch}/commits`

Získanie *minor* verzií — požiadavka pre získanie všetkých *minor* verzií *major* verzie s názvom `branch` patriacej pod repozitár určený cestou `path`. Čiastočne spĺňa požiadavku **P.1** v sekcii **Sec.5**.

`GET /repos/{path}/commit/{commit}`

Získanie podrobností *minor* verzie — požiadavka pre získanie doplňujúcich údajov o *minor* verzii v repozitári určenom cestou `path`. Vetvu vývoja nie je potrebné špecifikovať kvôli existujúcemu API pre prácu s verziovacími systémami, ktoré je súčasťou implementácie nástroja *Perun* — API totiž vyhľadáva konkrétne *minor* verzie len za pomoci ich SHA. Spĺňa požiadavku **P.2** v sekcii **Sec.5**.

`GET /repos/{path}/profiles`

Získanie zoznamu výkonnostných profilov repozitára — požiadavka pre získanie všetkých výkonnostných profilov vytvorených v repozitári špecifikovanom cestou `path`. Spĺňa požiadavku **P.1** sekcie **Sec.3**.

`GET /repos/{path}/{commit}/profiles`

Získanie zoznamu výkonnostných profilov konkrétnej *minor* verzie — požiadavka pre získanie registrovaných aj doposiaľ nesledovaných výkonnostných profilov spadajúcich pod *minor* verziu s identifikáciou `commit`, ktorá patrí repozitáru na ceste `path`. Spĺňa požiadavku **P.1** sekcie **Sec.3**.

`GET /repos/{path}/{commit}/profiles/profile/info`

Získanie detailov konkrétneho výkonnostného profilu — požiadavka pre získanie dát konkrétneho výkonnostného profilu s názvom *profile*, ktorý spadá pod *minor* verziu s identifikáciou `commit`, ktorá patrí repozitáru na ceste `path`. Spĺňa požiadavky **P.2** a **P.4** sekcie **Sec.3**.

`PATCH /repos/{path}/{commit}/profile-register`

Registrácia konkrétneho výkonnostného profilu — požiadavka pre registráciu výkonnostného profilu spadajúceho pod *minor* verziu s identifikáciou `commit`, ktorá patrí repozitáru na ceste `path`. Cesta k výkonnostnému profilu je odosielaná v tele správy. Spĺňa časť požiadavky **P.6** sekcie **Sec.3**.

`PATCH /repos/{path}/{commit}/profile-unregister`

Odregistrácia konkrétneho výkonnostného profilu — požiadavka pre odregistráciu výkonnostného profilu spadajúceho pod *minor* verziu s identifikáciou `commit`, ktorá patrí repozitáru na ceste `path`. Cesta k výkonnostnému profilu je odosielaná v tele správy. Záznam o výkonnostnom profile je odstránený z indexového súboru danej *minor* verzie (viď sekcia 3.5.1), ale jeho *blob* (viď sekcia 3.5.2) naďalej ostáva v súbore `.perun/objects` (viď sekcia 3.5) — jedná sa teda len o úpravu. Spĺňa časť požiadavky **P.6** sekcie **Sec.3**.

`GET /global-settings`

Získanie globálnych nastavení — požiadavka pre získanie globálnych nastavení nástroja *Perun*. Čiastočne spĺňa požiadavku **P.3** v sekcii **Sec.5**.

`PATCH /global-settings/save`

Upravenie globálnych nastavení — požiadavka pre úpravu globálnych nastavení nástroja *Perun*. Údaje špecifikujúce sekciu, podsekciu a novú hodnotu sú obsiahnuté v tele správy. Čiastočne spĺňa požiadavku **P.3** v sekcii **Sec.5**.

`GET /repos/{path}/local-settings`

Získanie lokálnych nastavení — požiadavka pre získanie lokálnych nastavenich instance *Perun* nad repozitárom na ceste `path`. Čiastočne spĺňa požiadavku **P.1** sekcie **Sec.2**.

`PATCH /repos/{path}/local-settings/save`

Upravenie lokálnych nastavení — požiadavka pre úpravu lokálnych nastavení instance *Perun* nad repozitárom na ceste `path`. Údaje špecifikujúce sekciu, podsekciu a novú hodnotu sú obsiahnuté v tele správy. Čiastočne spĺňa požiadavku **P.1** v sekcii **Sec.7**.

`PATCH /repos/{path}/local-settings/job-matrix/save`

Upravenie matice úloh — požiadavka pre úpravu špecifikácie matice úloh nachádzajúcej sa v lokálnych nastavení instance *Perun* nad repozitárom na ceste `path`. Údaje špecifikujúce sekciu, podsekciu a novú hodnotu/pole nových hodnôt sú obsiahnuté v tele správy. Spĺňa požiadavku **P.2** sekcie **Sec.2**.

`GET /repos/{path}/{commit}/job-matrix/collect-new`

Zber výkonnostných profilov pomocou matice úloh — požiadavka pre spustenie zberu výkonnostných profilov pomocou špecifikovanej matice úloh nad konkrétnou *minor* verziou s identifikáciou `commit`, ktorá patrí repozitáru na ceste `path`. Spĺňa požiadavku **P.3** sekcie **Sec.2**.

`GET /repos/{path}/{commit}/single-job/collect-new`

Zber výkonnostných profilov pomocou špecifikácie úlohy — požiadavka pre spustenie zberu výkonnostných profilov pomocou špecifikácie samostatnej úlohy nad konkrétnou *minor* verziou s identifikáciou `commit`, ktorá patrí repozitáru na ceste `path`. Špecifikácia je prenášaná v tele správy. Jedná sa o prípravu pre budúce rozšírenie funkcionality rozhrania.

`PUT /repos/{path}/{profile}/postprocess`

Spustenie dodatočnej úpravy výkonnostného profilu — požiadavka pre spustenie jednej z jednotiek *postprocessorov* nad výkonnostným profilom identifikovaným `profile` nachádzajúcim sa v repozitári na ceste `path`. Meno jednotky a jej dodatočné parametre sú prenášané v tele správy. Jej spustením sú vytvorené nové výkonnostné profily odvodené upravením hodnôt pôvodného profilu. Tieto profily sú potom uskladnené v súbore `.perun/jobs` (viď sekcia 3.5) a ich zoznamy *postprocessorov* sú doplnené o využitú jednotku. Spĺňa požiadavku **P.5** sekcie **Sec.3**.

Kapitola 7

Implementácia

Na základe špecifikácie požiadaviek a vytvorených návrhov bolo implementované grafické užívateľské rozhranie pre správcu výkonnostných profilov *Perun* vo forme jednostránkovej webovej aplikácie podporenej odľahčenou serverovou časťou. Pre implementáciu rozhrania bol zvolený JavaScriptový framework Vue.js, pre implementáciu podpornej serverovej časti Python mikroframework Flask.

Výsledné klientske webové rozhranie pozostáva z Vue.js komponentov zodpovedajúcich komponentom z návrhu obsiahnutého v sekcii 6.1. Komponenty zdieľajú vnútorný stav (ďalej popísaný v sekcii 7.1) a sú podporené logikou znázornenou na obrázku 6.1. Klient potom komunikuje s podporným serverom pomocou HTTP požiadaviek špecifikovaných v sekcii 6.2. Serverová časť pozostáva z kolekcie funkcií dekorovaných smerovacími cestami špecifikovanými v sekcii 6.2. Vlastné implementácie týchto funkcií potom obsahujú buď obaly nad volaniami funkcií *Perunu*, alebo vlastné obslužné rutiny.

Následuje stručný popis vybraných implementačných detailov ako napr. riešenie vnútorného stavu rozhrania alebo nové vizualizačné techniky.

7.1 Správa vnútorného stavu rozhrania

Na základe prieskumu dostupných JavaScriptových frameworkov, popísaného v kapitole 4, bol pre implementáciu rozhrania zvolený framework Vue.js. Implemetácia grafického rozhrania teda pozostáva z množstva menších Vue komponentov v rôznych príbuzenských vzťahoch a úrovniach zanorenia. Tieto komponenty musia reagovať na zmeny nie len vo svojich stavoch, ale aj v stavoch iných komponentov — nastáva teda situácia, kedy pohľady viacerých komponentov závisia od jedného stavu, alebo naopak, operácie reagujúce na zmenu v pohľadoch rôznych komponentov potrebujú modifikovať rovnaký stav.

Pre modifikáciu hodnôt vlastností patriacich rodičovskému komponentu zvnútra potomka ponúka Vue.js niekoľko možností; realizácia je možná napríklad pomocou prístupovej funkcie `this.$parent.data` či funkcie `$emit` vysieľajúcej udalosti odohrané v potomkovi naspäť k rodičovskému komponentu. V najjednoduchšom prípade je toto možné docieliť taktiež pomocou odovzdávania parametrov detskému komponentu (tzv. *props*). Pri narastajúcej komplexnosti vytváranej aplikácie sa však tieto spôsoby javia značne neprehľadnými a repetitívnymi.

