Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Mikrobenchmarky Javy

Plzeň, 2019 Marek Rojík

PROHLÁŠENÍ

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 15. května 2019 . . . . . .

Marek Rojík

ABSTRACT

The goal of this thesis is to create an application for microbenchmark executing in the Java programming language. The application compares code snippets to determine which code snippet is executed the fastest. In the first part, there are described available tools that are used to create a microbenchmark. Out of the available tools, it is determined which tool meets the defined criteria. In the next part of the thesis, it is decided whether to implement an online or offline application. According to the analysis, the online version of the application is created. The application used the Java Microbenchmark Harness framework. In the last part of the thesis, the application is thoroughly tested.

KEYWORDS

microbenchmark, Java Microbenchmark Harness, Java, online application

ABSTRAKT

Cílem práce je vytvoření aplikace pro provedení mikrobechmarku v programovacím jazyce Java. Aplikace porovnává fragmenty kódu a zjišťuje, jaký fragment je proveden nejrychleji. V první části jsou popsány dostupné nástroje, které slouží pro vytvoření mikrobenchmarku v jazyce Java. Z dostupných nástrojů je vybrán nástroj splňující definovaná kritéria. V další části práce je provedena analýza, zda se bude implementovat online, nebo offline aplikace. Na základě analýzy je vytvořena online aplikace pro provádění mikrobenchmarku pomocí nástroje Java Microbenchmark Harness. V poslední části práce je vytvořená aplikace detailně otestována.

KLÍČOVÁ SLOVA

mikrobenchmark, Java Microbenchmark Harness, Java, online aplikace

ROJÍK, Marek *Mikrobenchmarky Javy*: diplomová práce. Plzeň: Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky, 2019. 89 s. Vedoucí práce byl Ing. Richard Lipka, Ph.D.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Richardu Lipkovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

OBSAH

1 ÚVOD

Cílem diplomové práce je vytvoření aplikace pro provádění mikrobenchmarků v programovacím jazyce Java. Aplikace porovná zadané fragmenty kódu a zjistí, jaký fragment je proveden nejrychleji. Použití aplikace by mělo být pro uživatele co nejjednodušší. Uživatel pouze vloží dva nebo více fragmentů kódu, aplikace fragmenty kódu porovná a sdělí uživateli výsledek měření. Uživatel proto nemusí vědět, jakým způsobem lze vytvořit a spustit mikrobenchmark.

Benchmark je obecný termín používaný v různých odvětvích. V IT světě se benchmarky používají pro měření výkonnosti systému či aplikace.

V současné době existuje několik aplikací, které slouží pro porovnání zadaných fragmentů kódu. Tyto aplikace jsou schopny porovnat kód například v programovacím jazyce C++ nebo Javascript. Pro jazyk Java není veřejně dostupná žádná podobná aplikace. V jazyce Javascript je jednodušší provést mikrobenchmark než v jazyce Java. Fragmenty Javascript kódu není potřeba kompilovat a jsou provedeny přímo v internetovém prohlížeči uživatele. Provést mikrobenchmark v programovacím jazyce Java je mnohem obtížnější. Nejprve je potřeba mikrobenchmark zkompilovat a následně spustit. Nicméně i když je mikrobenchmark zkompilovaný, v průběhu měření může být vykonávaný kód optimalizován, čímž dojde ke změně v již zkompilovaném mikrobenchmarku.

Z tohoto důvodu pravděpodobně neexistuje žádná uživatelsky přívětivá aplikace, která by nabízela porovnání různých fragmentů kódu v programovacím jazyce Java. Existuje ale několik nástrojů, jimiž lze vytvořit a spustit mikrobenchmark. K použití takového nástroje musí uživatel již vědět, jakým způsobem se nástroj používá. V první řadě musí uživatel vytvořit mikrobenchark s fragmenty kódu pro změření. Dále musí podle dokumentace zjistit, jaké je vhodné nastavení nástroje pro provedení mikrobenchmarku. Nástroje mohou nabízet provedení mikrobenchmarku v různých režimech. Pokud chce uživatel například zjistit, v jaké kolekci se rychleji získá prvek na daném indexu, zabere mu vytvoření mikrobenchmarku mnoho času.

Implementovaná aplikace odstíní uživatele od všech výše uvedených problémů. Aplikace pouze přijme požadavek na provedení zadaných fragmentů kódu. Po skončení běhu aplikace zobrazí uživateli přehledně naměřené výsledky. Tímto odpadá uživateli nutnost učit se, jakým způsobem psát mikrobenchmarky.

V programovacím jazyce Java je možné naimplementovat metodu s určitou logikou několika způsoby. Jazyk Java poskytuje různé typy kolekcí. Každý typ kolekce má jiné vlastnosti, jako například rychlost získání nebo vložení nové hodnoty. Pokud má programátor za úkol napsat metodu s důrazem na rychlost a řeší, jaký typ kolekce je nejvhodnější pro její implementaci, může využít vytvořenou aplikaci k rychlému a pohodlnému porovnání jednotlivých typů kolekcí. Aplikace ušetří programátorovi čas, který by strávil s implementací mikrobenchmarku.

2 METODY A POSTUPY MĚŘENÍ

2.1 Benchmark

Benchmark je proces měření a porovnávání výkonnosti systému, programu nebo různých operací. Termínem benchmark se nejčastěji označuje program pro testování výkonu aplikace. Tímto termínem lze označit i naměřené výsledky. Jako benchmark může být použita speciálně připravená aplikace, která má být nasazena na testovací systém.

V IT světě se benchmarky používají především pro měření výkonnosti systému či aplikace. Na základě výsledků výkonnosti lze například zjistit, jaký systém je nejlepší pro určité potřeby, jaká konfigurace systému je pro daný systém optimální, kolik uživatelů systém zvládne obsloužit, co se stane když systém selže atd.

Benchmarkem můžeme například měřit, kolik operací je potřeba k dosažení výsledku, dostupnosti a průchodnosti systému nebo náklady dané akce systému.

2.1.1 Základní vlastnosti

Každý benchmark musí mít určité vlastnosti, aby bylo možné jeho výsledek považovat za validní. Pokud bude alespoň jedna vlastnost porušena, výsledek benchmarku může být považován za nepřesný.

Jedna z hlavních vlastností je izolace. Benchmark má být vytvořen tak, aby se vždy měřilo jen to, co je cílem měření. Benchmark by měl běžet v prostředí, kde neběží žádné zbytečné procesy, které by se mohly střídat s benchmarkem o výpočetní výkon systému.

Jedna z dalších vlastností správného benchmarku je ta, že je výsledek měřitelný. Je potřeba dávat pozor na to, jestli se skutečně měří to, co je cílem benchmarku.

Další vlastností je opakovatelnost. Aby bylo možné benchmark opakovat, je nutné mít přesně definovaný výchozí stav. Dále je potřeba zvolit vstupní parametry tak, aby byl benchmark prováděn vždy se stejnou zátěží. Nicméně porušení této vlastnosti nemusí nutně vést k znehodnocení výsledku benchmarku.

2.2 Příprava benchmarku

2.2.1 Určení cílů

Určení cílů je v benchmarkování jedna z důležitých prvotních fází návrhu, která nesmí být podceněna. V této fázi je nutné zanalyzovat, jaké prostředky a postupy budou použity v samotném benchmarku. Běžného uživatele nezajímá kolik procent CPU spotřebuje spuštění internetového prohlížeče. Uživatele spíše zajímá, jestli bude čekat na otevření prohlížeče

2 nebo 10 sekund. Dalším příkladem cíle může být porovnání, zda je algoritmus A rychlejší než B, nebo jestli hardware od firmy A je vhodnější pro řízení robota než hardware od firmy B.

2.2.2 Analýza služeb

Při návrhu benchmarku je zapotřebí promyslet, jaké služby jsou systémem poskytovány a dále jaké mohou mít výsledky. Na základě určených služeb lze zvolit správnou metriku. V analýze by se měly objevit pouze důležité služby. Pokud bude analýza obsahovat i minoritní služby, bude mnohem složitější příprava testů, jelikož testy budou komplexnější. Součástí analýzy jsou i možné výsledky jednotlivých služeb. Služba může mít teoreticky nekonečně mnoho výsledků, které lze rozdělit do různých skupin, jako například funguje nebo nefunguje. Poslední částí analýzy služeb je příprava zátěže jednotlivých služeb.

2.2.3 Metriky

Pro zobrazení výsledku benchmarku existuje několik různých metrik. Každá metrika prezentuje výsledek z jiného úhlu pohledu. Metriky bývají založené na čase, výkonu a spolehlivosti. Může se zdát, že benchmark poskytuje kvalitní výsledky, pokud je zvolena nesprávná metrika, benchmark nemá správnou vypovídající hodnotu. Nesprávně zvolených metrik mohou využívat například prodejci, kteří se snaží co nejlépe prezentovat daný produkt. Použitím nesprávné metriky mohou oklamat jejich zákazníky, kteří vidí kvalitní výsledek měření, nicméně je nesprávně zvolená metrika. Zákazník vidí číslo představující výsledek měření, nicméně již se nedozví žádné informace o tom, jak měření produktu probíhalo, a proto je prezentované číslo zavádějící.

Zvolit správnou metriku pro konkrétní benchmark není vždy jednoduché rozhodnutí. Výsledek benchmarkovaných služeb je možné rozdělit do tří kategorií - úspěšně zpracované, chybně zpracované a nepřijaté. Pokud benchmark dopadne úspěšně, uživatele bude pravděpodobně zajímat, jak dlouhou dobu systém potřeboval k úspěšnému zpracování požadavku, kolik požadavků je schopen systém úspěšně obsloužit za určitou dobu nebo jaké množství zdrojů systém spotřeboval. Může nastat situace, že se systému nepodaří zpracovat požadavek. V takovém případě může uživatele zajímat doba mezi chybami, popřípadě pravděpodobnost výskytu chyby. V poslední řadě nemusí vůbec přijmout požadavek na zpracování. V takovém případě je vhodnou metrikou doba, po kterou systém odmítá přijmout daný požadavek, nebo za jakých příčin systém požadavky nepřijímá.

Metriky zaměřené na čas

Jako časovou metriku lze označit různé metriky, které měří určitou dobu trvání. Jedná se především o měření doby odezvy, reakční doby, doby do poruchy, atd. Doba odezvy

znamená čas, který uplyne mezi odesláním požadavku uživatelem a dokončením zpracování požadavku systémem. Reakční doba znamená čas, který uplyne mezi odesláním požadavku uživatelem a přijetím požadavku systémem.

Metriky zaměřené na kapacitu

Jedna z možných hladin metriky se nazývá *nominální* (maximální) kapacita. Tato kapacita značí maximální možný počet požadavků, které je systém schopen obsloužit v ideálních podmínkách. Nominální kapacita se dá považovat spíše za teoretickou kapacitu, jelikož této kapacity není možné v reálném světě dosáhnout.

Hodnota metriky, která má lepší vypovídající hodnotu v běžných podmínkách systému, se jmenuje *využitelná* kapacita. Jedná se o maximální počet požadavků, které je systém schopen obsloužit v požadované době odezvy.

Poslední hodnotou metriky zaměřené na kapacitu je *zlomová* kapacita. Zlomová kapacita označuje maximální počet požadavků, které je systém schopen obsloužit ve velmi krátké až neznatelné době. Určení maximální doby obsluhy u využitelné a zlomové kapacity není možné dopočítat. Tyto časy musí definovat samotní uživatelé, kteří budou se systémem pracovat, jako požadavek na systém.

Metriky zaměřené na výkon

Metriky zaměřené na výkon se používají pro měření především doby zpracování, zrychlení a průtoku. Měření doby zpracování nebo doby obrátky se využívá hlavně pro dávkové zpracování úloh. Dobou zpracování se rozumí čas, který uplyne od zadání úlohy k výpočtu až k vracení konečného výsledku.

Zrychlení se používá v souvislosti s paralelizmem. Jedná se o poměr doby zpracování paralelizované a neparalelizované úlohy, respektive poměr doby zpracování úlohy na více procesorech oproti provedení úlohy na jednom procesoru. Na základě vypočítaného zrychlení lze usoudit, o jakou dobu zrychlila paralelizace prováděný výpočet.

Jako poslední příklad použití výkonové metriky je měření průtoku. Pomocí měření lze zjistit, kolik požadavků je systém schopen obsloužit za určitou jednotku času. Tato metrika uvádí výsledky benchmarků v jednotkách, jako jsou například MIPS[1], FLOPS[2], transakce za vteřinu a další.

Metriky zaměřené na spolehlivost

Mezi metriky zaměřené na spolehlivost patří například metrika střední doby mezi chybami, střední doby do selhání, střední doby do opravy nebo pravděpodobnosti selhání. Střední

[1] Million Instructions Per Second

[2] Floating point operations per second

doba mezi chybami značí dobu, která uplyne mezi první a druhou poruchou. Střední doba do selhání znamená dobu, za jakou se v systému vyskytne chyba. Poslední zmíněná metrika je střední doba do opravy. Metrika říká, za jakou dobu je systém uveden do funkčního stavu z předchozího selhání. Bohužel bývá tato metrika často podceněna. Zvláště u služeb nabízejících důležité funkce (bankovnictví, letadla, atd.) je naprosto zásadní [9].

2.3 Zátěž měření

2.3.1 Parametry a faktory

Definování parametrů je nezbytná část při přípravě analytického modelu nebo simulace. Parametry lze rozdělit do dvou kategorií. První kategorií jsou parametry systému, které popisují systém, jsou neměnné od začátku běhu až po jeho dokončení a jsou totožné i v rámci různých instalací systému. Příkladem parametru systému může být například hardware aplikačního serveru, počet povolených připojení k databázovému serveru nebo přenosová kapacita sítě.

Druhou kategorií jsou parametry zátěže. Tyto parametry charakterizují požadavky a chování uživatelů systému. Parametry zátěže se mohou lišit v průběhu běhu systému nebo v různých instalacích, což je hlavní rozdíl oproti parametrům systému. Parametry se mění na základě využití systému.

Faktory jsou nejdůležitější parametry systému, které se mohou měnit v čase. Faktor je parametr, který vysoce ovlivňuje výkon systému. V ideálním světě by byl v rámci benchmarku vybrán jeden faktor, se kterým by se pracovalo. V reálném benchmarku existuje vždy několik vlivů, které nelze přerušit během běhu měření, jako jsou například okolní běžící procesy, optimalizátory a dále. Benchmark je doporučováno provést s různými hodnotami, tzv. hladiny faktoru. V průběhu benchmarku je snaha provádět měření pro konstantní hladinu faktorů. Pro ověření vlivu chování měřené hladiny faktorů na chod systému je doporučováno provést opakované měření s odlišnou hladinou faktorů. Tyto hodnoty faktoru by měly nabývat typických hraničních hodnot, jako jsou například hodnoty s minimální nebo obvyklou zátěží sytému a dále hodnoty, které simulují přetížení systému.

2.3.2 Příprava zátěže

Pokud má být výsledek benchmarku použitelný, je potřeba použít správně nadefinovanou zátěž. Zátěž by měla odpovídat reálným požadavkům, aby se měřený systém choval maximálně přesně jako se chová reálný systém. Pokud bude zátěž špatně nadefinovaná, systém se může chovat jinak a výsledek měření je nepoužitelný. Pokud slouží benchmark pro porovnání výkonu různých systémů, musí být zátěž opakovatelná na různých zařízeních. Na základě opakovatelnosti lze porovnávat jednotlivé výkony různých systémů.

Při přípravě zátěže je dobré znát, jaké jsou typické požadavky se kterými systém pracuje. Dále je potřeba mít povědomí o počtu požadavků, které systém přijme za určitou jednotku času. Důležitou informací při přípravě zátěže je znalost, jak může být náročné zpracování požadavků. Každý systém má určitou maximální hranici požadavků, které je schopen obsloužit. Na základě této informace lze nastavit maximální hraniční omezení zátěže.

2.3.3 Druhy zátěže

Při benchmarkování se mohou použít celkem tři postupy měření (viz kapitola 2.5.2). Pro každý postup měření je nezbytné definovat zátěž rozdílným způsobem. Pro analytický model může být zátěž definována pravděpodobnostním rozdělením a jeho charakteristikami.

Pro simulaci se mohou použít získané požadavky od uživatelů. Jednou z nejjednodušších technik je použití logu z reálného systému, který obsahuje dostatek informací k vytvoření totožné simulace. Dále lze použít i pravděpodobnostní rozdělení.

Pro měření lze zátěž získat měřením výkonu reálného systému. K získání zátěže lze využít skripty napodobující chování skutečného uživatele, vzorové problémy, logy požadavků a nebo přímo testery.

2.3.4 Reálná a syntetická zátěž

Zátěže je dále možné rozdělit na dva druhy. První možnost se nazývá reálná zátěž. Tato zátěž vychází ze zachyceného běhu reálného systému. Jelikož se jedná o záznam běhu skutečného systému, není vhodnou volbou použít reálnou zátěž při začátku vývoje systému. K zachycení skutečného běhu je potřeba, aby systém již obsahoval důležitou funkcionalitu a byl využíván.

Druhou možností je syntetická zátěž. Jedná se o pokus vytvořit zátěž podobající se reálné zátěži. Syntetická zátěž je snáze kontrolovatelná než reálná zátěž. Další výhodou je její velikost. Soubor popisující syntetickou zátěž je mnohem menší než například log obsahující záznam chování reálného systému. Za další výhodu lze považovat snadnou modifikovatelnost, díky které lze zátěž použít na různých systémech. Bohužel může nastat situace, že se syntetická zátěž bude chovat v mnoha ohledech jinak než reálný systém. Na základě odlišného chování zátěže není možné považovat výsledek benchmarku za věrohodný.