Ako riešenie bol preto využitý Vuex — vzor pre správu stavu aplikácie (*state management pattern*), a zároveň knižnica pre webové aplikácie založené na Vue.js. Vuex slúži pre vytvorenie reaktívneho centralizovaného skladu (*store*) obsahujúceho dáta pre všetky

komponenty aplikácie, pričom stav skladu sa mení len v súlade s predpísanými pravidlami. Sklad je založený na jednostavovom strome (*single state tree*) — tento jediný stavový objekt obsahuje celkový stav na aplikačnej úrovni. Takto má každý, ľubovoľne hlboko zanorený, komponent stromu rovnaký prístup k stavu aplikácie, môže byť na jeho základe aktualizovaný a taktiež môže s týmto stavom manipulovať [7].

Celý zdieľaný stav vytvorenej aplikácie je umiestnený v reaktívnom `state` objekte — jedným z mnohých príkladov z implementácie rozhrania je zoznam *major* verzií obsiahnutých v repozitári, ktorý bolo potrebné získať zo serverovej časti a šíriť medzi rôznymi komponentmi. Pre tento zoznam je v objekte `state` vytvorená osobitná vlastnosť `optionalBranches` (výpis 7.1).

```
export const state = new Vuex.Store({
        state: {
                optionalBranches: [],
        },
})
```

Výpis 7.1: Ukážka objektu `state` s vlastnosťou `optionalBranches`.

V implementácii rozhrania si komponenty najčastejšie vyžadujú načítanie dát zo servera pri ich vytváraní — teda akcie (*actions*) slúžiace na asynchrónne získavanie dát sú naviazané na udalosť `created` životného cyklu komponentu (výpis 7.3). Vo výpise 7.2 je znázornený príklad zdrojového kódu akcie slúžiacej na získanie zoznamu *major* verzií vybraného repozitára.

```
export const state = new Vuex.Store({
        loadBranches: function({ commit }, payload) {

                payload.path = tidyURL(payload.path);
                var URL = '/repos/' + payload.path + '/branches';

                axios.get(URL).then(response => {
                        if (response.status == 200) {
                                commit('setOptionalBranches', response.data);
                        }
                })
                .catch(error => {
                        commit('setError', error)
                });

        },
})
```

Výpis 7.2: Ukážka implementácie akcie pre získanie dát zo serveru a vyvolanie metódy na modifikáciu stavu.

```
export default {
	created: function() {
		this.$store.dispatch({
			type: 'loadBranches',
			path: this.$store.getters.currentRepoPath,
		});
	},
}
```

Výpis 7.3: Ukážka životného cyklu `created` s vyvolávaním akcie.

Po úspešnom získaní dát je zdieľaná vlastnosť `optionalBranches` modifikovaná pomocou metódy vo Vuex nazývanej mutácia (*mutation*). Podobne ako pri udalostiach, každá mutácia má správcovskú funkciu — táto spravidla prijíma ako prvý parameter stav skladu a vo svojom vnútri obsahuje operáciu pre modifikáciu (výpis 7.4).

```
export default {
	setOptionalBranches: function(state, payload) {
		state.optionalBranches = payload.optionalBranches;
	},
}
```

Výpis 7.4: Ukážka metódy mutácie modifikujúcej stav aplikácie.

K stavu aplikácie z vnútra komponentu je možné pristúpiť pomocou špeciálnych reaktívnych vlastností (*computed properties*) Vuex nazývaných *getters*. Hodnoty takýchto vlastností sú ukladané do rýchlej pamäte a sú prehodnocované len vtedy, keď sa zmení hodnota niektorej z ich závislostí. Vo výpise 7.5 je znázornený *getter* pre vlastnosť `optionalBranches`, ktorý umožní načítanie hodnoty zdielanej vlasnosti `optionalBranches` do komponenty, kde je s touto hodnotou už možné pracovať ako s lokálnou.

```
export default {
	computed: {
		optionalBranches: function() {
			return this.$store.getters.optionalBranches;
		},
	}
}
```

Výpis 7.5: Ukážka *getter* pre vlastnosť `optionalBranches`.

7.2 Komunikácia klientskej časti so serverovou časťou

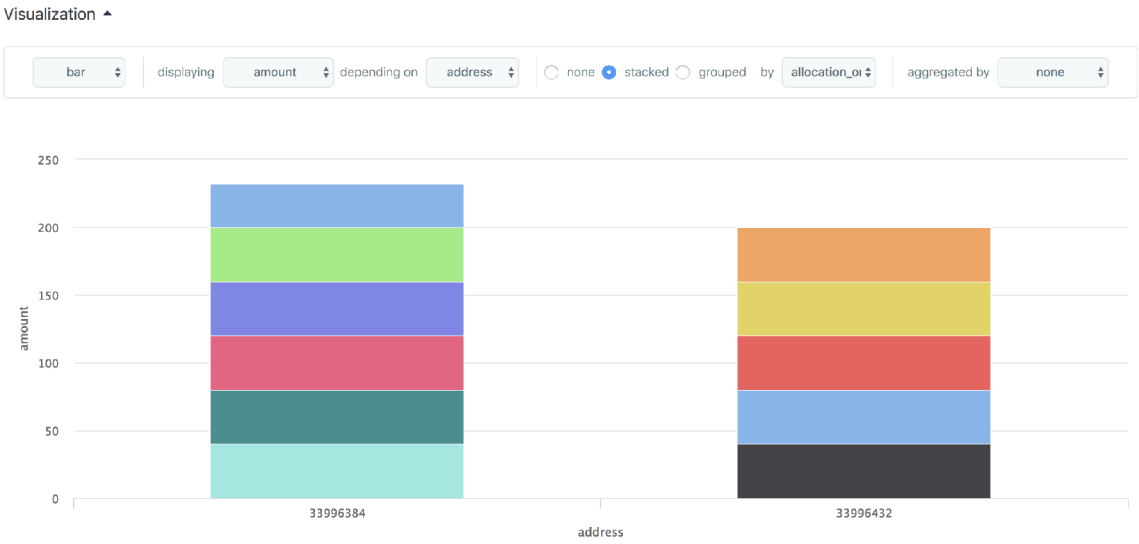
Prvá verzia Vue.js obsahovala zabudované HTTP aplikačné rozhranie `vue-resource`. Toto rozhranie ale prestalo byť pri Vue 2.0 podporované a ako alternatíva bola použitá knižnica

Axios. Jedná sa o knižnicu založenú na tzv. prísľuboch (*Promise-based*)[1], ktorá uľahčuje odosielanie asychrónnych HTTP požiadaviek na server. Príklad využitia knižnice Axios pre získanie dát zo servera bol uvedený na obrázku 7.2 v sekcii 7.1. V rozhraní sú asynchrónne volania pomocou tejto knižnice realizované len vo vnútri akcií (*actions*, viď sekcia 7.1).

7.3 Vizualizácia profilovacích dát

Súčasná verzia nástroja *Perun* poskytuje základné vizualizácie profilovacích dát výkonnostných profilov priamo v termináli alebo za pomoci knižnice *Bokeh*[2] jazyka Python. Výstupom tejto knižnice sú interaktívne HTML súbory, ktoré sú následne zobrazené za pomoci internetového prehliadača a súčasne by mohli byť využiteľné aj pri vizualizácii vo vytváranom rozhraní. Cenou za túto interaktivitu je však pomerne vysoká doba vykresľovania grafov a nízka škálovateľnosť. Ako alternatíva je možné presunutie logiky vizualizácie do klientskej časti a využiť tak jednu z knižníc v JavaScripte.

Profilovacie dáta (zdroje — *resources*, viď sekcia 3.3) obsiahnuté v profile sú na strane serveru za pomoci knižnice *Pandas*[3] pretransformované do tabuľkovej formy (tzv. *DataFrame* — dvojdimenzionálna asociatívna dátová štruktúra, kde hodnoty stĺpcov môžu byť rôznych dátových typov). Riadky takejto tabuľky sú následne pretransformované do formátu *JSON* a odoslané na klientsku stranu.



Obrázok 7.1: Ukážka z implementovaného rozhrania — stĺpcový čiarový graf a panel možností. Graf zobrazuje veľkosti alokovaných dát na dvoch pamäťových adresách (33996384 a 33996432) skladané podľa poradia alokácie.

Na klientskej strane sú dáta po prijatí uložené do zdieľaného stavu aplikácie. Komponent zobrazujúci konkrétny výkonnostný profil (viď sekcia 6.1) vyhodnotí aktuálne nastavenia zobrazenia určené užívateľom, zdieľané profilovacie dáta nahrá do reaktívnej lokálnej

[1] Prísľub (*Promise*) je objekt reprezentujúci stav čakajúci na prípadné dokončenie, či zlyhanie vykonania asynchrónnej operácie.

[2] https://bokeh.pydata.org/en/latest/

[3] https://pandas.pydata.org/

premennej a za pomoci funkcie `groupBy` (pochádzajúcej z `npm`[4] modulu `json-groupby`[5]) dáta pretransformuje do formy, ktorú dokáže vybraná JavaScriptová knižnica vizualizovať. Pre tento účel bola zvolená JavaScriptová knižnica Highcharts.js, pre ktorú je pri Vue.js dostupný obaľujúci komponent[6], a zároveň poskytuje možnosť tvorenia interaktívnych a vysoko škálovateľných grafov.

Z analýzy požiadaviek na rozhranie vyplynulo, že implementácia má podporovať tri základné typy grafov — stĺpcový (zodpovedajúci vizualizácii *bars*), čiarový (zodpovedajúci vizualizácii *flow*) a korelačný (zodpovedajúci vizualizácii *scatter*). Pri stĺpcovom a čiarovom grafe je možné okrem tradičného zobrazenia vykonať skupinovanie dát v rámci jedného grafu — pre stĺpcový graf je podporované skladané a skupinové zobrazenie, pre čiarový potom skladané a akumulované. Užívateľ má možnosť dynamicky voliť hodnoty osí grafu, pričom hodnoty osi Y je možné ďalej upravovať za pomoci niektorej z ponúkaných agregačných funkcií (ako je napr. súčet, priemer, maximum či minimum). Obrázok 7.1 zobrazuje úryvok z implementovaného rozhrania — skladaný stĺpcový graf, ktorý vizualizuje závislosť veľkosti dát (*amount*) na adrese (*address*) skladaných podľa poradia alokácie (*allocation order*).

7.4 Minimalizácia zdrojových kódov

V prípade, že výsledná aplikácia manipuluje s DOM len v malej miere, je možné Vue.js do projektu zahrnúť tradičnou cestou — pomocou HTML značky *script* odkazujúcou sa na externý zdroj pomocou atribútu *src*. V našom prípade, pri ktorom je celá klientska časť postavená na jednom z moderných frameworkov miesto čistého JavaScriptu alebo jQuery, bolo potrebné využiť tzv. *module bundler* — JavaScriptový nástroj, ktorý združuje kód a všetky jeho závislosti do jedného minimalizovaného súboru. Takto minimalizovaný kód je potom pripravený pre nasadenie do používania. Pri vytváraní implementácie grafického rozhrania bol využitý otvorený nástroj *Webpack*[7]. Použitie *Webpack* k zabaleniu výslednej aplikácie spĺňa mimofunkčnú požiadavku o možnosti jednoduchej distribúcie grafického rozhrania (viď sekcia 5.2).

[4] Správca balíčkov pre jazyk JavaScript
[5] https://www.npmjs.com/package/json-groupby
[6] https://github.com/highcharts/highcharts-vue
[7] https://webpack.js.org/

Kapitola 8

Experimentálne vyhodnotenie

Funkcionalita výsledného rozhrania je demonštrovaná na troch experimentoch. Pre každý experiment je dedikovaný samostatný repozitár obsahujúci program napísaný v jazyku C/C++/Python s netriviálnou históriou verzií s obaľujúcou instanciou nástroja *Perun* obsahujúcou výkonnostné profily priradené k jednotlivým verziám. Na prvých dvoch repozitároch je ukázaná využiteľnosť rozhrania sadou zaujímavých experimentov, na poslednom repozitári je potom otestovaný a demonštrovaný zbytok funkcionality.

8.1 Demonštrácia zobrazovania degradácie výkonu

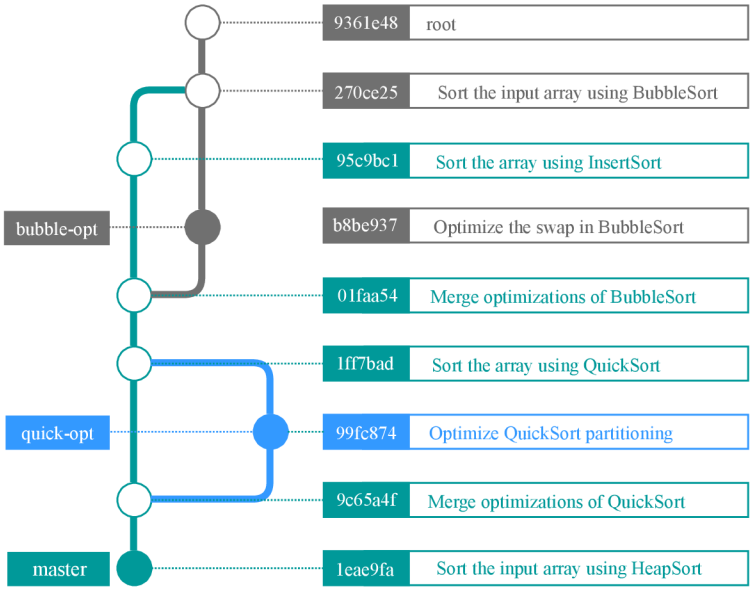
Cieľom prvého experimentu je demonštrovať zobrazovanie nájdených výkonnostných zmien (optimalizácií a degradácií) v kontexte jednotlivých *minor* verzií projektu. Program obsiahnutý v testovacom repozitári postupne prechádza cyklom a radí prvky v náhodne vygenerovaných poliach až po maximálnu veľkosť. Pre samotné radenie je postupne v jednotlivých *commitoch* využívaných niekoľko algoritmov, ako sú napríklad:

- ***Bubble sort*** — opakovane prechádza cez pole porovnávajúc dva prvky — pokiaľ prvky nie sú v správnom poradí, dochádza k výmene poradia. Algoritmus má teoretickú časovú zložitosť $\mathcal{O}(n^2)$ a priestorovú zložitosť $\mathcal{O}(1)$ [17].
- ***Insert sort*** — rozdeľuje pole na zoradenú a nezoradenú časť, pričom zoradenú časť buduje z nezoradenej časti postupne po jednom prvku. Podobne ako *Bubble sort* má teoretickú časovú zložitosť $\mathcal{O}(n^2)$ a priestorovú zložitosť $\mathcal{O}(1)$ [17].

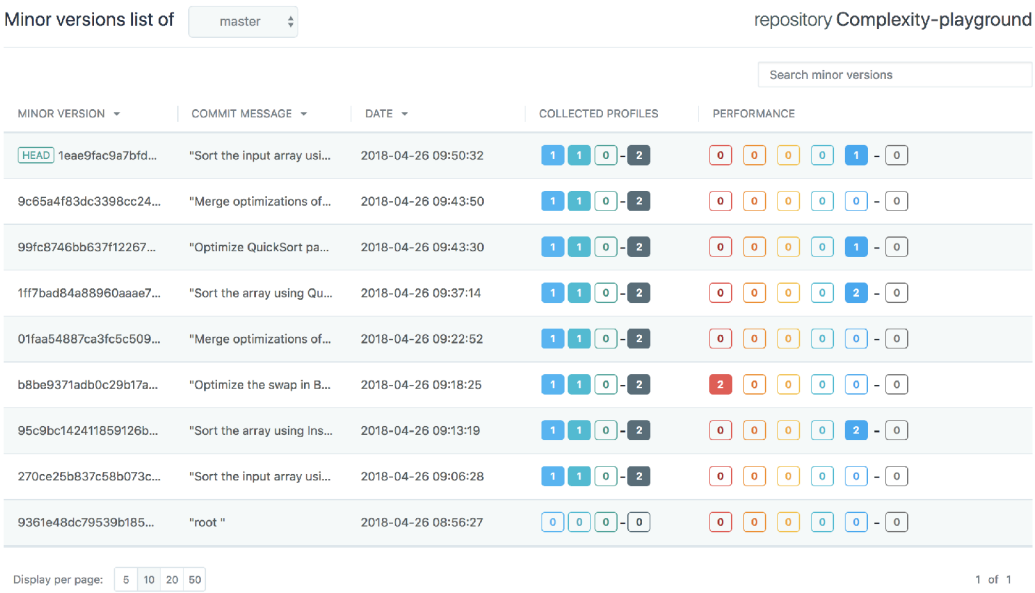
Na obrázku 8.1 je zobrazená história *minor* verzií repozitára využitého pri experimente. Projekt obsiahnutý v repozitári sa snaží experimentálne zisťovať, ktorá metóda radenia je pre jeho implementáciu optimálna. Vo verzii `95c9bc` došlo k výmene riadiaceho algoritmu z *Bubble sort* (využívaného pri verzii `270ce25`) na *Insert sort*. Oba algoritmy majú teoretickú kvadratickú časovú zložitosť, avšak v [17] bolo experimentálne preukázané, že z tejto dvojice je *Insert sort* o niečo rýchlejší. Obrázok 8.2 zobrazuje snímku implementovaného rozhrania — komponent všetkých *minor* verzií (viď sekcia 6.1) — kde panel výkonnosti (*performance*) signalizuje pri verzii `95c9bc` optimalizáciu spotreby zdrojov. Zobrazenie v rozhraní teda v tomto prípade potvrdzuje experimenty vykonané v [17].

Zaujímavosťou je vetva *bubble-opt*, kde sa vo verzii `b8b937` vývojár snažil o zrýchlenie reálnej implementácie *Bubble sort* optimalizáciou výmeny dvoch prvkov. Miesto klasickej

výmeny bola využitá C++ šablónová funkcia `std::swap`, ktorá je však *in-line* až pri kompilácii s parametrom `-o2` a vyššie, zatiaľ čo program, nad ktorým bola profilácia vykonaná, bol kompilovaný s parametrom `-o`. Volanie tejto funkcie si potom oproti predošlému riešeniu vyžadovalo vyššie réžijné náklady, čo potvrdzuje aj nájdená výkonnostná degradácia vo verzii `b8b937`.