V dnešní době se používají jako jedny z jednoduchých zátěží pro benchmarky CPU tzv. instrukční mixy. Jedná se přímo o počítačové instrukce, které daný program vykonává v rámci zpracování požadavku. Statistickou analýzou lze zjistit nejčastější provedené instrukce, díky kterém je možné vytvořit podobný program. Dalším příkladem je použití

jádra reálného systému. Výhodou tohoto přístupu je získání základních algoritmů reálného programu. Jako nevýhodu zmíněných přístupů lze považovat nevyužití I/O operací, což může značně odchýlit vytvářenou zátěž od reálné zátěže.

2.4 Analýza výsledků

Analýza výsledků je další z obtížnějších fází měření systému. K porozumění správnosti a významu benchmarku se používají statistické pojmy. Nejčastěji se pro analýzu výsledku benchmarku používá průměr z jednotlivých naměřených hodnot. Kdyby se výsledek analyzoval jen na základě průměru, v mnoha případech by mohly být různé benchmarky vyhodnoceny stejně i přes to, že mají odlišné hodnoty, jelikož výsledný průměr je podobný. Výsledek je vhodné popsat jako rozdělení. Druhým způsobem pro popsání výsledku benchmarku je uvedení několika statistických hodnot, jako je například střední hodnota, průměr, medián, percentily, rozptyl nebo odchylka. Při prezentaci naměřeného výsledku je doporučováno uvést i zvolený faktor [9].

2.5 Benchmarkování hardware

2.5.1 Základní dělení

Benchmarky hardwaru lze rozdělit na několik typů.

Reálné programy

Reálné programy jsou prvním typem benchmarků. Jedná se o programy, které jsou nejvíce podobné tomu, co se má skutečně měřit. U tohoto typu je složitější příprava měření, volba zátěže a volba správné metriky. Problém nastává u srovnání více testovaných systémů, neboť je složité zajistit u každého systému stejné podmínky.

Jádro aplikace

Existuje typ benchmarku, který měří samotné klíčové části aplikace, tzv. jádro aplikace. Tento typ benchmarku se nazývá kernel benchmark. Jeden z populárních benchmarků se jmenuje Livermore loop[3], který slouží pro měření paralelních výpočtů.

Syntetický benchmark

Dalším typem pro měření výkonu hardwarových komponent počítače jsou tzv. syntetické benchmarky. Tento typ benchmarku spustí sérii testů v určitém pořadí, na základě kterých

[3] https://en.wikipedia.org/wiki/Livermore_loops

je schopen změřit výkon počítače nebo jeho hardwarových komponent. Tyto testy jsou navrženy tak, aby měly snadno opakovatelné výsledky pro přesné srovnání. Díky tomuto přístupu je možné otestovat jednotlivé hardwarové komponenty nezávisle na sobě (například SSD disk a procesor). Nejrozšířenější programy pro syntetické benchmarkování jsou 3DMark[4] nebo PCMark8[5].

Real World benchmark

Real World benchmark používá k měření výkonu počítače programy, které provádějí například složitou výpočetní úlohu, vykreslování 3D modelu atd. Z těchto programů získá statistiky, na základě kterých je schopen vyhodnotit celkový výkon počítače. Výhodou benchmarku je získání výkonu počítače jako celku pro danou spuštěnou úlohu. Naopak nevýhodou je nemožnost změřit výkon jednotlivých hardwarových komponent, protože výsledné statistiky charakterizují počítač jako celek.

Syntetický a Real World benchmark slouží pro změření výkonu počítače. Bohužel není možné jednoznačně určit, který typ benchmarku je vhodnější pro změření výkonu, protože oba typy poskytují užitečné informace získané odlišným způsobem [5].

2.5.2 Postupy měření

Existují celkem tři základní postupy, pomocí kterých je možné změřit nebo odhadnout výkon systému nebo aplikace. Liší se použitou úrovní abstrakce a náročností na vytvoření. Je doporučováno použít vždy minimálně dva z níže uvedených postupů, aby se dalo výsledek měření považovat za věrohodný. Pokud by nastala situace, kdy by se naměřené hodnoty reálným benchmarkem vysoce lišily, například s výsledky analytického modelu, je možné, že benchmark měří odlišnou funkcionalitu než se skutečně očekává.

Analytický model

Analytický model je vhodný pro použití za předpokladu, že existuje přesná představa o systému, který je modelován. Analytický model pouze odhaduje možné výsledky. Ve většině případů se jedná o nejlevnější řešení. Obtížnost vytvoření analytického modelu závisí na tom, jak složité je najít výsledný model. Model se dá využít i ve fázi návrhu, neboť není potřeba mít k dispozici reálný systém. Modelování systému je možné považovat za nejrychlejší a nejlevnější dostupné řešení. Nedostatkem je nutnost mít jednoduchý analytický model, což může ovlivnit jeho výslednou hodnotu.

[4] https://benchmarks.ul.com/3dmark

[5] https://benchmarks.ul.com/pcmark8

Simulace

Podobně jako u analytického modelu, simulace nevyžaduje použití reálného systému pro potřeby měření. Na rozdíl od analytického modelu je navržení simulace obtížnější, protože je náročnější provést kalibraci systému. Kalibrace slouží k nastavení co nejvěrohodnějšího chování modelovaného systému. Může se jednat například o nastavení doby zpracování různých požadavků tak, aby se co nejvíce podobaly reálnému systému. Pokud je kalibrace špatně provedená, získané výsledky neodpovídají reálnému chování systému a není možné tento způsob měření považovat za důvěryhodný. Tvorba simulace je náročnější než vytvoření analytického modelu.

Měření

Měření se odlišuje od výše zmíněných postupů tím, že testovací programy jsou spuštěny přímo na skutečném zařízení, kde probíhá reálné měření. To znamená, že pro získání co nejpřesnějších výsledků měření je nutné, aby se testy prováděly na reálném testovaném systému. Tento způsob měření je ze všech uvedených ten nejsložitější na přípravu. Díky složité přípravě jsou získané výsledky reálné, pokud je spuštěn vhodný testovací program. Nejlepším benchmarkem je sledování programu, který běží na skutečném hardware a je reálně používán [9].

2.6 Benchmarkování software

2.6.1 Druhy benchmarků

Benchmarky softwaru se dají rozdělit do několika kategorií podle komplexnosti měřené aplikace.

Makrobenchmark

První typ se označuje termínem *makrobenchmark*. Tento typ slouží pro měření výkonnosti aplikace jako celku, například jak dlouho trvá zpracování požadavku aplikací. Tento typ měření je podobný end-to-end[6] testům.

Mesobenchmark

Další typ měření se nazývá *mesobenchmark*. Typ reprezentuje měření kompletních akcí vztahujících se k větším částem aplikace. Jedná se o akce, které provádějí skutečnou významovou práci, nicméně nejedná se o kompletní plnohodnotnou aplikaci. Jako příklad

[6]End-to-end znamená testy kompletních procesů a životních cyklů produktu

mesobenchmarků lze uvést autentizace či autorizace, čtení dat z databáze nebo volání externích služeb aplikace. Rozsah mesobenchmarku lze přirovnat k integračnímu testu. Mesobenchmark lze zařadit na základě komplexnosti mezi makrobenchmark a mikrobenchmark.

Mikrobenchmark

Třetí typ měření je označen termínem *mikrobenchmark*. Typ označuje měření výkonnosti „malých“ akcí aplikace. Jedná se o měření samostatných metod aplikace nebo malých částí aplikační logiky, které ke svému provedení nepotřebují volání dalších metod aplikace či externích služeb. Pomocí mikrobenchmarku lze například porovnat několik implementací se stejnou logikou a na základě výsledku vybrat nejrychlejší implementaci. Příkladem mikrobenchmarku může být měření doby seřazení prvků v poli podle hodnoty nebo provedení složitého výpočtu. Jeho rozsah lze přirovnat k jednotkovému testu. Na základě mikrobenchmarku identifikovat případné nedostatky v podezřelé oblasti kódu při optimalizaci výkonu.

Mikrobenchmark se skládá z několika důležitých fází. Jednotlivé fáze jsou: návrh, implementace, spuštění a vyhodnocení naměřených hodnot. Každá z fází je kritická a i kvůli malému nedostatku může měření výkonnosti skončit nepřesným výsledkem [14].

3 BENCHMARKOVÁNÍ V JAVĚ

Programy implementované v programovacím jazyce Java nejsou příliš vhodné pro vytváření mikrobenchmarků. Vytvoření kvalitního benchmarku v jazyce Java je velice obtížné, protože běžící kód může ovlivnit mnoho různých faktorů, se kterými by měl být autor benchmarku obeznámen.

3.1 Java Virtual Machine

Java Virtual Machine (JVM) je základním kamenem platformy Java. JVM je virtuální počítač, který umožňuje spouštět programy implementované v programovacím jazyce Java. Úkolem JVM je zpracovat pouze tzv. mezikód, který je v Javě označován jako Java bytecode. JVM pracuje pouze se zkompilovaným mezikódem, neví nic o programovacím jazyku Java. Výhodou použití JVM je možnost spustit zkompilovaný mezikód na libovolném systému (např. Windows, Linux), na kterém je k dispozici JVM. Na základě této vlastnosti lze spustit i programy implementované v jiném programovacím jazyce, pokud je lze zkompilovat do Java bytecode. Samotný JVM bývá zpravidla implementovaný v programovacím jazyce C++ [11].

JVM spouští v průběhu vykonávání mezikódu i další procesy. Jedná se o procesy sloužící k optimalizaci běžícího programu. Různé typy optimalizací jsou popsány v kapitole 3.2. Mezi další procesy, které se spouštějí nezávisle na běhu vykonávaného programu, patří proces garbage collection popsaný v kapitole 3.4.

3.2 Dynamická kompilace

Kompilace programů implementovaných v programovacím jazyce Java je odlišný proces než kompilace programů implementovaných ve staticky kompilovaných programovacích jazycích. Výstupem statického kompilátoru je převedený zdrojový kód přímo do bytecode.

Kompilátor programovacího jazyka Java převede zdrojový kód do přenosného JVM bytecode neboli mezikódu. Jedná se o „virtuální strojové instrukce“, které jsou provedeny ve speciálním programu JVM. Na rozdíl od kompilátorů jiných programovacích jazyků, kompilátor jazyka Java provede v průběhu kompilace velice málo optimalizací [3]. Optimalizace je následně provedena v průběhu vykonávání běhu programu.

První generace JVM byla zcela interpretována. JVM přímo interpretoval mezikód. Tato generace JVM poskytovala velice pomalý výkon pro běžící Java programy. Systém spotřeboval mnohem více času interpretováním mezikódu než program, který měl běžet. Jednalo se o prvotní návrh, jak by mohl JVM fungovat [3].

3.2.1 Just-in-time kompilace

V roce 1998 byla přidána do JVM kompilace type Just-in-time (JIT). JIT nekompiluje kompletní mezikód do strojových instrukcí najednou, ale kompiluje vždy konkrétní část mezikódu, když zjistí, že tato část bude opakovaně provedena. Na základě tohoto přístupu postupné kompilace byl vytvořen i název. JIT přístup umožňuje rychlejší spuštění programu, protože není nutné interpretovat přeložené části, ale přímo je vykonat na CPU.

Tento návrh kompilace se zdál být dostatečný, ale bohužel obsahoval nedostatky. JIT kompilace odstranila nedostatek z první generace Java kompilátorů. JIT kompilace neoptimalizovala mezikód dostatečně. Aby se vyhnulo zbytečně dlouhému startu Java aplikace, JIT kompilátor musel být velice rychlý. Z tohoto důvodu nebylo příliš mnoho prostoru na provedení hloubkové optimalizace prováděného mezikódu [3]. První JIT kompilátory byly opatrné při používání metody inlining, protože nevěděly, jaké třídy mohou být načteny později.

Technicky JIT zkompiluje část mezikódu těsně před jeho provedením, nicméně termín JIT je často používán pro jakékoliv zkompilování mezikódu do strojových instrukcí, dokonce i pro interpretování mezikódu [3].

3.2.2 HotSpot dynamická kompilace

HotSpot je označení pro konkrétní JVM, které je schopné provádět JIT kompilaci. Jedná se o open-source JVM vyvinutý firmou Oracle. HotSpot kombinuje interpretování, profilování a dynamickou kompilaci. Oproti JIT nekompiluje mezikód do strojových instrukcí před provedením dané části mezikódu, ale při prvním spuštění interpretuje mezikód přesně jako první generace JVM. V průběhu vykonávání programu shromažďuje profilovaná data o mezikódu, na základě kterých se rozhodne, jaké části mezikódu jsou prováděny frekventovaněji, aby se vyplatilo takové části zkompilovat a optimalizovat. Tyto nalezené části se označí jako „hot“.

Jelikož se kompiluje pouze často vykonávaný kód, neplýtvá se zbytečně procesorovým časem na kompilování částí mezikódu, které se vykonají například pouze jednou. JIT kompilátor má proto mnohem více času na důkladnou kompilaci „hot“ částí mezikódu, protože se budou tyto „hot“ části po optimalizaci rychleji zpracovávat. Díky získaným profilovaným datům se může kompilátor průběžně rozhodovat, jaké optimalizace budou nejlepší volbou pro daný „hot“ mezikód [3].

HotSpot přináší možnost použít dvě různé implementace JVM - *Client* a *Server*. Každá implementace obsahuje kompilátor s odlišnou prioritou. Client VM je optimalizován, aby bylo spuštění Java aplikace co nejrychlejší a využití paměti bylo co nejmenší. Client VM využívá méně komplexní optimalizace než Server VM, aby byla kompilace co nejrychlejší. Druhou implementací je Server VM, který obsahuje kompilátor pro důkladnější optima-

lizaci překládaného kódu. Na základě důkladnější kompilace je maximalizovaná rychlost aplikace. Nevýhodou je déle trvající spuštění Java aplikace. Server VM je určen pro dlouho běžící serverové aplikace. V dřívějších verzích používalo JVM ve výchozím stavu Client VM. Od Javy 7 se již používají Client VM a Server VM současně [12].

HotSpot Server VM umožňuje použít několik různých optimalizací. Mezi standardní optimalizace patří například code hoisting, common subexpression elimination, loop unrolling, range check elimination, dead-code elimination a data-flow analysis.

3.2.3 Průběžná rekompilace

Zajímavá vlastnost HotSpot přístupu je možnost zkompilovat v průběhu běhu aplikace určitou část mezikódu vícekrát. Pokud je interpretovaný kód proveden vícekrát, je zkompilován do strojových instrukcí. JVM pokračuje v profilování a na základě získaných dat může již zkompilovaný „hot“ mezikód zkompilovat znova s důraznější optimalizací nebo naopak zjistit, že optimalizace byla příliš agresivní. JVM může určitý „hot“ mezikód zkompilovat několikrát v průběhu jednoho běhu aplikace. Průběžná rekompilace je využívána v obou implementacích JVM (Client VM a Server VM) [12].

3.2.4 On-stack replacement

Původní verze HotSpot kompilovala frekventovaně zpracovávané metody. Pokud byla zpracována část mezikódu, kompilátor si vnitřně inkrementoval počet volání. Pokud došlo ke zkompilování zpracovávaného mezikódu, kompilátor v dané iteraci pokračoval v interpretování mezikódu a získané optimalizované strojové instrukce použil až v dalším volání. Teoreticky mohla nastat situace, že byl mezikód kompilován zbytečně, jelikož již nikdy nedošlo k jeho provedení. Například mohlo se jednat o počítačově náročný program, kde se všechny výpočty provádějí v jediném volání metody. Na základě profilovaných dat může kompilátor označit metodu jako „hot“ a zkompilovat ji. Jelikož ale program volá metodu pouze jednou, optimalizovaný zkompilovaný kód již nebyl nikdy použit.

Novější verze HotSpot kompilátoru již obsahují techniku nazývanou On-stack replacement (OSR), která již umožňuje využít zkompilovaný kód v průběhu iterace, ve které došlo ke kompilaci [3].

3.2.5 Dead-code elimination

Jedna z klíčových fází optimalizace je eliminace tzv. mrtvého kódu. Jedná se o kód, který je proveden, nicméně jeho provedení žádným způsobem neovlivní běh programu. Níže je ukázka metody, která obsahuje mrtvý kód.

```
1 public void doSomeStuff() {
2     int uselessSum = 0;
3     for (int i = 0; i < 1000; i++) {
4       uselessSum += i;
5     }
6 }
```

Ukázka 3.1: Metoda obsahující mrtvý kód

V ukázce 3.1 je implementovaná metoda `doSomeStuff()`. Metoda obsahuje cyklus, ve kterém se v každé iteraci provede součet dvou čísel. Kompilátor zjistí, že proměnná `uselessSum` není nikde použitá a vymaže tělo cyklu. Dále kompilátor vyhodnotí, že cyklus obsahuje prázdné tělo a vymaže i cyklus. Po optimalizaci má metoda `doSomeStuff()` pouze prázdné tělo.

Eliminace mrtvého kódu v kompilátoru typu *client* je méně důkladná než v typu *server*. Server kompilátor vyhodnotí, že metoda obsahuje mrtvý kód a kompletně vymaže její tělo. Client kompilátor nemusí vyhodnotit úplnou bezcennost těla metody.

Bohužel není možné vypnout eliminaci mrtvého kódu v průběhu kompilace. U benchmarkování je tato optimalizace spíše na škodu. Napsat kvalitní microbenchmark není vůbec jednoduchý úkol, protože měřené metody nemusí mít žádnou návratovou hodnotu. Z tohoto důvodu kompilátor vyhodnotí, že metoda obsahuje mrtvý kód a provede eliminaci, čímž zasáhne do implementace benchmarkované metody.