Obrázok 8.1: Grafická reprezentácia histórie repozitára využitého pri prvom experimente. Obdĺžniky na ľavej strane obrázku obsahujú názvy *major* verzií, ich umiestnenie indikuje poslednú *minor* verziu pridanú do vetvy vývoja (*head commit*).



Obrázok 8.2: Snímka komponentu *minor* verzií implementovaného rozhrania zobrazujúca panel výkonnostnej degradácie (*performance*). Význam čísel panelu zľava — Degradácia, Možná degradácia, Bez zmien, Možná optimalizácia, Optimalizácia, Neznáme.

Rozhranie tak užívateľa prehľadne informuje o nechcenej degradácii výkonu a umožňuje včas podchytiť výkonnostné chyby.

Zvyšok histórie projektu potom potvrdzuje očakávané optimalizácie [15][17] — použitím algoritmu *Quick sort* (vo verzii `1ff7bad`), optimalizáciou deliaceho algoritmu (vo verzii `99fc874`) a finálnym použitím algoritmu *Heap sort* (vo verzii `1eae9f`).

Výsledky analýzy degradácie zobrazované v implementovanom rozhraní sú vo väčšine prípadov v súlade s očakávaniami. Možnosť zobrazovať výsledky degradačnej analýzy v kontexte konkrétnych *minor* verzií prináležiacich *major* verziám, pre ktorú bol daný experiment vykonaný, je funkcionalita, ktorú textové rozhranie nástroja *Perun* momentálne neposkytuje a tento experiment preto demonštruje rozširujúcu využiteľnosť grafického rozhrania.

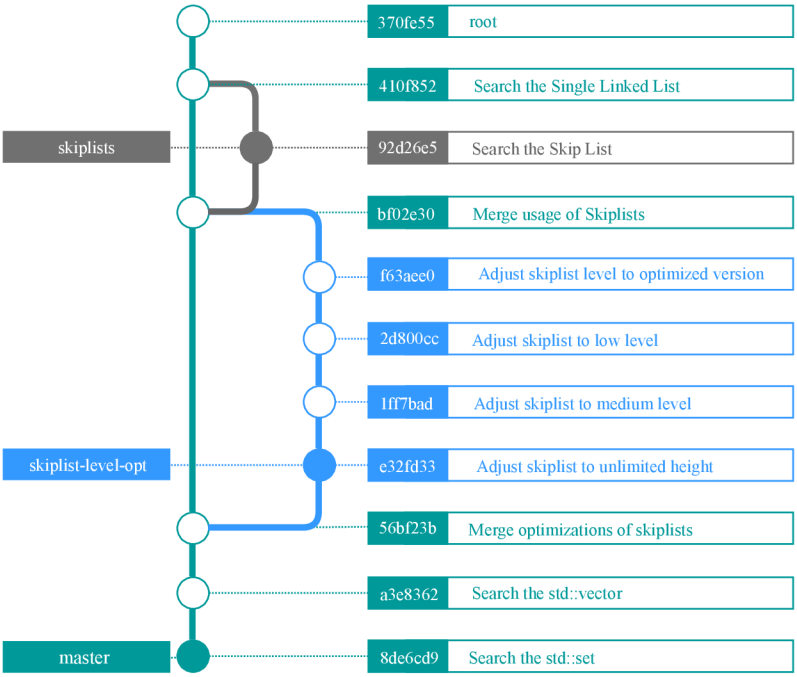
8.2 Demonštrácia vizualizácií

Cieľom tohto experimentu je demonštrovať vizualizácie výkonnostných profilov. Hlavná slučka programu v testovacom repozitári dynamicky vkladá do dátovej štruktúry náhodne generované prvky, pričom pri vložení môže náhodne dôjsť k vyhľadaniu určitého prvku. Pre ukladanie prvkov je vystriedaných niekoľko dátových štruktúr, ako je napríklad:

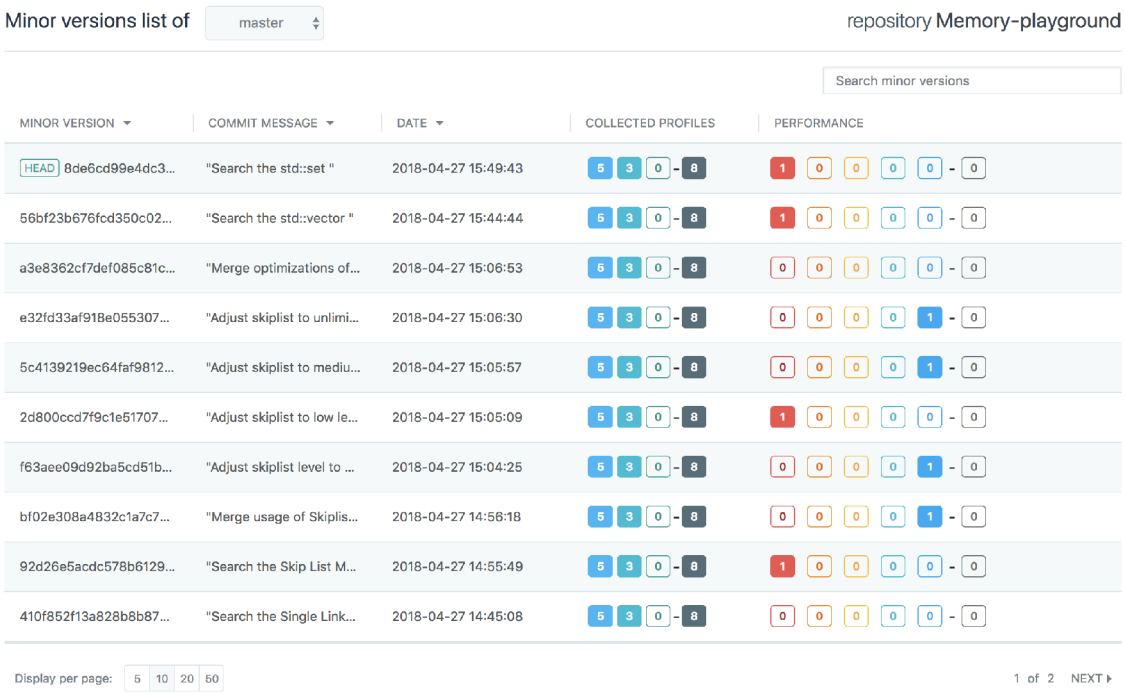
- **Jednosmerne viazaný zoznam** — dynamická dátová štruktúra, v ktorej sú prvky usporiadané lineárne za sebou, pričom každý uzol obsahuje odkaz na svojho následníka. Vkladanie prvku má konštantnú časovú zložitosť, vyhľadávanie potom lineárnu časovú zložitosť. Priestorová zložitosť je $\mathcal{O}(n)$ [14].
- ***Skip list*** — pravdepodobnostná dátová štruktúra. Prvok *skip listu* môže obsahovať viac ako jeden ukazateľ, pričom presný počet ukazateľov je určený generátorom pseudonáhodných čísel — dochádza tak k prepojeniu prvkov, ktoré navzájom nie sú v susedskom vzťahu, čím sa vytvára niekoľko úrovní ukazateľov, ktoré umožňujú rýchlejšie vyhľadávanie. Vkladanie aj vyhľadávanie prvkov v *skip liste* má logaritmickú časovú zložitosť. Pamäťová zložitosť je $\mathcal{O}(n \cdot \log n)$ [14].

Na obrázku 8.3 sa nachádza história repozitára využitého pri tomto experimente. Prvá implementácia programu využíva pre uskladnenie prvkov jednosmerne viazaný zoznam. Pri verzii `410f852` bol zoznam nahradený štruktúrou *skip list*, pričom v nasledujúcich verziách dochádzalo k experimentovaniu s jej parametrami (maximálneho počtu výškových úrovní a pravdepodobnosti vytvorenia novej výškovej úrovne pre daný prvok).

Na obrázku 8.4 je zobrazená snímka implementovaného rozhrania znázorňujúca *major* verziu *master* testovacieho repozitára. Rozhranie ukazuje, že nastala optimalizácia medzi verziou `e32fd33` s implementáciou optimalizovaného *skip listu* a verziou `410f852` využívajúcou jednosmerný zoznam. Implementácia rozhrania naviac umožňuje porovnanie akýchkoľvek dvoch výkonnostných profilov za pomoci ich vizualizácie — touto funkcionalitou je demonštrované možné použitie pre porovnanie profilov prináležiacich dvom zmieneným verziám. Výsledok porovnania výkonnostných profilov zameraných na spotrebu pamäti na obrázkoch D.2 a D.1 ukázal, že pri rovnakých podmienkach (pole o 1000 prvkoch) *skip list* vykoná okolo 3200 alokácií pamäti pre celkovo alokovaných 24 000 bitov, zatiaľ čo jednosmerný zoznam vykoná 1000 alokácii pre 16 000 alokovaných bitov. Tieto hodnoty súhlasia s predpokladom, že *skip list* má väčšiu pamäťovú náročnosť ako jednosmerný zoznam.



Obrázok 8.3: Grafická reprezentácia histórie repozitára využitého pri druhom experimente. Obdĺžniky na ľavej strane obrázku obsahujú názvy *major* verzií, ich umiestnenie indikuje poslednú *minor* verziu pridanú do vetvy vývoja (*head commit*).



Obrázok 8.4: Snímka komponentu *minor* verzií implementovaného rozhrania zobrazujúca panel výkonnostnej degradácie (*performance*). Význam čísel panelu zľava — Degradácia, Možná degradácia, Bez zmien, Možná optimalizácia, Optimalizácia, Neznáme.

Pri porovnaní výkonnostných profilov zameraných na spotrebovaný čas rozhranie ukazuje (obrázok D.3), že pri milióne vkladaných prvkov spotrebuje *skip list* menší procesorový čas (`sys`: 0.12 ms, `user`: 2.14 ms) ako jednosmerný zoznam (`sys`: 0.04 ms, `user`: 2.85 ms). Zo zobrazení teda vyplýva, že *skip list* síce má väčšiu pamäťovú zložitosť ako jednosmerný zoznam, ale na druhú stranu je rýchlejší.

8.3 Demonštrácia základnej funkcionality rozhrania

Predošlé dva experimenty boli zamerané na demonštráciu funkcionality spojenej s požiadavkami obsiahnutými v sekciách **Sec.5** a **Sec.6** a s rozšíreniami aktuálnej funkcionality. Posledný experiment sa zameriava na overenie základnej funkcionality vyplývajúcej z ostatných požiadaviek, pričom testy boli manuálne vykonávané nad repozitárom samotného nástroja *Perun*. Jednotlivé vykonané testy a očakávané výstupy sú obsiahnuté v tabuľkách 8.1, 8.2 a 8.3, pričom testovacia sada v tabuľke 8.1 je taktiež bližšie popísaná v prílohe D.

Testovanie integrácie nástroja *Perun*

Scenár experimentu	Očakávaný výsledok	Stav
Vyhľadanie dostupných repozitárov v neexistujúcom adresári	Zobrazí sa modálne okno hlásiace chybu	Splnené
Vyhľadanie dostupných repozitárov v adresári	Zobrazí sa zoznam dostupných repozitárov	Splnené
Prepnutie kontextu do repozitára, nad ktorým neexistuje instancia *Perunu*	Kontext nie je prepnutý	Splnené
Vytvorenie instancie *Perun* nad repozitárom	Zobrazí sa modálne okno hlásiace úspešnú operáciu a instancia je vytvorená	Splnené
Prepnutie kontextu do repozitára, nad ktorým existuje instancia *Perunu*	Kontext je prepnutý na nástenku repozitára	Splnené
Zmazanie instancie *Perun* repozitára	Zobrazí sa modálne okno pýtajúce potvrdenie zamýšľanej akcie. Po potvrdení je instancia odstránená.	Splnené

Tabuľka 8.1: Testovacia sada integrácie nástroja *Perun*.

Testovanie konfigurácie nástroja *Perun*

Scenár experimentu	Očakávaný výsledok	Stav
Zobrazenie globálnych nastavení	Zobrazia sa globálne nastavenia	Splnené
Zobrazenie lokálnych nastavení repozitára	Zobrazia sa lokálne nastavenia	Splnené
Upravenie a uloženie nastavení matice práce v lokálnych nastaveniach	Zobrazí sa modálne okno hlásiace úspešnú operáciu a nastavenie je uložené	Splnené

Tabuľka 8.2: Testovacia sada konfigurácie nástroja *Perun*.

Testovanie práce s výkonnostnými profilmi

Scenár experimentu	Očakávaný výsledok	Stav
Prepnutie kontextu do konkrétnej verzie projektu	Kontext je prepnutý do zobrazenia všetkých výkonnostných profilov prináležiacich verzii	Splnené
Spustenie automatizovaného zberu výkonnostných profilov nad konkrétnou verziou projektu pri nesprávne zadanej matici práce	Zobrazí sa modálne okno hlásiace chybu a žiadna akcia nie je vykonaná	Splnené
Spustenie automatizovaného zberu výkonnostných profilov nad konkrétnou verziou projektu	Výkonnostné profily sú vytvorené a načítané do sekcie neregistrovaných profilov verzie	Splnené
Priradenie konkrétneho výkonnostného profilu prináležiacej verzii	Zobrazí sa modálne okno hlásiace úspešnú operáciu a vybraný profil sa zobrazí na konci sekcie registrovaných profilov	Splnené
Prepnutie kontextu na zobrazenie informácii o konkrétnom vybranom výkonnostnom profile	Kontext je prepnutý	Splnené
Spustenie neskoršieho spracovania výkonnostného profilu (*postprocess*) s nesprávne zadanými parametrami	Zobrazí sa modálne okno hlásiace chybu a žiadna akcia nie je vykonaná	Splnené
Spustenie neskoršieho spracovania výkonnostného profilu (*postprocess*) so správnymi parametrami	Zobrazí sa modálne okno hlásiace úspešnú operáciu a nové výkonnostné profily, ktorých zoznam jednotiek *postprocessorov* bol aktualizovaný, sú pridané do sekcie neregistrovaných profilov verzie	Splnené
Odregistrovanie výkonnostného profilu od verzie	Zobrazí sa modálne okno pýtajúce potvrdenie zamýšľanej akcie. Po potvrdení je profil odstránený a kontext je prepnutý naspäť na zobrazenie verzie	Splnené

Tabuľka 8.3: Testovacia sada práce s výkonnostnými profilmi.

Kapitola 9

Záver

Cieľom tejto práce bolo špecifikovať, navrhnúť a implementovať grafické užívateľské rozhranie pre nástroj *Perun*. Výsledné rozhranie je vo forme webovej aplikácie so zameraním na globálnu štatistiku projektov a jednoduchosť používania a komunikuje s jednoduchým webovým serverom. Vo výsledku bola implementovaná väčšina súčasnej funkcionality existujúceho terminálového rozhrania.

Veľká pozornosť bola ďalej zameraná na sledovanie výkonnostnej degradácie medzi jednotlivými *minor* verziami projektu — rozhranie disponuje degradačným panelom informujúcim o stavoch degradácie jednotlivých *minor* a *major* verzií. Rozhranie taktiež umožňuje porovnanie dvoch výkonnostných profilov za pomoci vizualizácie profilovacích dát; jedná sa o funkcionalitu, ktorou nástroj *Perun* doteraz nedisponoval a rozširuje tak jeho aktuálne možnosti.

Výsledné rozhranie je demonštrované na rade experimentov za pomoci projektov s netriviálnou históriou. Z výsledkov experimentov možno pozorovať, že výstup rozhrania sa zhoduje s teoretickými predpokladmi a spĺňa požiadavky stanovené zadávateľom projektu.

Počas doby riešenia práce bola do nástroja *Perun* pridávaná nová funkcionalita, pričom nie so všetkými zmenami bolo možné rátať vopred — v blízkej budúcnosti bude rozhranie rozširované o nové prvky v závislosti od novo integrovanej funkcionality. Taktiež je v pláne pridať možnosť špecifikácie samostatnej úlohy pre zber profilovacích dát nad ľubovoľnou *minor* verziou projektu, pričom jednotlivé vstupné panely budú obsahovať predpripravené špecifikačné možnosti automaticky aktualizované podľa aktuálnej ponuky nástroja. Do budúcnosti je ďalej v pláne vytvorené rozhranie plne integrovať do hlavnej vetvy repozitára *Perun* (*upstreamu*).

Literatúra

[1] *Angular documentation*. [Online; navštívené 15.12.2017].
Dostupné z: https://angular.io/tutorial

[2] *AngularJS documentation*. [Online; navštívené 09.12.2017].
Dostupné z: https://docs.angularjs.org/guide

[3] *PerfRepo documentation*. [Online; navštívené 09.12.2017].
Dostupné z: https://github.com/PerfCake/PerfRepo

[4] *Perun documentation*. [Online; navštívené 21.01.2018].
Dostupné z: https://github.com/tfiedor/perun

[5] *React documentation*. [Online; navštívené 08.12.2017].
Dostupné z: https://reactjs.org/docs/

[6] *Understanding GitHub workflow*. [Online; navštívené 19.02.2018].
Dostupné z: https://guides.github.com/introduction/flow/

[7] *Vue.js documentation*. [Online; navštívené 17.01.2018].
Dostupné z: https://vuejs.org/v2/guide/

[8] *What is Git?* [Online; navštívené 10.02.2017].
Dostupné z: https://www.visualstudio.com/learn/what-is-git/

[9] A., E.; Scott, J.: *SPA Design and Architecture*. New York: Manning Publications, 1. vydanie, 2015, ISBN 9781617292439.

[10] Chacon, S.; Straub, B.: *Pro Git*. New York: Apress, 2. vydanie, 2014, ISBN 978-1484200773.

[11] Grunwald, J.: *PerfRepo – nájstroj pro reprezentaci výsledků výkonnostních testů a detekci výkonnostní regrese*. Magisterská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.
Dostupné z: http://hdl.handle.net/10467/69567

[12] Krug, S.: *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability*. Berkley,CA: New Riders, 3. vydanie, 2014, ISBN 978-0321965516.