Problém eliminace mrtvého kódu v benchmarkování je i ve staticky kompilovaných jazycích. Bohužel v dynamicky kompilovaném kódu je složitější zjistit, že došlo k optimalizaci mrtvého kódu. Ve staticky kompilovaném jazyku je to jednodušší, protože je možné nahlédnout do vygenerovaných strojových instrukcí a zjistit, že chybí část instrukcí, které prováděly mrtvý kód [3].

3.2.6 Loop unrolling

Loop unrolling je jedna z optimalizací, která je prováděna pouze v JVM typu *Server VM*. Pointa optimalizace je ve snížení počtů iterací daného cyklu, aby se odstranila zbytečná složitost způsobená skoky CPU. Jelikož čím je méně skoků CPU, tím lépe fungují pipeline. Snížit počet iterací cyklu je relativně jednoduchý úkol pro kompilátor. Na základě redukování iterací cyklu dojde k postupnému naplnění těla cyklu přímo prováděným kódem. Lépe je funkčnost loop unrolling znázorněna níže v ukázkách 3.2.

```
1 for(int i = 0; i < N; i++) {
2     S(i);
3 }
```

Ukázka 3.2: Metoda obsahující neoptimalizovaný cyklus

Metoda již po optimalizaci loop unrolling je v ukázce 3.3.

```
for(int i = 0; i < N; i += 4) {
    S(i);
    S(i+1);
    S(i+2);
    S(i+3);
}
```

Ukázka 3.3: Metoda obsahující optimalizovaný cyklus

V těle metody je několikrát zduplikovaný první řádek obsahující volání metody `S()` s odlišnou hodnotou v parametru. Tímto krokem se zvětšila doba zpracování jedné iterace cyklu. Počet iterací je zredukován oproti původnímu počtu (`i += 4`). Cílem této optimalizace je zvýšit rychlost zpracování cyklu. Jelikož je zredukován počet iterací cyklu, je snížen i počet skoků ve strojových instrukcích. Provádění skoků stojí procesor mnoho času, proto loop unrolling zvýší celkovou rychlost zpracování cyklu [8].

3.2.7 Metoda inlining

Programátor se snaží implementovat metody s co nejmenším počtem řádků, aby byly co nejjednodušší a čitelné na první pohled. Bohužel tento přístup není příliš vhodný pro procesor vykonávající strojové instrukce. Pokud je procesor nucen během provádění kódu metody začít vykonávat jinou metodu, musí si uložit aktuální stav registrů, adresu kde bude pokračovat ve vykonávání a začít vykonávat zavolanou metodu. Tato operace přidává další režii.

Metoda inlining znamená nahrazení zavolání metody její implementací. V ukázce 3.4 je znázorněn příklad dvou metod, kde jedna metoda volá druhou metodu.

```
public static void main() {
  doSomeStuff();
}

public void doSomeStuff() {
  System.out.println("lorem ipsum");
}
```

Ukázka 3.4: Zavolání metody v jiné metodě

Kompilátor může tento kód zoptimalizovat způsobem v ukázce 3.5.

```
public static void main() {
  System.out.println("lorem ipsum");
}
```

Ukázka 3.5: Method inlining

Nahrazením volání metody `doSomeStuff()` její implementací dochází ke zredukování volání metod, což šetří čas procesoru, který může použít například právě pro vykonání zoptimalizované metody.

Tato optimalizace poskytuje ještě jedno důležité vylepšení. Metoda inlining produkuje mnohem větší bloky kódů pro zpracování, což může vést k provedení různých dalších

typů optimalizace, jako je například eliminace mrtvého kódu. Pro kompilátory je vždy efektivnější optimalizovat větší blok kódu než blok kódu s minimálním počtem řádek. Metoda inlining pomáhá k pokročilejší budoucí optimalizaci prováděného kódu [12].

3.2.8 Constant Folding

Constant Folding je další z obvyklých JVM optimalizací. Optimalizace spočívá v nahrazení výpočtu skládající se z konstantních hodnot přímo jejich výsledkem. Programátor se snaží v rámci čitelnosti kódu konstantní hodnoty ukládat do proměnných, které následně využije k výpočtu. Pokud je výsledek výpočtu vždy stejný, JVM v rámci optimalizace nahradí proměnné ve výpočtu výslednou hodnotou. V ukázce 3.6 je znázorněna metoda obsahující primitivní výpočet.

```
1 public int calculate() {
2   int a = 1;
3   int b = 2;
4   int sum = a + b;
5   return sum;
6 }
```

Ukázka 3.6: Výpočet z konstantních hodnot

Kompilátor může tento kód zoptimalizovat způsobem v ukázce 3.7.

```
1 public int calculate() {
2   int sum = 3;
3   return sum;
4 }
```

Ukázka 3.7: Constant Folding

JVM může aplikovat na zoptimalizovanou metodu `calculate()` další optimalizaci, která zredukuje tělo metody pouze na jeden řádek, který bude obsahovat pouze `return 3`. JVM může dále pokračovat v optimalizaci. Na základě získaných informací zjistí, že metoda `calculate()` vrací vždy stejnou hodnotu a provede nahrazení volání metody přímo hodnotou `3` [7].

3.2.9 Dynamická deoptimalizace

V kapitole 3.2.7 je popsána jedna z nejužitnějších optimalizací programovacího jazyku Java. Optimalizace pomocí metody inlining je mnohem jednodušší a použitelnější u statických programovacích jazyků než u dynamicky objektově orientovaných programovacích jazyků. V průběhu detekování hotspotů a použití metody inlining se může stát, že bude nahrazená metoda z nesprávné třídy a v průběhu vykonávání bude nezbytné optimalizovanou metodu vrátit do původního stavu.

```
1 Foo foo = getFoo();
2 foo.doSomething();
```

Ukázka 3.8: Dynamic deoptimization

V ukázce 3.8 kompilátor není schopen s jistotou určit, jakou implementaci metody `doSomething()` má nahradit, jestli implementaci třídy `Foo` nebo implementaci nějaké oddděné třídy. Na základě globálních informací získaných z průběžného analyzování kódu může kompilátor relativně přesně odhadnout, o jakou implementaci metody se jedná a provést její nahrazení. Jedná se pouze o odhad.

Program napsaný v jazyce Java podporuje načítání nových tříd do běžícího programu. Tato vlastnost značně komplikuje optimalizaci metodou inlining, protože mění vztahy mezi jednotlivými třídami. Nově načtená třída může být potomek třídy `Foo` a obsahovat novou implementaci metody `doSomething()`. V takovém případě je optimalizovaný kód již neplatný a JVM musí provést dynamickou deoptimalizaci metody `doSomething()`, aby nahradil optimalizovaný kód jeho skutečnou implementací. Bez této vlastnosti by nebylo možné použít optimalizaci metodou inlining v programovacím jazyce Java.

Pomocí optimalizace metodou inlining může kompilátor značně urychlit běh programu, nicméně následná deoptimalizace a vrácení původní implementace stojí kompilátor zbytečný čas. Programovací jazyk Java obsahuje klíčové slovo `final`, pomocí kterého lze zabránit oddědění příslušné třídy nebo metody. Použití klíčového slova `final` značně ulehčuje rozhodování kompilátoru v průběhu optimalizace metodou inlining.

Dynamická deoptimalizace je podporována v obou implementacích HotSpot JVM [3].

3.3 Warmup

Při benchmarkování metody je vhodnější měřit čas, jak dlouho trvá provedení zkompilované metody místo interpretované. Pokud nedojde k optimalizaci měřené metody, je vhodné uvést tuto informaci do výsledku benchmarku. Kompilátor provede kompilaci metody až po určitém počtu opakování, do té doby je metoda interpretovaná. Z tohoto důvodu je nutné provést warmup před skutečným měřením. Warmup neboli zahřívání je část procesu benchmarku, ve které dojde k několikanásobnému provedení měřené metody, aby byly provedeny různé optimalizace, došlo ke kompilaci kódu a nahrazení interpretovaného kódu zkompilovaným před spuštěním měření.

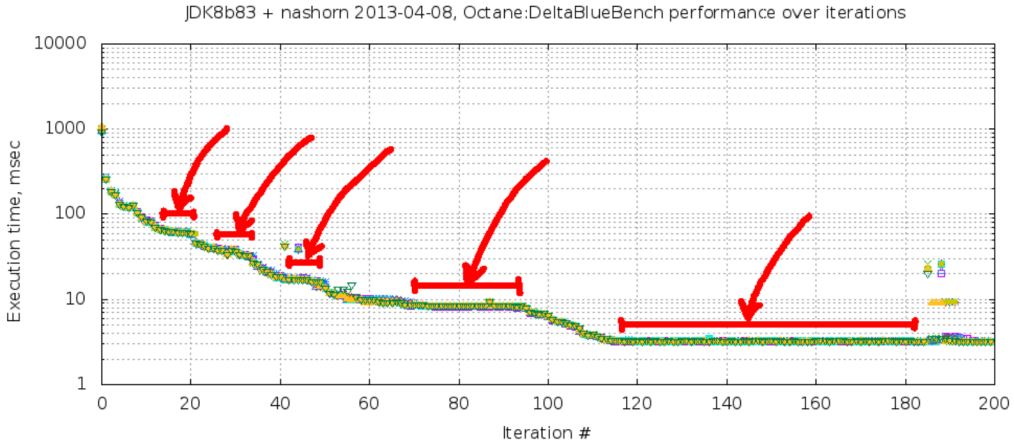
V dřívější verzi JIT a dynamickém kompilátoru bez *on-stack replacement* bylo mnohem jednodušší provést proces zahřívání měřeného kódu. Spustil se proces zahřívání s určitým počtem opakování volání. Pokud bylo nastaveno dostatečné množství opakování, kompilátor provedl kompilaci kódu a zahřívání bylo úspěšné. Při skutečném měření metody se již prováděla zkompilovaná metoda.

V současné verzi dynamických kompilátorů je tato část obtížnější. Kompilátor je spouštěn v méně předvídatelné době, JVM provede přepnutí z interpretovaného do zkompilovaného kódu v průběhu daného zpracování a v poslední řadě není zřejmé, kolikrát dojde ke zkompilování dané metody, jelikož kompilátor může metodu kompilovat opakovaně. Všechny tyto vedlejší události mohou zpomalit měření metody.

Pokud je zvolen nedostatečný počet zahřívání, může se stát, že dojde k měření interpretovaného kódu, což je pomalejší než kompilovaný kód. Dále může nastat situace, že kompilátor provede kompilaci kódu v průběhu měření, což vede ke zpomalení měřené metody, jelikož je spuštěna kompilace a musí dojít k přepnutí mezi interpretovaným a kompilovaným kódem. Pokud nastane tato situace, část měření probíhá s interpretovaným kódem a zbývající část již se zkompilovaným kódem.

Bohužel není nikde přesně uvedeno, jaký je dostatečný počet opakování zahřívání a pro každý microbenchmark se může hodit jiná nastavená hodnota. Teoreticky lze najít dostatečný počet opakování empiricky. Pokud se přestane čas po několik (desítek) iterací měnit/zmenšovat, metodu není nejspíše možné více optimalizovat a proces zahřívání lze ukončit [3].

Na obrázku 3.1 lze vidět, jakým způsobem ovlivňuje počet provedených iterací dobu běhu. S postupnými iteracemi klesá doba zpracování měřeného kódu.



Obr. 3.1: Graf účinnosti procesu warmup [13]

3.4 Garbage collection

Garbage collection je proces, který analyzuje haldu a hledá používané a již nepoužívané objekty. Používaný objekt znamená objekt, na který stále existuje reference z programu. Naopak nepoužívané objekty jsou ty, na které již nevede žádná reference z programu. Pro vyprázdnění nepoužívaných objektů z haldy se provádí neplánované čištění, tzv. garbage collection.

Z výše uvedených důvodů vyplývá, pokud má benchmark poskytovat nejpřesnější výsledky, je potřeba ho spustit opakovaně. Na základě opakovaného spouštění měření se v paměti začnou hromadit již nepotřebné nevyužívané objekty. Může nastat situace, že dojde k přerušení provádění benchmarkovaného kódu, protože JVM automaticky spustilo garbage collection. Toto neplánované přerušení se promítne i do výsledného naměřeného času a tím znehodnotí naměřenou iteraci. Bohužel nelze zakázat neplánované spuštění procesu garbage collection.

Programovací jazyk Java nabízí možnost spustit garbage collection manuálně. Některé benchmarkovací frameworky nabízejí možnost provést garbage collection vždy před začátkem měření, čímž je možné teoreticky eliminovat automatické neplánované spuštění tohoto procesu čištění [16].

3.5 Rozdíl proti benchmarkováním v jazyce C

Programovací jazyky C nebo C++ mají v tomto ohledu jednu zásadní výhodu, jsou to staticky kompilované programovací jazyky. Zkompilovaný kód obsahuje přímo strojové instrukce, které budou vykonávány.

To je rozdíl oproti zkompilovanému programu implementovaném v jazyce Java, protože strojové instrukce se nemohou v průběhu běhu programu změnit. Staticky kompilované programovací jazyky nepotřebují umět dynamicky deoptimalizovat kód. Optimalizaci získanou pomocí metody inlining, která je popsána v kapitole 3.2.7, není potřeba v průběhu běhu programu vracet do původní podoby, jelikož není možné, aby se změnila implementace optimalizované funkce. Pokud statický kompilátor provede například eliminaci mrtvého kódu, lze to zjistit přímo ve vygenerovaných strojových instrukcích. Instrukce pro vykonání eliminovaného mrtvého kódu se nikde nenachází.

Programy implementované v jazyce C nebo C++ lze změřit velice jednoduše. Dostačujícím řešením poskytujícím kvalitní výsledky je zaznamenání aktuálního času před a po provedení měřené části aplikace. Rozdíl zaznamenaných časů je doba trvání zpracování měřené části aplikace. Neplatí pravidlo, že chod programu v jazyce C nebo C++ není ovlivňován jinými procesy. Může například nastat výpadek stránek v různých úrovních cache nebo jiný nahodilý stav. Dobu běhu měřené části by mělo být možné odhadnout pouhým nahlédnutím do vygenerovaných strojových instrukcí.

V průběhu běhu programu teoreticky nikdy nenastane dynamická kompilace, uvolnění paměti procesem garbage collection, optimalizace kódu nebo další možné procesy, které ovlivňují běh programů implementovaných dynamickým programovacím jazykem.

3.6 Nástroje pro benchmarkování

Níže jsou zanalyzovány různé metody a nástroje, pomocí kterých lze vytvořit mikrobenchmark v programovacím jazyce Java. Na základě této analýzy bude vybrán optimální nástroj, který bude použit v aplikaci sloužící pro benchmarkování fragmentů kódu v jazyce Java.

3.6.1 Měření pomocí časových značek

Metoda `System.currentTimeMillis()` vrátí aktuální čas v milisekundách. Každého by mohlo napadnout zavolat tuto metodu před a po provedení určité části kódu, u kterého je cílem změřit dobu provedení.

Bohužel tento přístup je velice nepřesný. Pokud bude výpočet trvat velice krátkou dobu, nebude možné čas vyjádřit pomocí milisekund, protože po odečtení času před a po provedení výpočtu bude výsledek 0 ms. V případě, že by rozdíl časů byla nenulová hodnota, stále výsledný čas nemůžeme brát za přesný výsledek, neboť je nutné započítat chybu měření, která může dosahovat až 10 %.

Dále je možné použít metodu `System.nanoTime()`, která vrátí čas v nanosekundách. U těchto zmíněných metod si bohužel nemůžeme být jistí, že je vrácený čas přesný, protože zde záleží na použitém hardware a implementaci metody v JDK.

Jsou zde i další procesy, které by mohly výpočet ovlivnit a není možné zjistit přesnou dobu jejich vykonávání. Bohužel těmto činnostem není jednoduché se vyhnout. Jedná se především o všechny optimalizace popsané v kapitole 3.2. HotSpot dynamickou kompilaci je možné pomocí speciálního parametru úplně vypnout. Vypnutí HotSpot kompilace bohužel není řešením pro použití časových značek k benchmarkování, jelikož s vypnutou optimalizací je běžící program velice pomalý.

Další negativní efekt může způsobit Garbage collector, který se stará o uvolnění již nepotřebné alokované paměti. Garbage collector bohužel není možné ovlivnit, v náhodnou chvíli se může spustit během prováděného výpočtu a tím je zkreslen celkový naměřený čas výpočtu. Základní informace o garbage collection jsou popsány v kapitole 3.4.

Programovací jazyk Java je dynamický programovací jazyk, což znamená, že se běžící program může měnit v průběhu vykonávání. Pokud má benchmark poskytovat kvalitní a přesný výsledek, je nutné před skutečným měřením provést proces warmup popsaný v kapitole 3.3. Dále je potřeba vykonat měřenou část opakovaně a ze získaných výsledků spočítat průměr. Tyto dva zmíněné procesy je obtížnější provést v benchmarku vytvořeném metodou časových značek.

Pro implementaci a použití je to jedna z nejsnazších metod benchmarkování. Z výše uvedených negativních vlastností není možné získat přesnou dobu vykonávání měřené části.

3.6.2 JBenchX

Základní informace

Jedná se o malý framework, který slouží pro jednoduché psaní microbenchmarků. Framework je vytvořen, aby poskytoval možnost jednoduchého a přesného benchmarkování. Nastavení benchmarku se implementuje pomocí dostupných anotací.

Framework poskytuje možnost opakovaného spuštění měřeného kódu, aby bylo možné přesněji určit výsledný čas. Bohužel nenabízí možnost warmupu, ale již po spuštění benchmarku se provádí měření. Na základě tohoto nedostatku může nastat situace, že v prvotních iteracích proběhnou různé optimalizace, které ovlivní běh benchmarku.