[13] Milner, M.: *AngularJS: MVC implementation*. [Online; navštívené 09.12.2017].
Dostupné z: https://www.pluralsight.com/blog/software-development/tutorial-angularjs-mvc-implementation

[14] Pavela, J.: *Knihovna pro profilování datových struktur programů C/C++*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2017. Dostupné z: http://www.fit.vutbr.cz/study/DP/BP.php?id=20045

[15] Rösler, U.: *A limit theorem for "quicksort"*. [Online; navštívené 29.04.2018]. Dostupné z: http://www.numdam.org/item?id=ITA_1991__25_1_85_0

[16] Sommerville, I.: *Software engineering*. Boston: Pearson, 9. vydanie, 2011, ISBN 978-013703515-1.

[17] Yang, Y.; Yu, P.; Gan, Y.: Experimental study on the five sort algorithms. In *2011 Second International Conference on Mechanic Automation and Control Engineering*, July 2011, doi:10.1109/MACE.2011.5987184.

Príloha A

Zoznam využitých balíčkov a grafiky

A.1 Balíčky a knižnice

- **webpack** https://www.npmjs.com/package/webpack
- **lodash** https://www.npmjs.com/package/lodash
- **vue-highcharts** https://www.npmjs.com/package/vue-highcharts
- **vue-loader** https://github.com/vuejs/vue-loader
- **vue-router** https://www.npmjs.com/package/vue-router
- **vuex** https://www.npmjs.com/package/vuex
- **vuex-persistedstate** https://www.npmjs.com/package/vuex-persistedstate
- **axios** https://www.npmjs.com/package/axios
- **json-groupby** https://www.npmjs.com/package/json-groupby

A.2 Grafika

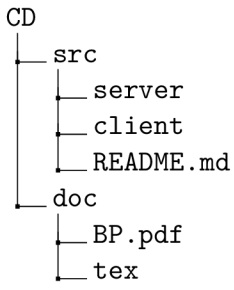
Ikony a obrázky využité pri implementácii rozhrania pochádzali z nasledujúcich zdrojov:

- https://fontawesome.com/?from=io (licencia CC BY 4.0)
- https://www.flaticon.com/ (licencia CC BY 3.0)

Príloha B

Obsah pamäťového média

Na priloženom pamäťovom médiu sa v adresári `src` nachádza podadresár `server`, ktorý obsahuje Python skripty implementujúce serverovú časť a podadresár `dist` obsahujúci jedinú HTML stránku rozhrania, statické súbory a minimalizované JavaScriptové skripty. Podadresár `client` obsahuje Vue.js skripty v pôvodnom stave. Adresár `doc` obsahuje text technickej správy vo formáte PDF a v podadresári `tex` sa nachádzajú zdrojové kódy práce vo formáte LaTeX.



Príloha C

Manuál pre inštaláciu

C.1 Inštalácia nástroja *Perun*

Pre správne fungovanie rozhrania je najprv potrebné stiahnuť a nainštalovať nástroj *Perun*. Nástroj je voľne dostupný na stránke *GitHub*.

```
$ git clone https://github.com/tfiedor/perun.git
$ cd perun
$ make init
$ make install
```

V prípade problémov alebo inej formy inštalácie je potrebné konzultovať súbor `README` *Perun* repozitára, projektovú dokumentáciu alebo stránku *Github*.

C.2 Inštalácia rozhrania

Po extrahovaní adresára `server` do adresárového systému je rozhranie možné nainštalovať za pomoci pridaného `Makefile`.

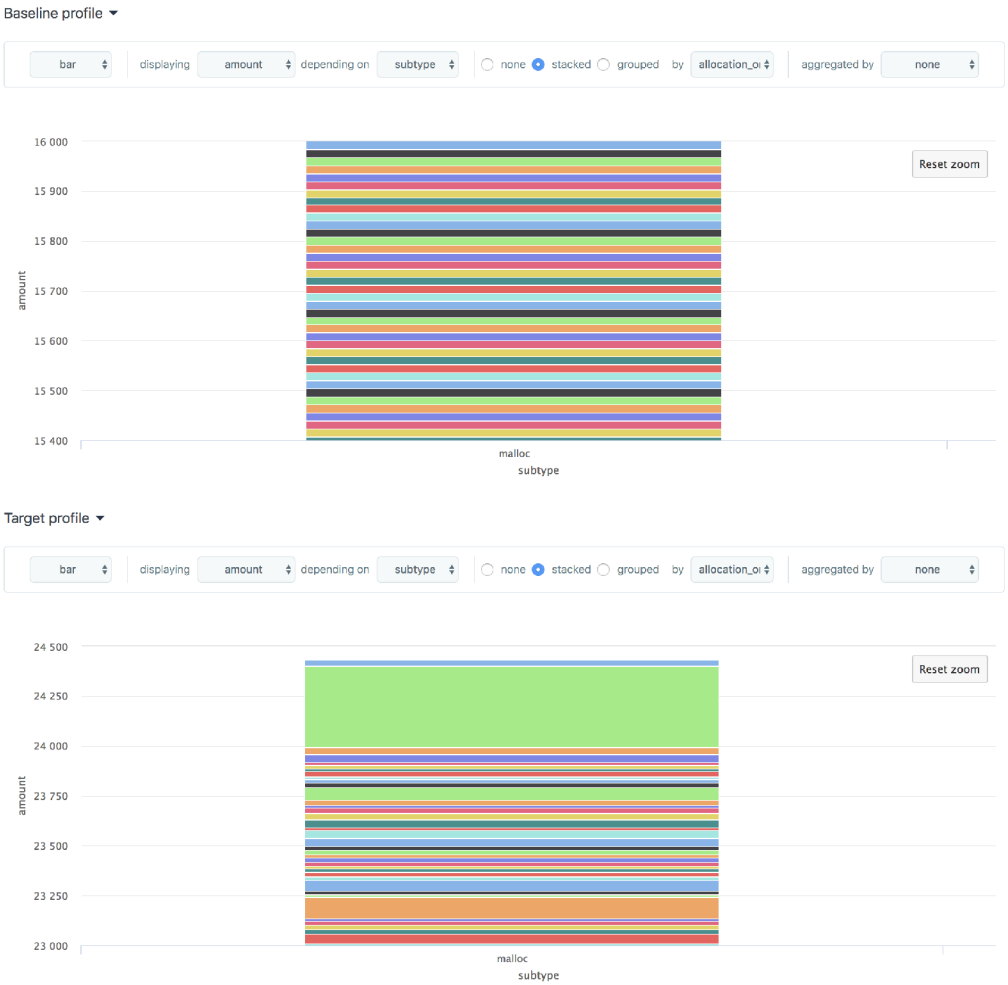
- `$ make install` nainštaluje závislosti obsiahnuté v súbore `requirements.txt`.
- `$ make run` spustí Flask server, rozhranie sa nachádza na adrese `10.211.55.7:5000`.

Takto spustený server pracuje s minimalizovanými JavaScriptovými skriptami rozhrania, ktoré sú obsiahnuté v podadresári `dist` adresára `server`. V prípade záujmu o spustenie originálnych skriptov Vue.js vo vývojárskom móde je potrebné nainštalovať `Node.js`, `npm` a všetky závislosti spomenuté v A.1 ako lokálne balíčky.

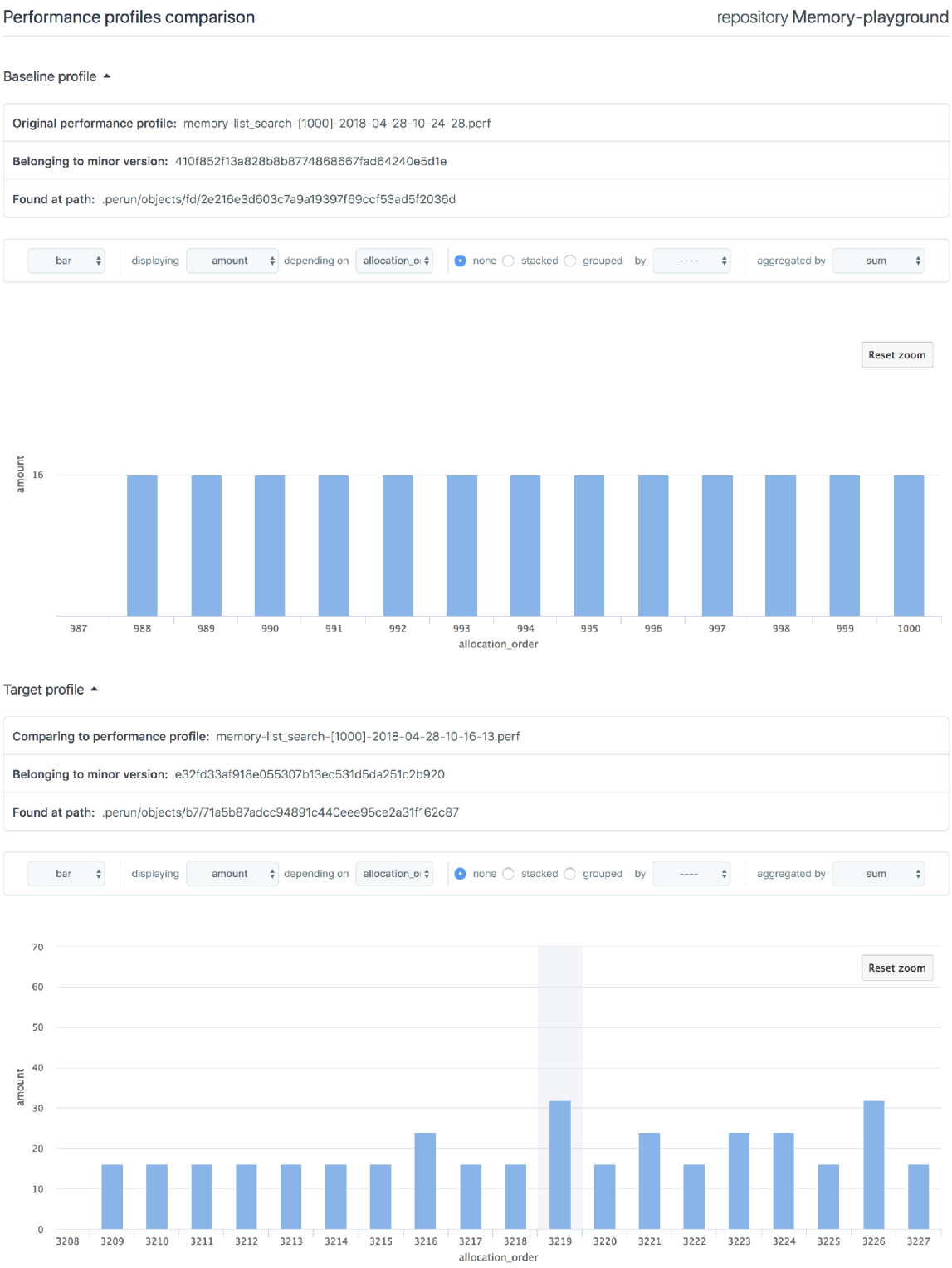
Príloha D

Grafické výstupy experimentálneho overenia

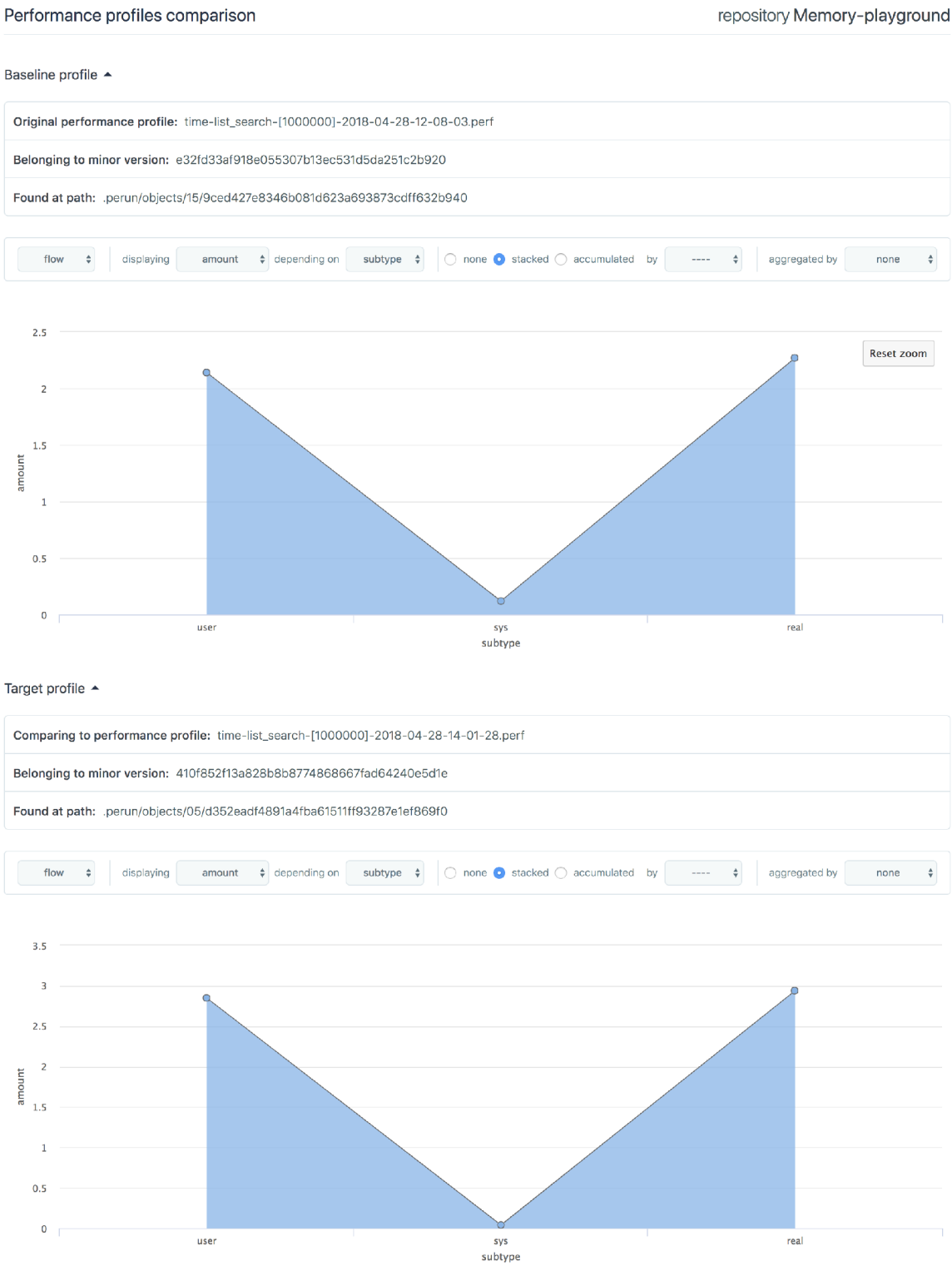
D.1 Demonštrácia porovnania dvoch výkonnostných profilov



Obrázok D.1: Porovnanie dvoch výkonnostných profilov — grafy zobrazujú počet bitov alokovaných počas behu profilovaného programu.



Obrázok D.2: Porovnanie dvoch výkonnostných profilov — grafy zobrazujú počet alokácií po behu profilovaného programu.

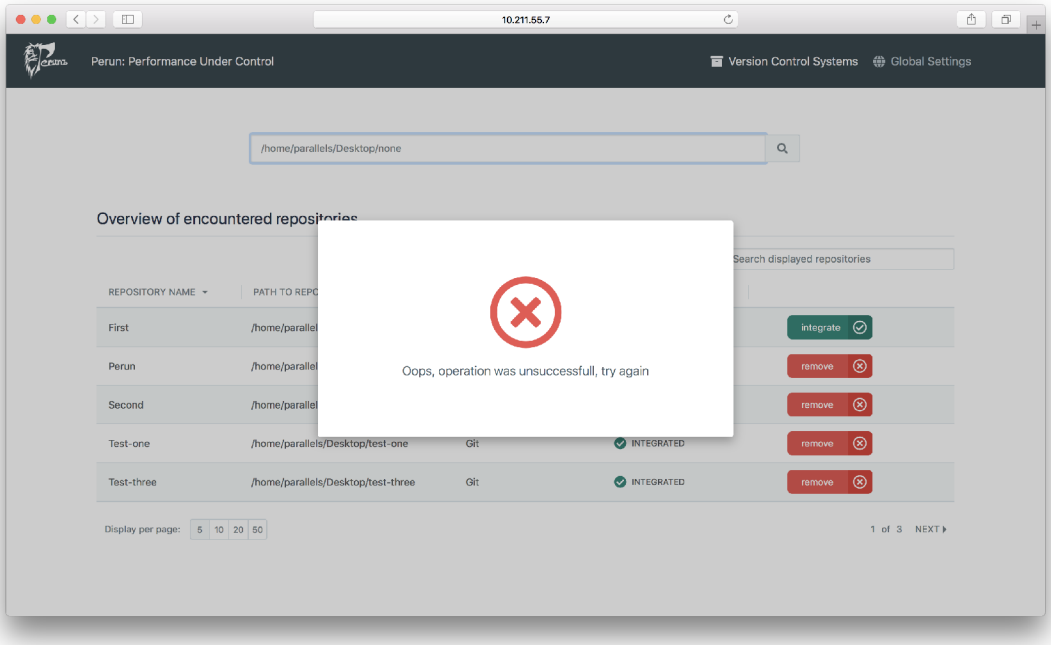


Obrázok D.3: Porovnanie dvoch výkonnostných profilov — grafy zobrazujú procesorový čas využitý pre beh profilovaného programu.