Každý benchmark by měl obsahovat základní nastavení, které určí počet iterací daného měření. Framework nabízí celkem čtyři anotace pro nastavení počtu iterací. Framework funguje na principu definování minimálního a maximálního počtu iterací. Každý benchmark obsahuje buď výchozí nebo definovanou maximální odchylku, která slouží k akceptování proběhlé iterace. Pokud má naměřená hodnota větší odchylku než je definovaná, výsledek konkrétního měření není akceptován. Tento přístup by měl eliminovat iterace, které proběhly pomaleji například kvůli spuštěné optimalizaci měřeného kódu. Poslední anotací se nastavuje, kolik iterací musí skončit s akceptovaným výsledkem, aby byl benchmark úspěšně dokončen. Nastavení těchto anotací není povinné, pokud některá anotace není definována, použije se výchozí hodnota frameworku.

Eliminace mrtvého kódu

V kapitole 3.2.5 je popsána eliminace mrtvého kódu. Pokud se nemá smazat kód v průběhu optimalizace, framework poskytuje jedinou možnou volbu, jak zabránit odstranění. Je nutné vrátit objekt, se kterým metoda pracuje, jako návratovou hodnotu metody. Pokud má metoda nastavenou návratovou hodnotu typu `void`, je velice pravděpodobné, že z měřené metody bude po optimalizaci smazána veškerá implementace. Pokud nastane tato situace, benchmark je špatně napsaný a výsledek není možné považovat za správný.

Dokumentace

Dokumentace k použití nástroje JBenchX je velice chabá[1]. Oficiální stránky frameworku obsahují pouze jeden ukázkový příklad. Ukázkové příklady nejsou uvedeny ani v oficiálním repozitáři nástroje[2]. Bohužel nelze dohledat na internetu ani žádné neoficiální tutoriály nebo ukázkové příklady.

I přes absenci podrobnější dokumentace lze vytvořit jednoduchý mikrobenchmark, jelikož použití JBenchX frameworku je velice jednoduché.

[1] http://iquadrat.github.io/jbenchx/documentation.html

[2] https://github.com/iquadrat/jbenchx

Ukázka

V ukázce 3.9 je implementace metod pro provedení měření. Měří se doba provedení metody `contains()` na kolekcích typu `ArrayList` a `LinkedList`.

V konstruktoru třídy se deklarují kolekce a dojde k jejich naplnění uspořádanou množinou čísel. Toto je provedeno pouze jednou před spuštěním prvního měření. Anotací `@Bench` jsou označeny dvě metody, u kterých se má měřit doba jejich běhu.

Parametry měřených metod obsahují anotaci `ForEachInt` obsahující definovaná čísla, pro která se má metoda provést. Každá měřená metoda se provede celkem s třemi různými parametry.

```
1 public ContainsBenchmark() throws Exception {
2   arrayList = new ArrayList<>();
3   linkedList = new LinkedList<>();
4   for (int i = 1; i <= LIST_SIZE; i++) {
5     arrayList.add(i);
6     linkedList.add(i);
7   }
8 }
9
10 @Bench
11 public boolean arrayListContains(@ForEachInt({100, 50000, 99999})
     int value) {
12   return arrayList.contains(value);
13 }
14
15 @Bench
16 public boolean linkedListContains(@ForEachInt({100, 50000, 99999})
     int value) {
17   return linkedList.contains(value);
18 }
```

Ukázka 3.9: Metody pro měření frameworkem JBenchX

Výsledek

Po úspěšném dokončení mikrobenchmarku jsou naměřené hodnoty jednotlivých metod přehledně vypsány do konzole. Současně je i vygenerován `XML` soubor, který obsahuje veškeré informace týkající se benchmarku - nastavené parametry pro měření, počet iterací, časy jednotlivých měření atd.

V ukázce výstupu 3.10 jsou přehledně vypsány informace o provedeném benchmarku z ukázky 3.9.

```
1 Initializing Benchmarking Framework...
2 Running on Windows 10 Windows 10
3 Max heap = 3793747968 System Benchmark = 2,06ns
4 Performing 6 benchmarking tasks..
5 [0] ContainsBenchmark.arrayListContains(100    113ns
6 [1] ContainsBenchmark.arrayListContains(50000)  63.4us
7 [2] ContainsBenchmark.arrayListContains(99999)  188us
8 [3] ContainsBenchmark.linkedListContains(100) 113ns
9 [4] ContainsBenchmark.linkedListContains(50000) 60.5us
```

```
10 [5] ContainsBenchmark.linkedListContains(99999) 189us
11 Success.
```

Ukázka 3.10: Výstup měření frameworkem JBenchX

3.6.3 Caliper

Základní informace

Tento nástroj na provádění mikrobenchmarků vyvíjí společnost Google. Kromě měření mikrobenchmarků v aplikacích napsaných v programovacím jazyce Java podporuje i aplikace napsané pro platformu Android.

Metoda, jejíchž provedení se má měřit, musí být označena anotací `@Benchmark`. Aby se metoda, která se stará například o inicializaci dat potřebných pro měření, nevolala z měřené metody, je možné příslušnou metodu označit jako inicializační. Takto označená metoda se zavolá pouze jednou před spuštěním všech měření. Kdyby se metoda sloužící pro inicializaci dat volala z měřené metody, došlo by k znepřesnění naměřených hodnot, protože by se do měřeného času započítal i čas strávený inicializací dat.

Caliper obsahuje parametr, pomocí kterého lze nastavit počet iterací ve fázi warmup. Jedná se o provedení měřené metody s dostatečným počtem opakování, kdy ještě neprobíhá měření délky běhu. Díky této fázi se lze teoreticky vyhnout tomu, aby v další fázi, ve které již probíhá měření, HotSpot dynamický kompilátor provedl různé optimalizace kódu a zneplatnil naměřené hodnoty. Na rozdíl od jiných zde uvedených frameworků neobsahuje možnost nastavit počet, kolikrát se mají měřené metody provést. Tyto dva zmíněné nedostatky značně komplikují psaní mikrobenchmarků pomocí nástroje Caliper.

Podobně jako u frameworku JBenchX, i zde se neřeší již zmíněný mrtvý kód. Aby mrtvý kód neodstranil díky optimalizaci, je nutné použít stejný způsob řešení jako u frameworku JBenchX - jako návratovou hodnotu metody označit výsledek zavolané metody `contains()`.

Dokumentace

Pro framework Caliper neexistuje žádná oficiální stránka pod doménou firmy Google. Existuje pouze GitHub repozitář[3], který nicméně obsahuje dostatečnou dokumentaci. Repozitář obsahuje Wiki stránku, kde je zdokumentováno, jak vytvořit a spustit mikrobenchmark nástrojem Caliper. Dokumentace obsahuje například všechny parametry k ovlivnění běhu měření, se kterými je možné mikrobenchmark spustit.

[3] https://github.com/google/caliper

Součástí repozitáře je i sada několika ukázkových mikrobenchmarků[4]. Na základě těchto ukázkových příkladů lze jednoduše pochopit, jak vytvořit mikrobenchmark pomocí nástroje Caliper. Na internetu se nachází i řada neoficiálních tutoriálů.

Ukázka

V ukázce 3.11 je implementace metod pro provedení měření. Měří se doba provedení metody `contains()` na kolekcích typu `ArrayList` a `LinkedList`.

V metodě `setUp()` dojde k inicializaci a naplnění kolekcí uspořádanou množinou čísel. Díky anotaci `@BeforeExperiment` se provede volání této metody pouze jednou před začátkem provedení měření. Anotací `@Benchmark` jsou označeny dvě metody, u kterých se má měřit doba jejich běhu.

Aby se provedlo více iterací měření metody, je potřeba volání metody `contains()` obalit do cyklu, jehož počet iterací závisí na hodnotě `reps` uvedené v parametru. Tato hodnota je doplněná frameworkem automaticky. Bohužel není v dokumentaci uveden způsob, jak ovlivnit proměnnou `reps`, aby se nastavil libovolný počet iterací.

```
1 @Param({"100", "50000", "99999"})
2 private int testedValue;
3
4 @BeforeExperiment
5 void setUp() {
6   arrayList = new ArrayList<>();
7   linkedList = new LinkedList<>();
8
9   for (int i = 1; i <= LIST_SIZE; i++) {
10    arrayList.add(i);
11    linkedList.add(i);
12  }
13 }
14
15 @Benchmark
16 boolean arrayListContains(int reps) {
17   boolean cointains = false;
18   for (int i = 0; i < reps; i++) {
19     cointains = arrayList.contains(testedValue);
20   }
21   return cointains;
22 }
23
24 @Benchmark
25 boolean linkedListContains(int reps) {
26   boolean cointains = false;
27   for (int i = 0; i < reps; i++) {
28     cointains = linkedList.contains(testedValue);
29   }
30   return cointains;
31 }
```

Ukázka 3.11: Metody pro měření frameworkem JBenchX

[4]https://github.com/google/caliper/tree/master/caliper-examples

Výstup

Na výstup do konzole jsou průběžně vypisovány informace o tom, která metoda je aktuálně měřená.

```
1 Experiment {instrument=runtime, benchmarkMethod=arrayListContains,
    vm=default, parameters={testedValue=100}}
2 Results:
3   runtime(ns): min=99,21, 1st qu.=100,25, median=100,82, mean
      =100,89, 3rd qu.=101,90, max=102,21
```

Ukázka 3.12: Výstup do konzole frameworku Caliper

V ukázce 3.12 se nachází výpis z měření metody `arrayListContains(100)`. Výpis obsahuje minimální a maximální dobu běhu, medián a další informace. Na závěr mikrobenchmarku je vypsána URL adresa, na které se nachází v přehledné grafické podobě informace o měření. Tabulka je velice obsáhlá, obsahuje i různé nastavení JVM.

3.6.4 Java Microbenchmark Harness

Základní informace

Framework Java Microbenchmark Harness (JMH) pochází z dílny programátorů OpenJDK. Tato skupina lidí se podílí na vývoji Javy. Dá se očekávat, že díky znalostem vývojářů týkajících se Javy a Java virtual machine bude tento nástroj dobře odladěný a použitelný. Původně měl být součástí Java 1.9, ale z neznámých důvodů bylo jeho začlenění odloženo.

JMH stejně jako framework Caliper provádí fázi warmup. V druhé fázi již probíhá měření doby běhu teoreticky optimalizovaných testovaných metod. Pro obě fáze lze anotacemi nastavit, kolikrát se mají provést. Bohužel není úplně jednoduché zvolit správně hodnoty těchto anotací. Nikde není uvedeno, kolikrát by měly být měřené metody provedeny nanečisto ve fázi warmupu, aby se provedly všechny optimalizace.

Frameworkem lze zabránit, aby kompilátor začal provádět různé JIT optimalizace. Tímto lze například zjistit vliv optimalizace na měřené metody. JMH se umí teoreticky vyhnout provedení procesu garbage collector v průběhu měření doby běhu tím, že se vynutí spuštění procesu garbage collector po každé provedené iteraci. Tímto opatřením by měl mít JVM vždy dostatek paměti a neměl by mít potřebu vyčistit již nepoužívané objekty v paměti. Framework dále nabízí možnost nastavit spuštění mikrobenchmarku s kompilátorem v režimu server (viz kapitola 3.2.2), čímž je měřený mikrobenchmark důkladněji optimalizován.

Naměřené hodnoty je možné získat v různých jednotkách. Výsledné hodnoty značí počet cyklů za jednotku času nebo průměrný čas běhu benchmarku, záleží na nastavení mikrobenchmarku.

Eliminace mrtvého kódu

Podobně jako nástroje JBenchX a Caliper ani tento framework neřeší problematiku eliminace mrtvého kódu automaticky. Nabízí dvě možnosti, jak se s ní vypořádat manuálně. První možnost je stejná jako v předchozích nástrojích, označit jako návratovou hodnotu metody výsledek zavolané metody (viz metoda `arrayListContains()` v ukázce 3.13). Druhou možností je použít třídu `Blackhole`, která slouží právě k zabránění smazání mrtvého kódu. Místo toho, aby se objekt vrátil jako návratová hodnota metody, vloží se objekt do parametru metody `consume()` ze třídy `Blackhole`, čímž je zaručeno, že je daný objekt potřebný pro spuštěný program a nebude označen jako mrtvý kód (viz metoda `linkedListContains()` v ukázce 3.13).

Pokud metoda obsahuje více vypočtených proměnných, které nejsou dále v metodě použity a byly by pravděpodobně smazány optimalizací, lze každou proměnnou použít jako parametr v metodě `consume()` ze třídy `Blackhole`, čímž nedojde k jejich smazání optimalizací. Použití tohoto přístupu může být značná výhoda oproti prvnímu přístupu, protože jako návratovou hodnotu metody lze použít pouze jednu proměnnou.

Dokumentace

Oficiální dokumentace nástroje JMH se nachází na stránkách OpenJDK[5]. Tato dokumentace obsahuje pouze základní informace o frameworku včetně návodu na vygenerování mikrobenchmarku z existujícího Maven archetypu a jeho spuštění. Na stránce se dále nachází odkaz na ukázkové příklady[6] a oficiální repozitář projektu[7].

JMH obsahuje bohatou sadu ukázkových příkladů včetně podrobných komentářů a postupů ke spuštění. Na základě ukázkových příkladů je možné pochopit, jakým způsobem lze napsat kvalitní mikrobenchmark v JMH. Jednotlivé příklady obsahují podrobné komentáře, jaký význam mají různé anotace a jak je použít. Na internetu lze dále dohledat velké množství různých neoficiálních tutoriálů popisujících chování, dostupné anotace a ukázkové příklady frameworku JMH.

Dokumentace neobsahuje pouze ukázkové příklady pro vytvoření mikrobenchmarku. Obsahuje například, jak lze sledovat průběh běžícího mikrobenchmarku pomocí různých profilerů nebo jak nastavit bezpečnostní pravidla pro JVM.

Ukázka

V ukázce 3.13 je implementace metod mikrobenchmarku frameworkem JMH. Ukázka obsahuje implementaci stejné funkcionality jako v ukázkách již zmíněných frameworků.

[5] https://openjdk.java.net/projects/code-tools/jmh/

[6] https://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/

[7] https://hg.openjdk.java.net/code-tools/jmh/

V metodě `setUp()` dojde k inicializaci a naplnění kolekcí uspořádanou množinou čísel. Díky anotaci `@Setup(Level.Trial)` se provede volání této metody před spuštěním celého benchmarku. Anotací `@Benchmark` jsou označeny 2 metody, u kterých se má měřit doba jejich běhu. Volání metody `contains()` je vloženo jako parametr do metody `consume()` třídy `Blackhole`. Díky tomuto kroku nedojde ke smazání této části kódu optimalizací. Každá metoda bude měřena celkem pro tři různá čísla díky anotaci `Param()` nad proměnnou `testedValue`.

```
@Param({"100", "50000", "99999"})
int testedValue;

@Setup(Level.Trial)
public void setUp() {
  arrayList = new ArrayList<>();
  linkedList = new LinkedList<>();

  for (int i = 1; i <= LIST_SIZE; i++) {
    arrayList.add(i);
    linkedList.add(i);
  }
}

@Benchmark
public boolean arrayListContains() {
  return arrayList.contains(testedValue);
}

@Benchmark
public void linkedListContains(Blackhole bh) {
  bh.consume(linkedList.contains(testedValue));
}
```

Ukázka 3.13: Metody pro měření frameworkem JMH

Výstup

Framework nabízí vypsání výsledku benchmarku do konzole nebo do souboru. Na rozdíl od jiných frameworků poskytuje možnost vypsat další informace o proběhlém měření a informací z Java virtual machine pomocí různých profilerů. Dále lze vygenerovat soubory v různých formátech obsahující výsledek mikrobenchmarku. JMH podporuje vygenerování souboru ve formátech `TEXT`, `CSV`, `SCSV`, `JSON` a `LATEX`.

```
Parameters: (testedValue = 50000)
Result "cz.rojik.ContainsTest.arrayListContains":
  64,599 ?(99.9%) 2,634 us/op [Average]
  (min, avg, max) = (61,976, 64,599, 73,232), stdev = 3,033
  CI (99.9%): [61,965, 67,233] (assumes normal distribution)
```

Ukázka 3.14: Naměřené hodnoty pro metodu arrayListContains()

V ukázce výstupu 3.14 se nachází informace o provedení měření metody pro kolekci `ArrayList` s hodnotou `50000`, kde je zobrazena průměrná hodnota ze všech měření včetně možné chyby, dále minimální a maximální doba běhu.

```
1 Benchmark                                (testedValue)  Mode  Cnt     Score
      Error  Units
2 ContainsTest.arrayListContains                     100  avgt   20     0,107
    ? 0,001  us/op
3 ContainsTest.arrayListContains                   50000  avgt   20    64,599
    ? 2,634  us/op
4 ContainsTest.arrayListContains                   99999  avgt   20   130,039
    ? 2,558  us/op
5 ContainsTest.linkedListContains                    100  avgt   20     0,107
    ? 0,002  us/op
6 ContainsTest.linkedListContains                  50000  avgt   20    62,366
    ? 1,205  us/op
7 ContainsTest.linkedListContains                  99999  avgt   20   173,902
    ? 3,026  us/op
```

Ukázka 3.15: Výstup do konzole frameworku JMH

Ukázka výstupu 3.15 obsahuje výsledné naměřené hodnoty pro porovnání. Sloupec `Score` obsahuje dobu zpracování volané metody, ve sloupci `Error` je uvedena možná chyba. Poslední sloupec `Units` zobrazuje použité jednotky.

3.7 Porovnání nástrojů

V tabulce 3.1 jsou uvedeny základní informace o každém z výše uvedených frameworku na implementaci mikrobenchmarků v programovacím jazyce Java.

	Vývojáři	**Rok vydání**	**Verze**	**Poslední úprava**	**Licence**
JBenchX	Micha Riser	2018	0.3.1	10/2018	Eclipse Public License
Caliper	Google	2014	v1.0-beta-2	03/2019	Apache 2.0 License
Java Micro-benchmark Harness	OpenJDK	2018	1.21	01/2019	General Public License v2

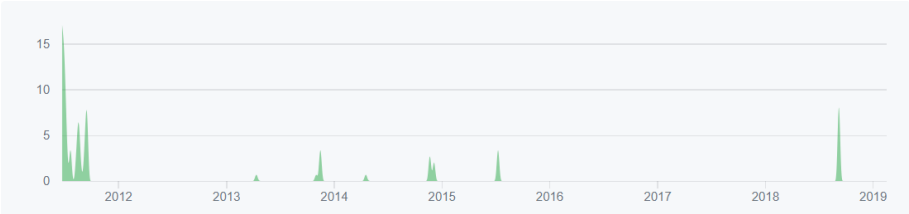
Tab. 3.1: Porovnání frameworků

3.7.1 Nedostatky nástrojů

JBenchX

Nevýhodou JBenchX frameworku je nutnost spouštět benchmarky pouze v programu Eclipse, do kterého je nutné doinstalovat příslušný doplněk `JBenchX`. Jiným způsobem není bohužel možné mikrobenchmark spustit.

Další nevýhodou je již zastavený vývoj tohoto frameworku, což lze vidět na obrázku A.1. Podle GitHub stránky[8] je poslední vydaná verze `0.3.1`. Tato verze byla vydána v roce 2018 a obsahuje pouze drobné úpravy. Verze `0.3.0` byla vydána v roce 2011. Nejspíše lze předpokládat, že již není plánován v budoucnu žádný další vývoj, což může být škoda, jelikož framework by toho mohl umět daleko více.

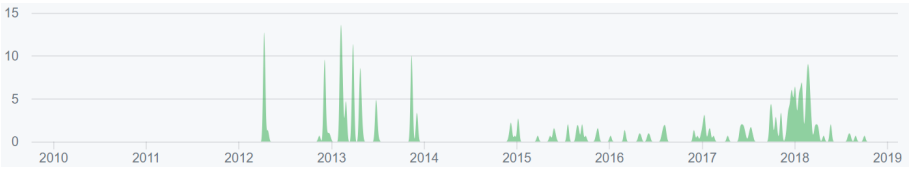


Obr. 3.2: Graf úprav JBenchX (7. srpen 2011 – 1. duben 2019) [6]

Caliper

Nevýhodou Caliper frameworku je nemožnost ovlivnit počet měření. V dokumentaci je přímo napsáno, že není možné ovlivnit počet měření. Nástroj Caliper provede určitý počet iterací na základě jeho automatického nastavení.

V současné verzi toho dokáže nabídnout relativně mnoho, ale chybí mu snaha o eliminování JVM procesů, které se náhodně spouští v průběhu měření a znehodnotí naměřené hodnoty. GitHub stránka[9] obsahuje celkem tři vydané verze frameworku. Jedná se o verze `v0.5-rc1`, `caliper-1.0-beta-1` a `v1.0-beta-2`. První verze byla vydána v roce 2012 a poslední v roce 2015. Na základě grafu 3.3 to vypadá, že je framework stále vyvíjen, nicméně v posledních letech nebyla vydána žádná nová verze. Framework není bohužel vydaný v žádné finální verzi, vždy se jednalo pouze o zkušební verze sloužící pro otestování funkčnosti. Dále obsahuje GitHub stránka Caliper frameworku mnoho otevřených dotazů (ke dni 8. 4. 2019 celkem 140 požadavků). Na základě výše uvedených informací se lze domnívat, že do frameworku již nebudou implementovaná žádná nová vylepšení.



Obr. 3.3: Graf úprav Caliper (29. listopad 2009 – 6. duben 2019) [1]

[8]GitHub frameworku JBenchX - https://github.com/iquadrat/jbenchx
[9]GitHub frameworku Caliper - https://github.com/google/caliper

Java Microbenchmark Harness

V prvotní analýze frameworku JMH nebyly nalezeny žádné větší nedostatky. Jako nedostatek lze označit, že neexistuje žádná podrobná oficiální dokumentace od společnosti OpenJDK. Nicméně jako dokumentaci lze považovat veřejně dostupný repozitář, který obsahuje mnoho ukázkových příkladů, jak je možné vytvořit mikrobenchmark pomocí JMH. Framework JMH je velice oblíbeným nástrojem pro vytváření mikrobenchmarků, takže lze dohledat neoficiální dokumentace, jak lze framework použít.

3.7.2 Výběr frameworku

Naměřené hodnoty ze všech třech frameworků v ukázkových příkladech jsou relativně podobné. Různé odchylky mohou být způsobeny jiným aktuálním vytížením procesoru. Na základě těchto hodnot lze prohlásit, že kvalita měření je u všech testovaných frameworků podobná. Výsledky jednotlivých měření každého frameworku jsou v příloze A.

Všechny nástroje nabízí jednoduchou tvorbu mikrobenchmarků v Javě. Mikrobenchmark je možné vytvořit po prohlédnutí implementace již napsaného ukázkového příkladu například v dokumentaci. Nevýhoda nástroje JBenchX je ta, že ke spuštění mikrobenchmarku je zapotřebí program `Eclipse` s nainstalovaným pluginem `JBenchX`. Zbylé nástroje lze spustit pomocí nástroje `Maven`.

Na základě informací uvedených v kapitolách 3.6 a 3.7.1 lze usoudit, že nejlépe vyvinutý a použitelný framework pro implementování mikrobenchmarků v programovacím jazyce Java je Java Microbenchmark Harness popsaný v kapitole 3.6.4.

Na rozdíl od ostatních frameworků je stále vyvíjen. Navíc na vývoji se podílí firma OpenJDK, která zároveň vyvíjí i samotný Java Virtual Machine, takže lze předpokládat, že vývojáři skutečně vědí, jakým způsobem by se měly psát mikrobenchmarky v programovacím jazyce Java. Výraznou informací je i fakt, že JMH měl být implementován přímo do jazyka Java verze 9. Nejspíše z časových důvodů k tomuto kroku nedošlo. Framework je implementován až ve verzi Java 12[10]. Další pozitivní informace je, že JMH se nesoustředí pouze jen na změření doby trvání zpracování dané metody, ale i na možnost sledování, co v průběhu mikrobenchmarku prováděl JVM za neplánované operace pomocí různých profilerů.

Na základě výše uvedených pozitivních informací lze prohlásit, že JMH je skutečně nejlepší volbou z existujících frameworků pro psaní mikrobenchmarků v jazyce Java.

[10]Odkaz na ticket s integrací JMH do Java 12 - https://bugs.openjdk.java.net/browse/JDK-8050952

4 APLIKACE PRO BENCHMARKOVÁNÍ

4.1 Úvod do aplikace

4.1.1 Cíl aplikace

Hlavním cílem diplomové práce je vytvoření aplikace pro benchmarkování fragmentů kódu v programovacím jazyce Java. Aplikace by měla poskytovat službu, která porovná zadané fragmenty kódu v jazyce Java. Výsledkem úspěšně provedené akce by se měl uživatel dozvědět, jaký fragment kódu je zpracován nejrychleji pro stejnou množinu dat.

Řešený problém

V rámci kapitoly 3.6 byly zanalyzovány dostupné frameworky pro implementaci mikrobenchmarků. Bohužel tyto nástroje poskytují pouze knihovny, které umožňují vytvořit a provést mikrobenchmark. Aby si mohl uživatel vytvořený mikrobenchmark spustit, musí mít nainstalovaný Java SE Development Kit sloužící pro kompilaci a spuštění Java projektu. Dále musí mít uživatel povědomí o tom, jak vytvořit mikrobenchmark pomocí vybraného nástroje. Na závěr musí mít uživatel povědomí o získaných výsledcích, co znamenají jednotlivé naměřené hodnoty. Jedná se o několik kroků sloužících pro získání informace o tom, jaký fragment kódu je nejrychlejší. Právě tento problém by měla řešit vytvořená aplikace.

Aplikace by měla uživatele odstínit od všech výše zmíněných problémů a potřebných znalostí týkajících se použití frameworku pro vytvoření mikrobenchmarku. Dále by měla uživateli ušetřit mnoho času automatickým vytvořením, spuštěním a vyhodnocením mikrobenchmarku. Uživatel by měl pouze aplikaci říct, že chce porovnat tyto fragmenty kódu na zadané testovací množině dat. Aplikace by se měla vnitřně postarat o veškerou nutnou práci týkající se vytvoření, spuštění a získání výsledku mikrobenchmarku. Jako odpověďna uživatelův dotaz by měla aplikace vrátit odpověď, která bude obsahovat časy, jak dlouho trvá zpracovat jednotlivé zadané fragmenty kódu. Tento úkol by měl být hlavním cílem aplikace.

Podobné aplikace

V rámci analýzy nebyla nalezena žádná aplikace řešící stejný problém. Psaní mikrobenchmarků v programovacím jazyce Java je relativně komplikované, což může být důvod, proč prozatím nevznikla žádná veřejně dostupná aplikace pro porovnání doby běhu různých fragmentů kódu.

Pro jazyk C++ je vytvořeno několik online aplikací sloužící tomuto účelu. Například velice uživatelsky přívětivě vypadá aplikace Quick++ Benchmark[1]. Rozdíl proti benchmarkování v jazyce Java je popsán stručně v kapitole 3.5.

Dále existují online aplikace pro porovnávání fragmentů kódu v jazyce Javascript. V tomto jazyce je ještě jednodušší provést měření než v jazyce C++ nebo Java, jelikož prováděný kód běží přímo v prohlížeči uživatele. Tím pádem autor aplikace nemusí řešit bezpečnost vykonávaného kódu nebo jestli vykonávaný kód neovlivňují jiné běžící procesy. Jedna z ukázkových aplikací se jmenuje JSBench.me[2].

Další vlastnosti aplikace

Mezi další vlastnosti aplikace by měla patřit správa benchmarků. Jedná se o možnost uživatele spravovat si již proběhlé úspěšné nebo neúspěšné mikrobenchmarky. Uživatel by měl mít možnost si zobrazit detail již proběhlého benchmarku včetně jeho výsledku.

Proces mikrobenchmarku je relativně komplikovaný. V prvotní fázi je nutné vygenerovat mikrobenchmark na základě přijatých dat. V další fázi je potřeba provést kompilaci. Po úspěšně kompilaci lze teprve spustit samotný mikrobenchmark, který může běžet v řádu minut. Proto by bylo přívětivé, aby aplikace dokázala uživatele informovat, v jaké fázi se zrovna nachází proces mikrobenchmarku.

4.1.2 Typ aplikace

V rámci analýzy je potřeba rozhodnout, jestli se bude implementovat online, nebo offline aplikace. Níže jsou rozebrány oba přístupy. Na základě analýzy bude vybráno finální řešení k implementaci.

Offline aplikace

Jedna z hlavních výhod offline aplikace je bezpečnost. Vytvořený mikrobenchmark z požadovaných fragmentů kódu je nutné zkompilovat a spustit. Jelikož se zkompilovaný program spouští na systému uživatele, odpadá nutnost řešit bezpečnost spouštěného programu. Předpokládá se, že uživatel ví, jaký kód bude proveden na jeho systému. Pokud by zkompiloval a spustil škodlivý kód, bude napaden jeho systém.

Další výhodou je přehled o běžících procesech, které mohou ovlivnit běh mikrobenchmarku. Uživatel přesně ví, jaké další procesy vytěžují jeho systém. Pokud si uživatel dobrovolně nespustí více mikrobenchmarků zároveň, má zajištěno, že mu běh mikrobenchmarku neovlivňuje běh jiného mikrobenchmarku.

[1] http://quick-bench.com
[2] https://jsbench.me/

Mezi výhodu může patřit znalost hardware, na kterém je prováděn mikrobenchmark. Pokud by uživatel chtěl vyzkoušet, jak dlouho trvá zpracovat fragment kódu na méně výkonném počítači, stačí pouze spustit offline aplikaci na vybraném počítači.

Hlavní nevýhoda offline aplikace je nutnost mít na počítači nainstalovanou Javu požadované verze, ve které byla aplikace implementovaná a zkompilovaná. Aplikace může vyžadovat i další služby pro svůj běh, jako například databázi. Součástí musí být i dokumentace obsahující všechny informace potřebné pro zprovoznění aplikace.

Na počítači běžného uživatele běží současně několik procesů, které se střídají o procesorový čas. Takové prostředí není úplně vhodné pro spouštění mikrobenchmarku, protože tyto běžící procesy mohou silně ovlivnit naměřené hodnoty.

Nevýhodou může být i nemožnost zjištění, jestli je vytvořená aplikace skutečně využívaná veřejností.

Online aplikace

V dnešní době začínají být preferované online aplikace před offline aplikacemi. Hlavní výhodou online aplikace je její schopnost běžet v internetovém prohlížeči. Díky internetovému prohlížeči odpadá nutnost instalace všech programů, které jsou nutné pro spuštění aplikace. Backend aplikace[3] běží na serveru. Díky tomu lze aplikaci ovládat skrze dostupnou frontend aplikaci[4]. K ovládání aplikace může být vytvořeno několik frontend aplikací, které komunikují pouze s jednou backend aplikací.

Další výhodou tohoto řešení je běžící mikrobenchmark mimo uživatelův počítač. Mikrobenchmark je prováděn na straně serveru, takže žádným způsobem nevytěžuje počítač uživatele, který pouze pošle požadavek na spuštění benchmarku a dále už jen čeká na výsledek. Lze předpokládat, že na serveru určeném pouze pro běh backend aplikace běží jen nejnutnější procesy, proto by neměly být výsledky mikrobenchmarku příliš ovlivněny.

Mezi další výhody patří dostupnost. Aplikace může být dostupná teoreticky z celého internetu, pokud by běžela na dostatečně výkonném hardwaru. Díky tomu může uživatel například získat historii jeho proběhlých benchmarků z různých počítačů. Další výhodou je možnost spustit dlouhotrvající benchmark, vypnout počítač a následně se za určitou dobu podívat na výsledky benchmarku.

Mezi výhody lze zařadit i běh všech mikrobenchmarků na stejném prostředí. Na základě tohoto faktu je možné porovnávat výsledky různých benchmarků vykonávající podobné fragmenty. Jelikož by se všechny mikrobenchmarky prováděly na jednom místě,

[3] Aplikace se kterou lze komunikovat pomocí HTTP požadavků. Aplikace nemá žádný vzhled pro pohodlnější ovládání.

[4] Aplikace sloužící pro přívětivější práci s backend serverem. Poskytuje uživatelské rozhraní, které na základě vyvolaných akcí posílá HTTP požadavky na definovaný server.

uživatel spravující aplikaci by měl detailní přehled o všech provedených mikrobenchmarcích. Jednak by se uživatel mohl podívat, jaké fragmenty kódu programátoři porovnávají, ale hlavně má informaci o tom, zda je vůbec aplikace využívaná.

Nevýhodou online aplikace je nutnost vlastnit server, kde by mohla aplikace běžet. Pokud bude na aplikaci posíláno mnoho požadavků ve velice krátkém intervalu, aplikace se může přetížit a přestat komunikovat. To ovšem záleží na nastavení serveru a jeho zátěži.

Nevýhodou online aplikace je její veřejná dostupnost. Pokud může aplikaci používat kdokoliv, musí být aplikace velice dobře zabezpečena proti různým útokům. Pokud by se povedlo aplikaci zneužít, bylo by možné například ovládnout celý server. U online aplikace je požadovaný mikrobenchmark kompilován a vykonán přímo na serveru. Uživatel může tímto způsobem spustit na serveru jakýkoliv škodlivý Java kód. Aplikace musí být zabezpečena, aby spuštěný kód nijak neohrozil běh aplikace nebo samotného serveru. Tuto vlastnost je možné považovat za jednu z největších nevýhod online aplikace.

Jako poslední nevýhoda může být považována komunikace s backend aplikací. Pokud není k backend aplikaci vytvořené žádné uživatelské rozhraní, je nutné komunikovat s aplikací pouze skrze HTTP požadavky, což není příliš uživatelsky přívětivé. S tímto problémem souvisí i nutnost poskytovat kvalitní dokumentaci sloužící pro snadné pochopení, jak komunikovat s backend aplikací pomocí HTTP požadavků.

Vyhodnocení analýzy

Vytvořit offline aplikaci by bylo snazší než online aplikaci. Vývojář by musel řešit mnohem méně problémů při implementaci. Nicméně koncový uživatel by měl mnohem více práce se zprovozněním aplikace na svém počítači než otevřít aplikaci v internetovém prohlížeči. Cílem diplomové práce je vytvořit aplikaci, která bude veřejně dostupná a využívaná, proto je nutné cílit na uživatelskou přívětivost. Na základě výše uvedených výhod a nevýhod obou typů aplikace s ohledem na uživatelskou přívětivost je vybrána online aplikace. Vytvořit online aplikaci bude sice obtížnější, ale ve výsledku přinese více výhod pro uživatele než offline aplikace.

4.2 Implementace aplikace

4.2.1 Technologie

Aby nebylo nutné použít k implementaci webové aplikace tzv. servlety, pro usnadnění vývoje je potřeba vybrat framework. Servlety jsou stavební kameny webové aplikace, které zajišťují nízkoúrovňovou komunikaci přes protokoly založené na bázi dotaz - odpověď. Typicky se jedná o HTTP protokol, na kterém jsou webové aplikace založeny. Vytvořit apli-

kaci pomocí servletů není vůbec jednoduché, jelikož se programátor musí zabývat složitou konfigurací, vytvářením instancí jednotlivých tříd, složitou komunikací s databází atd.