Príloha E

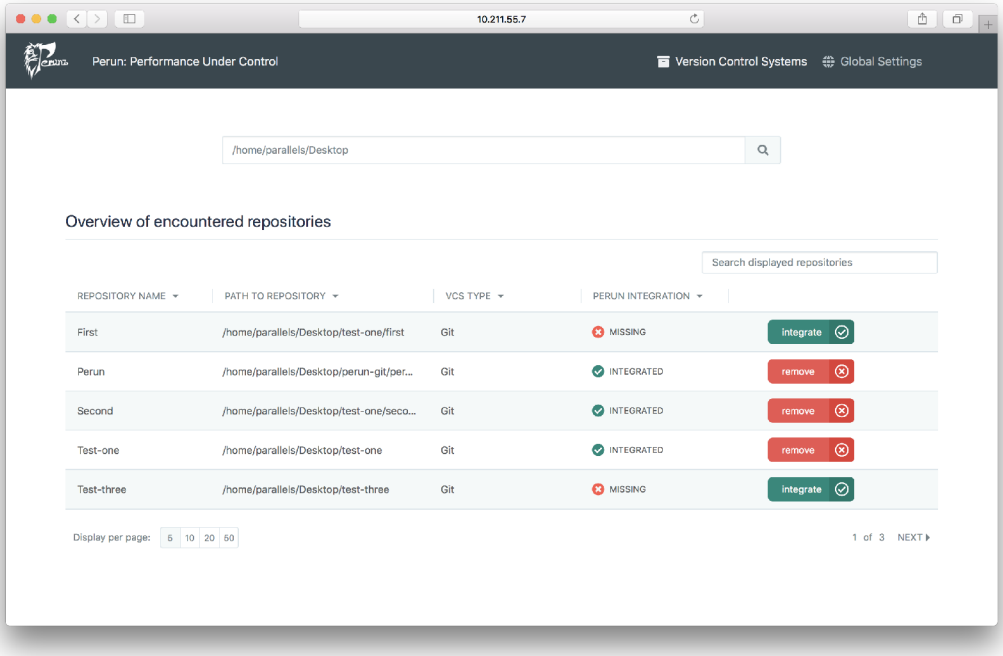
Priebeh testovania integrácie nástroja *Perun*

1. Neexistujúca cesta. Výstup rozhrania po zadaní neexistujúcej cesty. Repozitáre prípadne načítané v predošlom vyhľadávaní ostávajú v zozname, avšak žiadne nové repozitáre pridané nie sú.



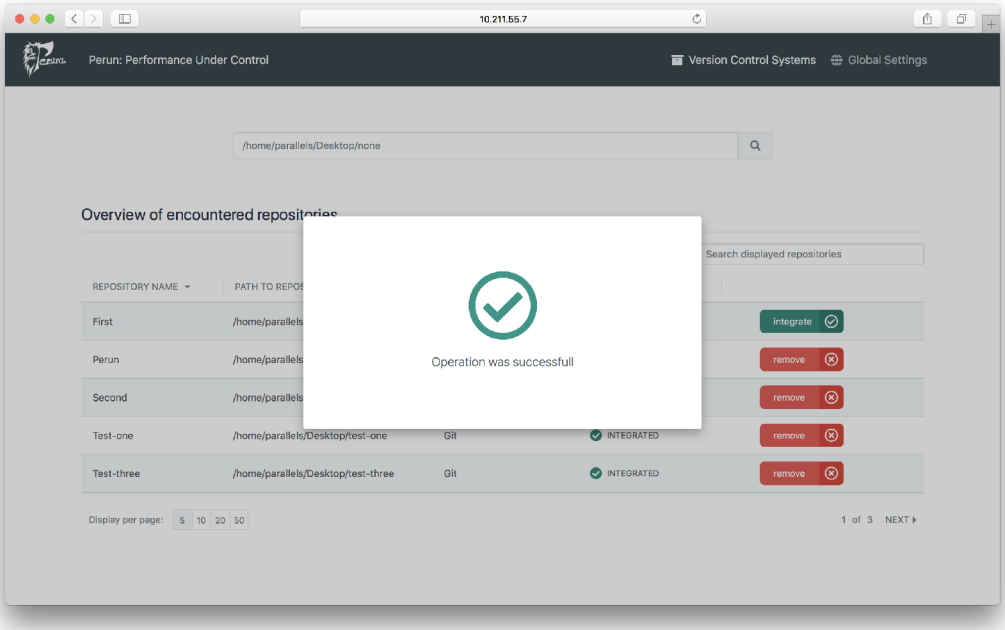
Obrázok E.1: Modálne okno rozhrania hlásiace neúspech operácie.

2. Správne zadaná cesta. Výstup rozhrania po zadaní správnej cesty, rozhranie načítalo zoznam repozitárov dostupných v adresári a jeho podadresároch.



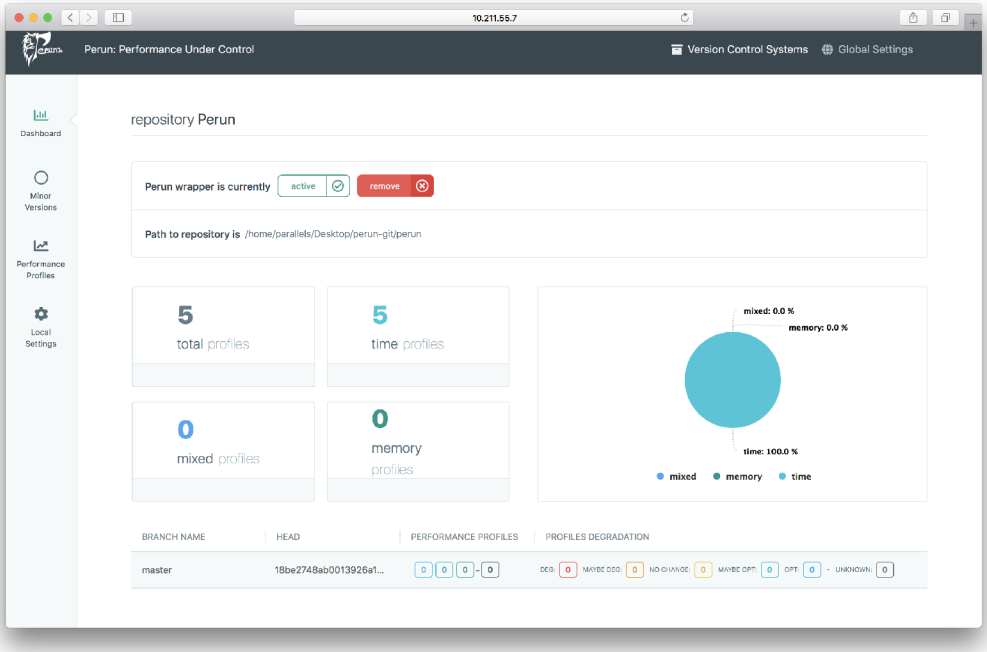
Obrázok E.2: Zoznam načítaných repozitárov.

3. Vytvorenie instancie *Perun*. Výstup rozhrania po úspešnom vytvorení instancie *Perun* nad vybraným repozitárom.



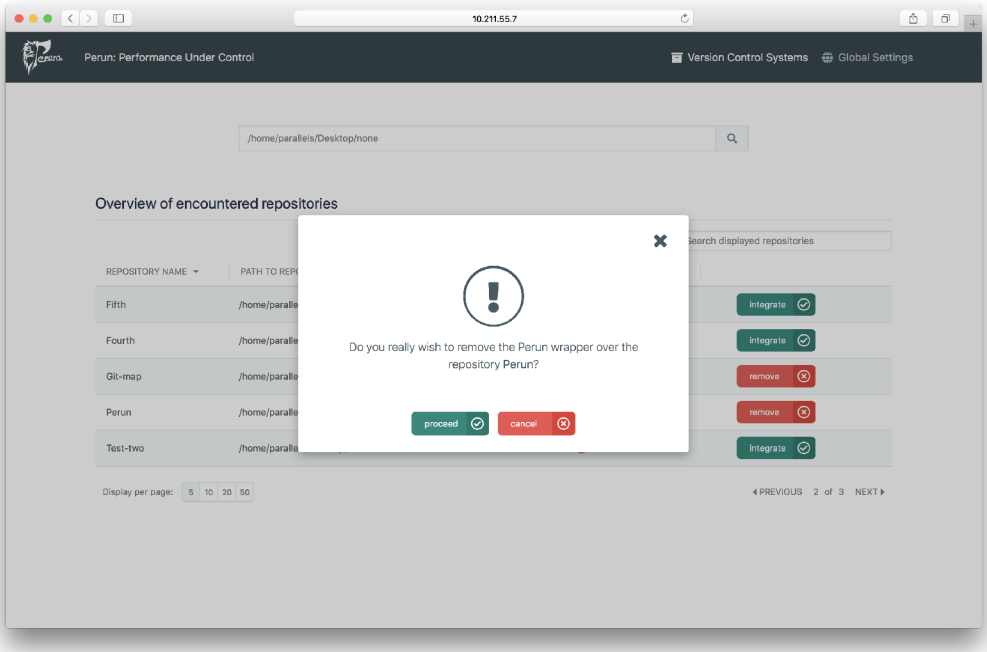
Obrázok E.3: Modálne okno hlásiace úspech operácie vytvorenia instancie.

4. Nástenka repozitára. Výstup rozhrania po vybratí konkrétneho repozitára s instancou *Perun* zo zoznamu.



Obrázok E.4: Nástenka vybraného repozitára.

5. Mazanie instancie *Perun* Výstup rozhrania po zvolení možnosti zmazania instancie *Perun*, zobrazené modálne okno pýta potvrdenie užívateľa pre vykonanie akcie.



Obrázok E.5: Modálne okno pýtajúce potvrdenie užívateľa pre vykonanie operácie.