V dnešní době existuje mnoho frameworků usnadňujících implementaci webové aplikace v programovém jazyce Java. Mezi nejpoužívanější frameworky patří String, JavaServer Faces, Struts, Vaadin a další[5]. V této práci je pro implementaci webové aplikace vybrán framework Spring Boot.

Spring Boot

Spring Boot je užitečný nástroj pro Java vývojáře sloužící k snazšímu vytvoření samostatně běžící webové aplikace. Jedná se o rozšíření populárního Spring frameworku, které odstraňuje nutnost definování složité konfigurace Spring aplikace.

Každá aplikace vytvořená nad tímto frameworkem má v sobě zabudovaný server (Tomcat, Jersey) sloužící k jednoduchému a rychlému spuštění aplikace. Pro spuštění aplikace není potřeba nasazovat zkompilovaný `WAR` soubor na samostatně běžící aplikační server. Další výhodou je již snadná konfigurace aplikace. Aplikace je konfigurována pomocí dostupných anotací. Zároveň Spring Boot používá princip autokonfigurace. Pokud framework nalezne v projektu závislost například na databázi, připojení k databázi automaticky nakonfiguruje.

Další výhodou je implementace Dependency Injection. Jedná se o techniku pro vkládání závislostí mezi jednotlivými komponentami programu. V aplikaci se definují tzv. Beany. Pokud je třída závislá na jiné třídě, Spring Boot se o tuto závislost automaticky postará. Programátor je díky tomu odstíněn od manuálního vytváření instancí jednotlivých tříd.

Pomocí frameworku je možné velice jednoduše vytvořit REST API podporující různé HTTP metody (GET, POST, PUT, DELETE, PATCH, ...). Použití Spring Boot frameworku podporují vývojová prostředí, jako například IntelliJ Idea nebo Eclipse. Framework je velice oblíbený a rozšířený. Má podrobnou dokumentaci a dále lze dohledat mnoho neoficiálních tutoriálů popisujících nejběžnější problémy, které se během implementace webové aplikace mohou řešit. Framework obsahuje mnoho dalších rozšíření, které ulehčují programátorovi naimplementovat webovou aplikaci [15].

Docker

Docker je open-source framework, který nabízí odlehčený druh virtualizace pomocí linuxových kontejnerů používaných namísto dosud běžných virtuálních strojů. Docker vychází z tradičních linuxových distribucí, jako jsou Red Hat Enterprise Linux nebo Ubuntu. Umožňuje zabalit aplikace a služby jako bitové kopie, které běží ve svých vlastních přenositelných kontejnerech. Kontejner je zjednodušeně řečeno virtualizace, u které se ovšem

[5] Seznam populárních frameworků - https://javapipe.com/blog/best-java-web-frameworks/

nesimuluje veškerý hardware, nad kterým potom běží celý virtuální počítač. Kontejnery sdílejí kernel hosta a jejich izolace využívá mnoha Linux technologií. Všechny kontejnery běží na jádře hostitelského systému, ale jsou zamknuté ve svém vlastním runtime prostředí, odděleně od prostředí hostitele. Kontejner je velmi malý, snadno přenositelný, extrémně rychle startuje a je podstatně efektivnější co do využití zdrojů. Hlavním cílem Dockeru je přenositelnost a škálovatelnost aplikací. Kontejnery lze přesouvat mezi fyzickými, virtuálními a cloudovými prostředími bez nutnosti jakékoli úpravy [17].

Docker funguje na principu virtualizace na úrovni operačního systému. Virtualizace používá pro svůj běh jádro operačního systému. Umožňuje spouštět více vzájemně nezávislých a izolovaných virtuálních počítačů, které se označují jako kontejnery. Aplikace pro virtualizaci na úrovni operačního systému vytvoří virtualizační vrstvu, která řídí běh více virtuálních kontejnerů, zajišťuje abstraktní rozhraní pro přístup k hostitelskému systému a zajišťuje přidělování prostředků od systému. Hlavní výhodou je rychlost [10].

Využití Dockeru v rámci práce je popsáno v kapitole 4.2.3.

4.2.2 Architektura aplikace

Aplikace je implementována v programovacím jazyce Java 1.8. Dále je použit nástroj Maven, který slouží pro snadné získání potřebných knihoven a dále pro usnadnění práce při kompilaci aplikace. Další technologie použité v aplikaci jsou popsány níže. Aplikace je rozdělena do třech modulů:

- `microbenchmark-service`
- `microbenchmark-backend`
- `microbenchmark-web`

Modul microbenchmark-service

Modul `microbenchmark-service` slouží pro obsloužení požadavku na provedení mikrobenchmarku. Obsahuje veškerou aplikační logiku sloužící pro vykonání mikrobenchmarku. Modul je schopen vygenerovat, zkompilovat, vykonat a vyhodnotit mikrobenchmark. Nachází se zde třídy pro spuštění externích nástrojů, jako jsou Maven a Docker. Modul je detailněji rozebrán v kapitole 4.2.4.

Modul microbenchmark-backend

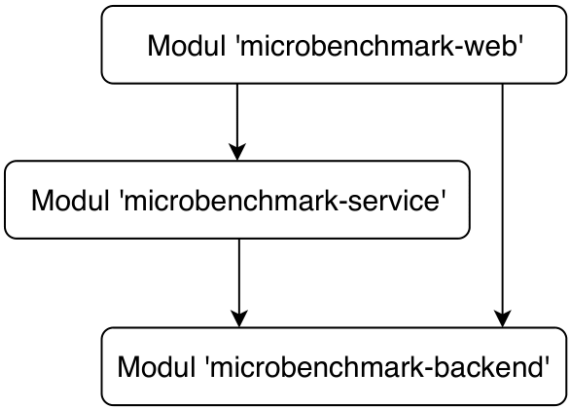
Modul `microbenchmark-backend` slouží převážně pro práci s databází. Aplikace používá databázi PostgreSQL. Modul obsahuje aplikační logiku pro práci s uživateli, přihlašování uživatelů, konfiguraci pro běh mikrobenchmarku a další. Dále zajišťuje správu všech provedených nebo aktuálně prováděných mikrobenchmarků, uživatelů a rolí. Modul je detailněji rozebrán v kapitole 4.2.5.

Modul microbenchmark-web

Hlavním přínosem tohoto modulu je definování REST API sloužící pro ovládání aplikace. Mimo REST API je zde definováno i API pro websockety. Modul dále obsahuje nastavení, jaké oprávnění uživatel potřebuje pro zavolání REST endpointu. Modul je detailněji rozebrán v kapitole 4.2.6.

Závislosti

V modulu `microbenchmark-web` dochází k obsluhování příchozích požadavků. Na základě požadované akce je zavolána třída ze zbylých dvou modulů. Z tohoto důvodu potřebuje modul závislosti na oba zbylé moduly. Modul `microbenchmark-service` potřebuje pro správné vykonání mikrobenchmarku číst a ukládat informace do databáze. Proto je tento modul závislý na `microbenchmark-backend`. Poslední modul nepotřebuje ke svému fungování ani jeden z výše uvedených modulů. Závislost jednotlivých modulů je znázorněna na obrázku 4.1.



Obr. 4.1: Závislosti modulů aplikace

4.2.3 Zabezpečení vykonávaného kódu

Jak je již zmíněno v kapitole 4.1.2, jedna z největších nevýhod webové aplikace je její bezpečnost. Uživatel pošle na server fragmenty kódu, které chce porovnat. Server nemá ponětí o tom, jakou akci vyvolají zaslané fragmenty kódu. Mohou zavolat pouze metodu pro seřazení kolekce čísel, ale taky může uložit na disk serveru škodlivou aplikaci nebo využít server k nějaké jiné činnosti. Proto je nutné nejpozději ve fázi spuštění zkompilované aplikace ověřit, zdali fragmenty kódu obsahují pouze neškodný kód nebo jestli nevolají nepovolené metody.

Analyzátor kódu

První možností je zanalyzovat zaslané fragmenty kódu. Analyzátor by měl odhalit, jestli výsledkem fragmentu kódu nebude nějaká škodlivá akce, jako například spuštění nového procesu, otevření portu, vytvoření nebo smazání souboru na disku. Analyzátor používá například portál Courseware ZČU. Na tento portál studenti nahrávají vypracované úkoly. Portál po nahrání úkolu zdrojové kódy zkompiluje a spustí.

Naimplementovat analyzátor schopný detekce škodlivého kódu je velice obtížné. Programátor by musel mít dokonalé znalosti o tom, jaké všechny akce je potřeba zakázat, aby byla zaručena vysoká bezpečnost aplikace.

Java Security Policy

Java Security Policy je soubor pravidel, která definují všechny akce, jaké může aplikace vykonat. Java umožňuje pro již zkompilovanou aplikaci při startu explicitně definovat pravidla, na jaké akce má aplikace nárok. Pokud například nemá aplikace povoleno číst soubory z disku nebo navazovat socketové spojení, při startu aplikace dojde k chybě. Pokud aplikace nemá nastavené bezpečnostní omezení, může provádět všechny dostupné metody.

Jedná se o velice jednoduchý způsob, kterým lze docílit bezpečného běhu aplikace. Pomocí parametru `java.security.policy` lze definovat seznam povolených akcí. Pokud je tento parametr použit, o bezpečnost se již stará JVM. Drobnou nevýhodou tohoto přístupu je aplikování omezení aplikace až při jejím běhu. Seznam všech dostupných pravidel je zdokumentovaný na stránkách Oracle[6].

Docker

Spuštění mikrobenchmarku v Docker kontejneru by přineslo značnou výhodu v možnosti izolace spuštěného mikrobenchmarku a hostitelského systému. Docker umožňuje zakázat veškerou síťovou komunikaci s běžícím kontejnerem. Pak není možné se jakýmkoliv způsobem připojit na běžící kontejner skrze otevřený port. Pokud to není explicitně povoleno, z kontejneru se nelze připojit na disk hostitelského systému. Jestliže je kontejner takto nastaven, běžící mikrobenchmark může provádět jakékoliv změny pouze v rámci kontejneru. Nemá žádný způsob, jakým by mohl ovlivnit hostitelský systém. V tomto případě není nutné analyzovat spouštěný kód nebo zakazovat provedení určitých metod pomocí nastavení JVM.

Vytvoření Docker kontejneru je velice rychlé. Procesy běžící v rámci kontejneru nejsou zpomalovány oproti procesům běžícím přímo v hostitelském systému. Běh mikrobenchmarku v kontejneru by měl trvat stejnou dobu jako kdyby běžel mimo kontejner. Případně kdyby trval běh delší dobu, výsledky nejsou zkresleny, jelikož cílem mikroben-

[6] https://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html

chmarku je porovnat, jaký fragment kódu je rychlejší. Pro všechny měřené fragmenty kódu platí stejné výkonové podmínky.

Vyhodnocení

Na internetu nebyl nalezen žádný open-source analyzátor Java kódu. Implementace vlastního analyzátoru by byla příliš obtížná. Z tohoto důvodu nelze použít první možnost pro zabezpečení běhu mikrobenchmarku. Další možnost Java security policy by bylo možné použít. Pro běh mikrobenchmarku by se definovaly pouze bezpečné akce, čímž by byl zajištěn bezpečný běh mikrobenchmarku. Nicméně tímto přístupem by byl uživatel velice omezen. Mohl by ve svých fragmentech kódu použít pouze některé akce, které nejsou zakázané. Pokud by bylo nastaveno pravidlo, které by zakazovalo vytvoření souboru na disku, uživatel by nemohl vytvořit mikrobenchmark, který by mu porovnal, jaká metoda pro zápis do souboru je nejrychlejší. Takové omezení lze považovat za nedostatek použití Java Security Policy.

Poslední navrhovanou možností je spuštění mikrobenchmarku v Docker kontejneru. Jelikož by byly mikrobenchmarky izolovány od hostitelského systému, může uživatel v mikrobenchmarku provádět jakékoliv akce. Po ukončení mikrobenchmarku lze kontejner smazat, čímž dojde k odstranění veškerých stop po běhu mikrobenchmarku. Kontejner je možné spustit v interaktivním režimu. Pokud by měl administrátor podezření, že se v kontejneru děje něco zvláštního, díky aktivnímu interaktivnímu režimu se může do kontejneru připojit a podívat se například na změny na disku v rámci kontejneru.

Jedinou nevýhodou tohoto přístupu je zkomplikování procesu mikrobenchmarku. Server, kde běží aplikace, musí mít nainstalovaný Docker. Dále musí aplikace pro každý mikrobenchmark vytvořit kontejner, nakopírovat do něj `JAR` soubor s mikrobenchmarkem, po dokončení běhu získat z kontejneru výsledky měření a kontejner ukončit.

I přes zkomplikování procesu mikrobenchmarku je použití Dockeru nejlepší možností, jak zabezpečit běh mikrobenchmarku a neomezit uživatele, aby mohl používat všechny třídy a metody Javy. Na základě analýzy je spouštěn každý mikrobenchmark v Docker kontejneru.

4.2.4 Modul microbenchmark-service

Jak již je psáno výše, modul `microbenchmark-service` slouží pro zpracování kompletního procesu mikrobenchmarku. Proces mikrobenchmarku je velice komplikovaný. Jednotlivé fáze procesu jsou níže detailně rozebrány. V příloze B.2 je zakreslen kompletní proces pro provedení mikrobenchmarku.

Generování třídy

První fáze se týká vygenerování třídy se zadanými fragmenty kódu pro porovnání. Třída musí obsahovat anotace a konfiguraci frameworku JMH, který byl vybrán jako nejlepší framework pro tvorbu mikrobenchmarků v jazyce Java (viz kapitola 3.7.2). Aplikace používá předpřipravenou třídu obsahující JMH anotace. Dále se zde nacházejí speciální značky, které se v průběhu generování nahradí zadanými parametry od uživatele.

Aby bylo zadávání metod pro benchmarkování přívětivější, aplikace obsahuje funkci automatického importu potřebných knihoven. Díky tomuto importu nemusí uživatel při psaní metod uvádět importy, v jakých balíčcích se nacházejí použité třídy. Bohužel nebylo nalezeno vhodné hotové řešení, které by poskytovalo automatický import knihoven. Proto bylo potřeba automatický import naimplementovat.

Základem automatického importu je znalost, v jakém baličku se nachází hledaná třída. Aplikace obsahuje mapu všech tříd včetně jejich balíčků dostupných v Java 1.8. V první fázi se zadané fragmenty kódu zanalyzují a vyhledají se pouze názvy tříd. Název třídy se hledá na základě prvního velkého písmena pomocí regulárního výrazu. Nalezená třída je následně vyhledána v mapě všech tříd. Výsledek hledání může skončit celkem ve čtyřech stavech. Nalezená třída se nachází pouze v jednom balíčku, lze předpokládat, že uživatel chce použít právě nalezenou třídu a import je úspěšně dokončen. Dalším stavem je výskyt třídy ve více balíčcích. Typickým příkladem je třída `List`, která se v Java 1.8 nachází celkem ve třech balíčcích. V takovém případě je nutné požádat uživatele o vybrání, z jakého balíčku chce danou třídu použít. Dalším stavem je nalezená třída, která pochází z balíčku `java.lang`. Třídy z tohoto balíčku nevyžadují import. Poslední stav je nenalezení třídy v mapě všech tříd. V tomto případě se buď nejedná o název třídy, nebo se daná třída nenachází ve verzi Java 1.8. Jedná se o relativně jednoduché řešení, které je velice efektivní. Pokud aplikace již vlastní všechny potřebné importy, doplní je do předpřipravené třídy.

Mapa všech tříd se ve výchozím stavu čte ze souboru. Po prvním přečtení je nahrána do cache, aby se při generování každého mikrobenchmarku nemusel číst soubor z disku. Zároveň je stejným způsobem zacházeno se souborem obsahující seznam tříd, které nevyžadují import.

Kompilace projektu

Jestliže aplikace nahradila všechny značky v předpřipravené třídě, následuje proces vytvoření Maven projektu obsahující definici potřebných knihoven a vygenerovanou třídu. Cílem kompilace je vygenerovat spustitelný `JAR` soubor s knihovnami frameworku JMH. V průběhu kompilace je analyzován log, který poskytuje nástroj Maven. Kompilace může skončit chybou. Bohužel knihovna pro spuštění Maven procesu neposkytuje možnost jednoduše zjistit, že se v kompilaci vyskytla chyba a na jakém místě. Je nutné vypisované

informace analyzovat pomocí regulárního výrazu. Jestliže se vyskytne v průběhu kompilace chyba, aplikace získá z logu chybovou hlášku, kterou pošle v HTTP odpovědi zpátky uživateli. Odpověď obsahuje i číslo řádky, kde se chyba pravděpodobně vyskytuje. Proto je součástí odpovědi i kompletně vygenerovaná JMH třída, aby si mohl uživatel jednoduše zjistit, kde udělal v implementaci chybu. Pokud doběhne kompilace úspěšně, nástroj Maven vygeneruje spustitelný `JAR` soubor obsahující kompletní JMH mikrobenchmark.

Spuštění mikrobenchmarku

Jestliže se aplikace dostane do stavu, že vlastní vygenerovaný JMH mikrobenchmark, může začít s jeho vykonáním. V kapitole 4.2.3 je popsáno, proč je nutné, aby běžel mikrobenchmark v Docker kontejneru. Pro spuštění Docker kontejneru je potřeba použít Docker image. Pro účely mikrobenchmarku byl vytvořen nový image obsahující operační systém Alpine Linux. Jedná se o malou, jednoduchou a bezpečnou linuxovou distribuci. V Docker image je dále nainstalován OpenJDK verze 8, vytvořen adresář pro zkopírování `JAR` souboru a adresář pro uložení výsledku mikrobenchmarku. Kompletní definice Docker image je v příloze B.1.

Po vytvoření Docker kontejneru následuje zkopírování vygenerovaného `JAR` mikrobenchmarku přímo do Docker kontejneru. Následně je konečně mikrobenchmark spuštěn. Běh mikrobenchmarku je dlouhotrvající proces, zvláště pokud má nastaveno mnoho iterací. Proto je nutné uživatele průběžně informovat o aktuální prováděné iteraci mikrobenchmarku. Bohužel ke zjištění, v jaké fázi se mikrobenchmark nachází, je nutné, podobně jako u kompilace, analyzovat log poskytnutý JMH. Protože je předem známo kolik iterací bude celkem provedeno a dále existuje informace o aktuálně prováděné iteraci, lze jednoduše vypočítat, v jakém čase přibližně mikrobenchmark doběhne. Tato informace je uživateli zasílána společně s informací o aktuálně prováděné iteraci. Zároveň je tato informace uložena do databáze, kde se uchovávají data o všech aktuálně běžících měřeních. Stav mikrobenchmarku se ukládá do databáze z důvodu, aby administrátor aplikace měl přehled o aktuálně běžících mikrobenchmarcích a zároveň mohl zjistit, kdy přibližně mikrobenchmark doběhne. Administrátor má dále právo násilně ukončit jakýkoliv mikrobenchmark, pokud se mu bude zdát například, že běží příliš dlouho.

Pokud mikrobenchmark úspěšně doběhne, uloží JMH do Docker kontejneru soubor s výsledky ve formátu `JSON`. Výsledky jsou následně zkopírovány na disk serveru. Mikrobenchmark může skončit i chybou. Může být například vyhozena runtime výjimka. V takovém případě je uživateli zaslán chybový stav opět včetně kompletní spuštěné JMH třídy, aby mohl jednoduše identifikovat, jaký řádek způsobil chybu. Na závěr je zrušen běžící Docker kontejner, jelikož již nebude v budoucnu nikdy využíván.

Vyhodnocení výsledků

Při úspěšném doběhnutí mikrobenchmarku je vygenerován `JSON` soubor s naměřenými hodnotami. V poslední fázi procesu jsou naměřené hodnoty přečteny ze souboru a zpracovány. V odpovědi přijde uživateli informace o jednotlivých naměřených hodnotách v rámci každé iterace pro měřené metody, průměr z těchto naměřených hodnot a možná odchylka. Dále odpověď obsahuje informaci o tom, jaká metoda je nejrychleji zpracována. Naměřené časy mikrobenchmarku mohou být nepřesné, pokud v jednu chvíli běželo na serveru více mikrobenchmarků. Proto odpověď obsahuje i informaci o počtu současně zpracovávaných mikrobenchmarků v době, kdy běžel uživatelův mikrobenchmark. Na základě této informace lze případně uživatele varovat o možné nepřesnosti.

4.2.5 Modul microbenchmark-backend

Modul `microbenchmark-backend` obsahuje aplikační logiku především pro práci s databází. Dále se zde nachází implementace nutná k přihlašování uživatelů. Níže jsou rozebrány jednotlivé části modulu včetně schématu databáze.

Komunikace s databází

Pro práci s databází se využívá technika Objektově relační mapování (ORM). Jedná se o techniku zajišťující automatickou konverzi dat mezi relační databází a objektově orientovaným programovacím jazykem. Výhodou ORM techniky je odstínění programátora od SQL dotazů. Usnadňuje provádění tzv. CRUD[7] operací. Mezi jednu z největších výhod ORM je nezávislost aplikace na databázovém systému.

V aplikaci je použita specifikace Java Persistence API (JPA). Jedná se o standard popisující programátorské rozhraní (API) a chování knihoven pro objektově-relační mapování. V dnešní době existuje několik implementací, jako například Hibernate, EclipseLink, OpenJPA, DataNucleus nebo ObjectDB. Základem JPA je entita. Entita se ve většině případů rovná databázové tabulce. Pro nastavení vlastností entity se používají různé konfigurační anotace [4]. V aplikaci je použit Hibernate jako implementace standardu JPA.

Liquibase je open-source nástroj napsaný v programovacím jazyce Java sloužící pro verzování databází. Nástroj pracuje na principu změnových logů. Změnové logy lze vytvořit v různých formátech, jako například XML, YAML nebo jako SQL příkazy. Liquibase potřebuje ke svému fungování vytvořit dvě tabulky v databázi, kde si bude uchovávat informace, jaké změny jsou již v databázi aplikovány. Při startu aplikace se Liquibase podívá do své tabulky a v případě, že tabulka neobsahuje změny, které nástroj nalezl v projektu,

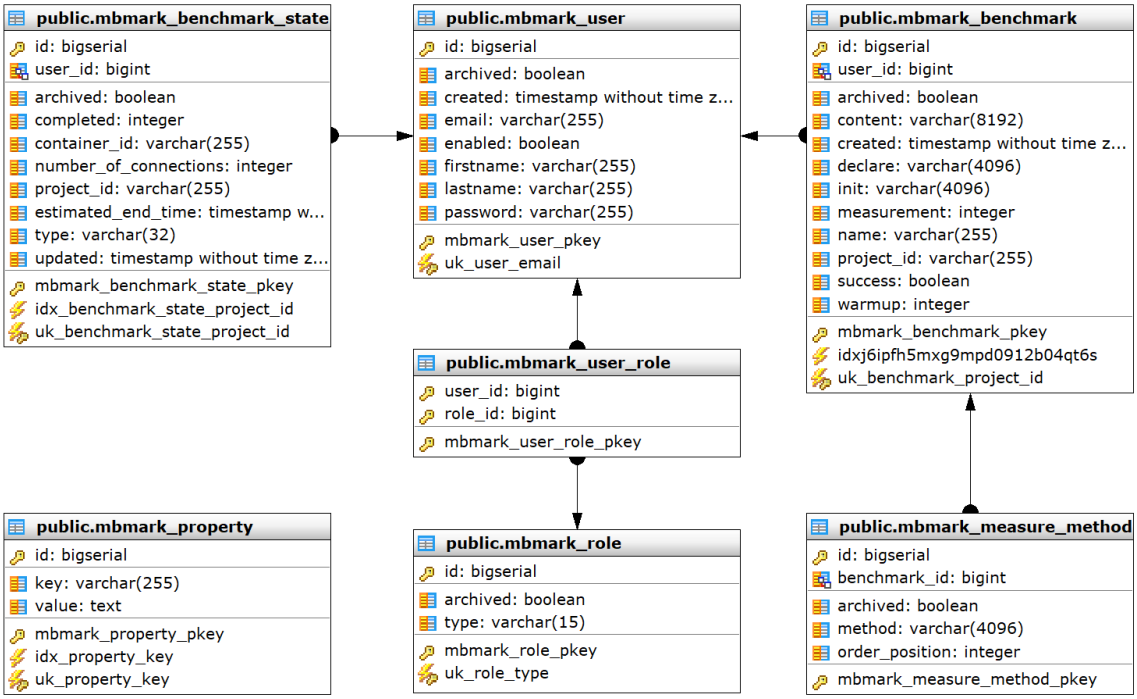
[7] CRUD (Create, Read, Update, Delete) je zkratka shrnující čtyři základní operace nad záznamem v databázi.

změny provede. Tímto způsobem lze velice bezpečně modifikovat databázovou strukturu tabulek.

V aplikaci je Liquibase použit pro vytvoření všech tabulek do databáze. Framework Hibernate sice zvládne vytvořit a modifikovat databázi, tento přístup ale není doporučován [2]. Framework modifikuje databázi na základě entit v aplikaci. Informace o změně v databázi pomocí Hibernate je uložena pouze v logu aplikace. Pokud by Hibernate po startu aplikace pozměnil například produkční databázi, mohl by nastat problém. To je důvod, proč se o kompletní správu nad databází stará Liquibase. Po startu aplikace se mimo vytvoření tabulek provedou změny pro naplnění určitých tabulek daty. Dojde k vytvoření všech možných rolí a uživatele s rolí administrátora.

Databázový model

Databáze obsahuje celkem sedm tabulek. Významově by bylo možné rozdělit databázi na dvě skupiny tabulek. První skupinou jsou tabulky týkající se benchmarků. Do druhé skupiny patří tabulky pracující s uživateli aplikace. Dále je vytvořena tabulka pouze pro uchovávaní různých konfigurací. ER model databáze je zobrazen na obrázku 4.2.



Obr. 4.2: ER model databáze

Tabulky databáze

V databázi existují celkem tři tabulky uchovávající data ohledně benchmarku. Tabulka `benchmark` obsahuje data o již proběhlých benchmarcích. Do této tabulky se ukládají

úspěšně i neúspěšně dokončené benchmarky. Je ukládána i kompletní uživatelova konfigurace benchmarku včetně fragmentů kódu pro měření. Fragmenty kódu se uchovávají v tabulce `measure_method`. Součástí benchmarku může být až N fragmentů, proto je v aplikaci vytvořená relace 1:N mezi těmito tabulkami. Poslední tabulkou patřící do skupiny je `benchmarks_state`. Tabulka slouží pro logování aktuálně běžících mikrobenchmarků. Na základě této tabulky se může administrátor podívat, jaké aktuálně běží mikrobenchmarky, kdo je spustil, v jaké jsou fázi a kdy přibližně doběhnou. Dále se v tabulce nachází informace, kolik mikrobenchmarků běží současně a můžou se vzájemně ovlivnit. Další důvod pro logování běžících mikrobenchmarků je umožnění administrátorovi násilně vypnout Docker kontejner, čímž dojde ke zrušení běhu benchmarku. Tato možnost se může hodit v různých situacích, jako například když bude spuštěn omylem mikrobenchmark běžící hodiny až dny. Data tabulky `benchmark_state` jsou při každém spuštění aplikace vymazána. Informace o aktuálně běžících benchmarcích by bylo možné ukládat i do jiného sdíleného serverového úložiště. Tento přístup by byl mnohem komplikovanější, protože by se musela řešit například synchronizace mezi vlákny.

Ke správě uživatelů slouží celkem tři tabulky. První tabulka `user` se používá pro uchování základních informací o jednotlivých uživatelích. Tabulka `role` obsahuje seznam všech rolí, jaké mohou být uživatelům přiřazeny. Jelikož uživatel může mít více rolí, pro přiřazení slouží spojovací tabulka `user_role`. Ve výchozím stavu obsahuje aplikace základní tři role - `ADMIN`, `USER` a `DEMO`. Práva jednotlivých rolí jsou popsány v kapitole 4.2.6.

Poslední tabulka `property` se používá pro uchování konfiguračních hodnot aplikace. Aplikace je v určitých oblastech konfigurovatelná. Pokud chce administrátor použít novější verzi Javy, do této tabulky provede změny týkající se aktualizace verze Javy, nově vytvořenou mapu všech tříd sloužící pro automatický import a dále seznam všech tříd z balíčku `java.lang`, které lze ignorovat. Dále se zde nachází verze knihovny JMH. Běžící benchmark může využít pouze určitou část paměti, aby nenastala situace, že jeden benchmark využije veškerou dostupnou paměť na serveru. Tato maximální hodnota se též nachází jako záznam v tabulce. Jako poslední lze nastavit, na jaký e-mail mají chodit skryté kopie nově registrovaných uživatelů.

Aplikace nemaže přímo data z databáze. Používá tzv. „Soft delete“ princip. Při zavolání metody pro smazání záznamu z tabulky nedojde k jeho fyzickému smazání, ale je mu nastavena hodnota sloupce `archived` na `true`. Jediná tabulka `property` tento princip nepoužívá.

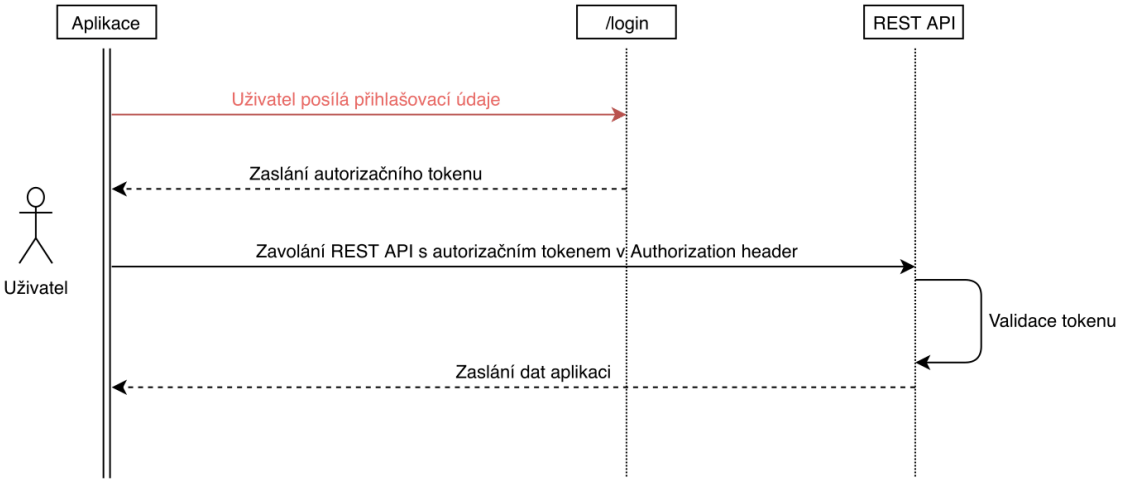
Přihlášení do aplikace

Přihlášení do aplikace využívá protokolu OAuth. Cílem protokolu je poskytnout bezpečnou autentizaci a autorizaci oproti API službám. Výhodou přihlašování pomocí protokolu

OAuth je nezávislost na typu aplikace. Pomocí tokenu může s API komunikovat webová, desktopová nebo mobilní aplikace.

Uživatel zašle backend aplikaci přihlašovací údaje. Po úspěšném ověření údajů posílá aplikace v odpovědi požadavku autorizační token. Jedná se o Bearer token. Pro další komunikaci musí být token součástí hlavičky požadavku. Backend aplikace ověří, jestli nebyl token upraven a zároveň jestli již neexpiroval. Pokud je vše v pořádku, vrátí odpověď na uživatelův požadavek. Proces přihlašování je znázorněn na obrázku 4.3.

Přihlašovací údaje uživatele jsou uloženy v databázové tabulce `user`. Přihlašovacím identifikátorem je e-mail. Heslo je v databázi zahashováno funkcí Bcrypt. Pokud přihlašovací údaje souhlasí, je následně uživateli vygenerován Bearer token obsahující jeho identitu a přiřazené role. Zároveň obsahuje informaci o době expirace. Standardní doba expirace tokenu je nastavená na dvě hodiny.



Obr. 4.3: OAuth proces

Registrace uživatele

Registrace nového uživatele probíhá dvoufázově. V první fázi dojde k vytvoření účtu uživatele s požadovanou rolí. Účet je ve stavu `disable`. V druhé fázi musí administrátor potvrdit registraci nového uživatele. Po potvrzení se účet přepne do stavu `active`. Dvoufázová registrace zabraňuje, aby mohl mít v aplikaci účet kdokoliv. Pokud vytváří nového uživatele administrátor, registrace je již jednofázová. Aplikace po každém kroku informuje e-mailem registrovaného uživatele, aby věděl, v jakém stavu je jeho účet. V posílaném e-mailu je nastavena skrytá kopie na e-mail uvedený v konfigurační tabulce `property`.

4.2.6 Modul microbenchmark-web

Poslední modul aplikace se nazývá `microbenchmark-web`. V modulu se nachází třída obsahující metodu `main()`. Jelikož se jedná o Spring Boot projekt, třída obsahuje důležité anotace potřebné pro konfiguraci. Baliček `config` obsahuje další důležité konfigurace aplikace, jako například konfiguraci cache paměti, nástroje Swagger, websocketového spojení a zabezpečení. Nejdůležitější funkcí modulu je definování REST API endpointů, aby bylo možné aplikaci používat.

REST API

Representational State Transfer (REST) je architektura rozhraní navržená pro distribuované prostředí. REST je možnost jak snadno komunikovat se serverem pomocí jednoduchých HTTP volání. Nejčastěji je použit pro komunikaci HTTP protokol. RESTová aplikace používá HTTP požadavky pro práci s daty. Nejčastěji implementuje typy požadavků pro vytvoření (`POST`), aktualizaci (`PUT`), čtení (`GET`) a smazání (`DELETE`). Jedná se o základní CRUD operace. Díky těmto zmíněným požadavkům je REST API velice jednoduché na pochopení a používání. Existují i další HTTP požadavky, jako například `PATCH`, `OPTIONS`, `CONNECT`, `TRACE` a `HEAD`.

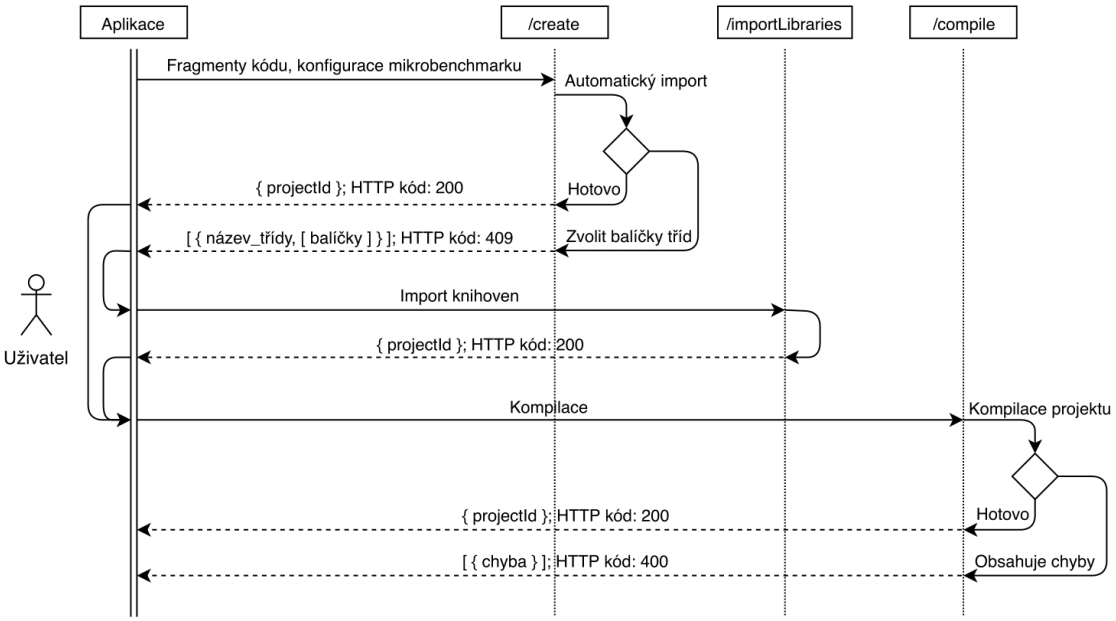
Pro výměnu dat mezi klientem a serverem lze použít několik datových formátů. Nejpoužívanějším formátem je `JSON`. Jedná se o jednoduchý a snadno čitelný formát dat. Jeho syntaxe je velice podobná syntaxi objektů v Javascriptu. Právě tento datový formát používá aplikace pro přijetí požadavku a produkování odpovědi.

Implementované REST API

REST API definuje celkem sedm tříd. Jednotlivé endpointy používají jeden z HTTP typu požadavku - `GET`, `POST`, `PUT` a `DELETE`. Všechny REST endpointy mají nastavenou validaci, jaké hodnoty musí obsahovat požadavek pro úspěšně přijetí. Pokud není validace úspěšná, odpovědět na požadavek má HTTP stavový kód 400 včetně popisku, kvůli jaké hodnotě není požadavek validní.

První třída obsahuje REST endpointy týkající se provedení mikrobenchmarku. Pro kompletní provedení mikrobenchmarku je nutné provést minimálně dvě volání. První volání slouží k vytvoření JMH třídy, ve které jsou nahrazeny značky daty od uživatele. Součástí této operace je i provedení automatického importu. Pokud automatický import doběhne bez problémů, v HTTP odpovědi se uživateli vrátí vygenerované ID mikrobenchmarku. Pokud ale bude nutné určit, z jakého balíčku se má provést import tříd, vrátí se uživateli nalezené třídy. V takovém případě je potřeba provést druhé volání obsahující třídy včetně balíčků, které se naimportují do vygenerované JMH třídy. Třetí volání slouží

pro spuštění kompilace vygenerovaného mikrobenchmarku. Pokud skončí kompilace chybou, uživateli se vrátí v odpovědi chybová hláška. Jestli ale proběhne kompilace úspěšně, mikrobenchmark je připraven ke spuštění. HTTP volání pro vytvoření, import knihoven a kompilaci benchmarku jsou zobrazeny na obrázku 4.4. Jak zahájit spuštění mikrobenchmarku je popsáno níže. Poslední endpoint v této části slouží pro násilné ukončení běžícího mikrobenchmarku.



Obr. 4.4: HTTP volání pro vytvoření mikrobenchmarku

Další třída obsahuje pouze jeden REST endpoint, který slouží pro získání všech aktuálně běžících benchmarků.

Pro správu již proběhlých benchmarků jsou připraveny celkem čtyři REST endpointy. První endpoint slouží pro získání všech proběhlých benchmarků. Druhý endpoint se používá pro získání detailu konkrétního benchmarku na základě jeho ID. Detail benchmarku obsahuje mimo základních informací i odkaz, kde lze vygenerovaný `JAR` soubor stáhnout. `JAR` soubor lze následně spustit na různých počítačích a porovnat naměřené hodnoty v závislosti na hardware počítače. Třetí endpoint se využívá k přiřazení již proběhlého benchmarku konkrétnímu uživateli. Poslední slouží ke smazání konkrétního benchmarku.

Další třída slouží pro správu uživatelů. Podobně jako u správy benchmarků i zde jsou připraveny REST endpointy pro získání konkrétního uživatele podle ID nebo seznam všech uživatelů. Dále jsou vytvořeny endpointy pro vytvoření nového uživatele nebo editaci existujícího uživatele.

Třída pro práci s rolemi poskytuje pouze jeden REST endpoint. Umožňuje získat seznam všech vytvořených rolí, které lze přiřadit uživatelům.

Předposlední třída se používá pro správu konfiguračních hodnot aplikace. Jedná se o hodnoty uložené v databázové tabulce `property`. Ve třídě se nacházejí REST endpointy pro získání konkrétní konfigurační hodnoty na základě ID nebo pro získání všech nastavených konfiguračních hodnot. Dále obsahuje endpoint pro vytvoření nebo úpravu konfigurační hodnoty. Poslední slouží ke smazání konfigurační hodnoty.

Poslední třída slouží pro manipulaci konfiguračních hodnot využívaných automatickým importem. Ve třídě jsou definovány celkem tři endpointy. První slouží pro získání aktuálně nastavené mapy tříd z databáze. Další endpoint se využívá k vytvoření nové mapy tříd. Do parametru je nutné zadat cestu, kde se na disku nachází `JAR` soubory, ze kterých se má vytvořit mapa. Jednotlivé `JAR` soubory jsou zanalyzovány a nalezeny všechny dostupné třídy. Poslední endpoint lze použít pro smazání mapy tříd z databáze.

Websocket

Websocket je protokol poskytující obousměrný komunikační kanál přes jediné TCP spojení. Protokol slouží pro výměnu zpráv mezi klientskou a serverovou aplikací. Používá se převážně pro real-time přenos dat mezi serverem a aplikací.

Websockety se v aplikaci používají pro zasílání informace o aktuální fázi zpracovávaného mikrobenchmarku. Mikrobenchmark může běžet desítky minut až hodiny. Proto je vhodné uživatele informovat, v jaké fázi se mikrobenchmark aktuálně nachází a kdy pravděpodobně doběhne.

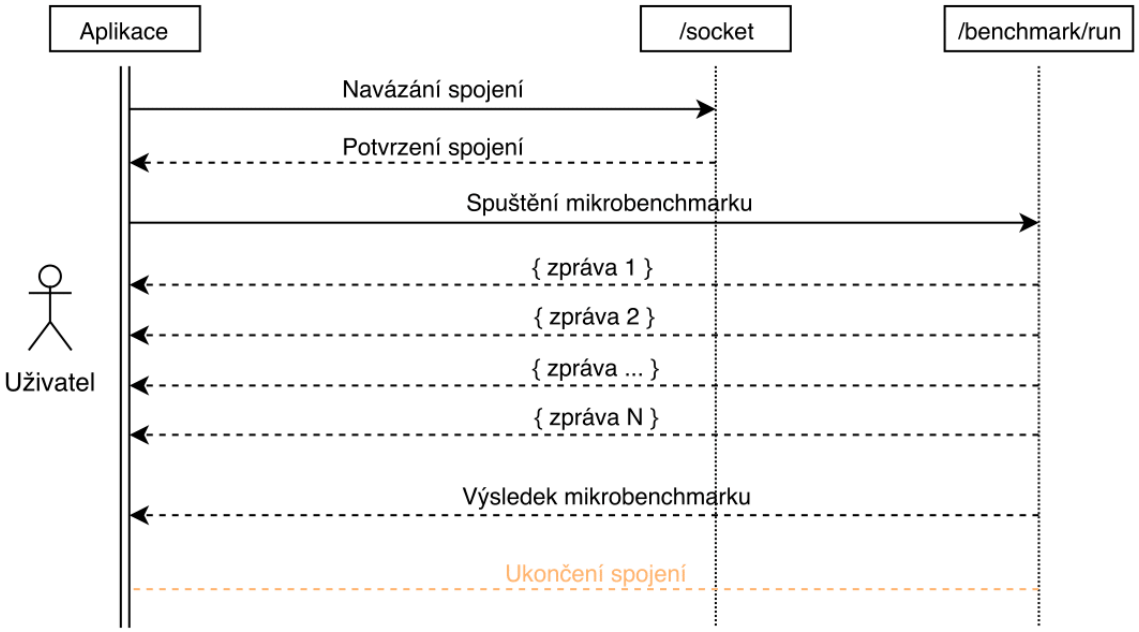
Pomocí RESTu by se informování realizovalo velice obtížně, protože REST funguje na principu požadavek-odpověď. Frontendová aplikace by musela posílat pravidelně požadavky pro získání aktuálního stavu benchmarku, protože backendová aplikace nemá možnost, jak samovolně zaslat informaci frontend aplikaci.

Frontend aplikace naváže s backend serverem websocketové spojení pomocí kterého lze následně vyměňovat zprávy mezi frontend a backend aplikacemi. Tímto způsobem může backend aplikace kdykoliv poslat informační zprávu frontend aplikaci o aktuálně prováděné fázi mikrobenchmarku.

Využití websocketu

Pro zahájení vykonání mikrobenchmarku se již posílá zpráva pomocí websocketu. Přijetím zprávy se provede metoda z modulu `microbenchmark-service`, která spustí Docker kontejner a zahájí mikrobenchmark. V této metodě se analyzuje průběžně získávaný JMH log. Pokud dojde k zahájení nové fáze mikrobenchmarku, backend aplikace pošle zprávu pomocí navázaného websocketového spojení frontend aplikaci. Pokud mikrobenchmark doběhne úspěšně, backend aplikace zašle získané naměřené hodnoty. V případě neúspěšného dokončení zašle aplikace zprávu obsahující vyhozenou runtime výjimku. Na závěr

backend aplikace ukončí websocketové spojení. Obrázek 4.5 znázorňuje zasílané zprávy pomocí websocketového spojení.



Obr. 4.5: Zpracování mikrobenchmarku pomocí websocketu

Dokumentace REST API

Aby bylo zřejmé, jakým způsobem lze aplikaci používat, musí existovat REST API dokumentace. V dnešní době je velice populární nástroj Swagger, který je schopen vygenerovat přehlednou REST API dokumentaci. Swagger je schopen pomocí knihovny spolupracovat s frameworkem Spring Boot.

Swagger si zanalyzuje zdrojové kódy aplikace. Výstupem analýzy jsou všechny nalezené REST endpointy. U každého endpointu je zjištěn typ požadavku, komentáře, parametry, vstupní nebo výstupní objekty, návratová hodnota a další informace. Endpoint může vracet více typu odpovědí. Odpovědi lze rozdělit na základě jejich stavových kódů. Pokud v rámci vyvolané akce proběhne vše v pořádku, endpoint vrací odpověď se stavovým kódem 200. Pokud se ale vyskytne nějaká chyba, je vrácena odpověď s příslušným chybovým stavovým kódem.

Swagger vytvoří nový REST endpoint, který obsahuje výslednou REST API dokumentaci ve formátu `JSON`. Pro zobrazení dokumentace poskytuje Swagger i UI aplikaci, do které lze nahrát vygenerovaný `JSON`. Zároveň lze Swagger aplikaci nastavit, aby si automaticky získala dokumentaci a zobrazila ji uživateli. Náhled Swagger UI aplikace je přiložen v příloze B.3.

Zabezpečení

Spring používá k zabezpečení REST API knihovnu `Spring security`. Pomocí této knihovny lze jednoduše definovat, jaké role musí uživatel vlastnit, aby mohl použít daný endpoint. Pokud uživatel nemá přiřazenou požadovanou roli, jeho požadavek je zamítnut a odpověď obsahuje stavový kód 403. Pokud se pokusí uživatel zavolat zabezpečený REST endpoint a není přihlášen, v odpovědi je uveden stavový kód 401. Ve Swagger dokumentaci obsahuje každý endpoint informaci o tom, jakou roli uživatel potřebuje, aby mohl daný endpoint zavolat.

V aplikaci jsou vytvořeny celkem tři role - `ADMIN`, `USER` a `DEMO`. Nepřihlášený uživatel má možnost pouze provést mikrobenchmark a zaregistrovat se. Role `DEMO` nemá oprávnění provádět téměř žádnou akci. Nemá dokonce ani právo na provedení mikrobenchmarku. Uživatel s touto rolí slouží pro prohlížení již provedených mikrobenchmarků, které jsou uživateli přiřazené. V reálném provozu budou tento účet využívat například studenti, kteří si budou chtít prohlédnout několik ukázkových provedených mikrobenchmarků. Právo spouštět mikrobenchmarky nemá z důvodu, aby uživateli nemohl přidávat nové benchmarky. Dále se může podívat na základní informace o svém učtu.

Uživatel s rolí `USER` má právo provádět základní operace. Může spustit benchmark, podívat se na všechny své již proběhlé benchmarky a smazat vybraný benchmark. Dále má právo na získání informací o svém učtu a editovat je.

Poslední role je `ADMIN`. Uživatel s touto rolí může volat všechny REST endpointy aplikace. Oproti roli `USER` může prohlížet všechny již proběhlé bechmarky. Pokud endpoint pro vrácení všech endpointů zavolá uživatel s rolí `USER`, jsou mu vráceny pouze jeho benchmarky. Pokud stejný endpoint zavolá uživatel s rolí `ADMIN`, jsou mu vráceny benchmarky všech uživatelů. Role `ADMIN` má dále možnost přiřadit již proběhlý benchmark jinému uživateli. Dále má právo se podívat, jaký benchmarky aktuálně běží a případně může násilně zrušit běh benchmarku. Administrátor si může získat seznam všech zaregistrovaných uživatelů. V poslední řadě má právo na editaci konfiguračních hodnot aplikace.

Kromě REST API je zabezpečen i websocketový endpoint pro spuštění benchmarku. Nepřihlášený uživatel nebo uživatel s rolí `ADMIN` nebo `USER` má právo poslat zprávu na tento endpoint.

Knihovna `Spring security` se využívá i pro přihlášení uživatelů a generování tokenů.

4.3 Testování aplikace

Aplikace je otestována na školním virtuálním serveru běžícím na adrese `147.228.63.36`. Server je dostupný pouze ze školní sítě.

4.3.1 JUnit testy

Funkčnost aplikace je otestována pomocí 66 JUnit testů. Téměř všechny testy slouží k otestování REST API. Jedná se spíše o integrační testy než o jednotkové testy. Výhodou testování přímo RESTového rozhraní je otestování několika částí aplikace v rámci jednoho testu. Je otestována aplikační logika včetně akcí spojených s databází. Pro testování je využita databáze `H2`, která běží v paměti. Pro každý test dochází k vytvoření nové databáze obsahující přesně daná testovací data.

4.3.2 Frontend aplikace

Cílem práce je vytvoření backend aplikace pro porovnání fragmentů kódu. Jak je již popsáno výše, provedení mikrobenchmarku je složitý proces, který obsahuje dva až tři HTTP požadavky, navázání websocketového spojení a výměnu websocketových zpráv. Provádět všechny uvedené kroky skrze například program Postman[8] by bylo velice nepřívětivé.

K účelu testování vzniklé backendové aplikace je vytvořena jednoduchá frontendová aplikace ve frameworku Angular 5. Pomocí frontendové aplikace lze jednoduše vytvořit a provést mikrobenchmark. Aplikace zároveň testuje, jestli funguje správně výměna zpráv pomocí websocketů. Aplikace průběžně informuje uživatele o aktuálně prováděné operaci. Vzhled aplikace je přiložen v příloze C.1. Příloha obsahuje obrázky různých částí frontend aplikace (C.1, C.2 a C.3). Aplikace umí pouze přihlásit uživatele a spustit benchmark. Jiné dostupné REST endpointy nevolá. Aplikace je dostupná na adrese `http://147.228.63.36/mbmark`.

4.3.3 Testovací benchmarky

V této kategorii jsou popsány různé mikrobenchmarky, které měří stejné fragmenty kódu. Každý mikrobenchmark je spuštěn s jinou konfigurací nebo za jiných podmínek.

Testovací prostředí

Server je vytvořen pouze pro účely testování implementované aplikace. Na serveru je nainstalován operační systém Linux Debian 4.9.110-3+deb9u2. Server je tvořen dual procesorem typu Intel(R) Xeon(R) CPU E5-4620 v2 @ 2.60GHz. Server obsahuje celkem 8 GB paměti.

Na serveru běží základní procesy pro běh operačního systému. Dále zde běží vytvořená aplikace, PostgreSQL databáze a Docker. Žádné další procesy na serveru neběží, proto by benchmark neměl být příliš ovlivňován okolními procesy.

Nástroj JMH pro provedení benchmarku je v rámci testování použit ve verzi 1.21.

[8] Program pro posílání HTTP požadavků.

Eliminace mrtvého kódu

V kapitole 3.2.5 je popsáno, jak JVM optimalizuje prováděný kód od mrtvého kódu. Pokud benchmark obsahuje metodu, která například jen seřadí čísla v poli a žádné další operace s polem neprovádí, JVM může vyhodnotit celou metodu jako mrtvý kód, jelikož nepřináší žádnou hodnotu pro vykonávaný kód.

Aplikace nabízí možnost zavolat metodu `consume()` třídy `Blackhole`, čímž dojde k zamezení, aby JVM vyhodnotil veškerý kód pracující s polem jako mrtvý. Ukázka 4.1 obsahuje ukázku použití metody `consume()`.

```
1 Arrays.sort(array);
2 blackhole.consume(array);
```

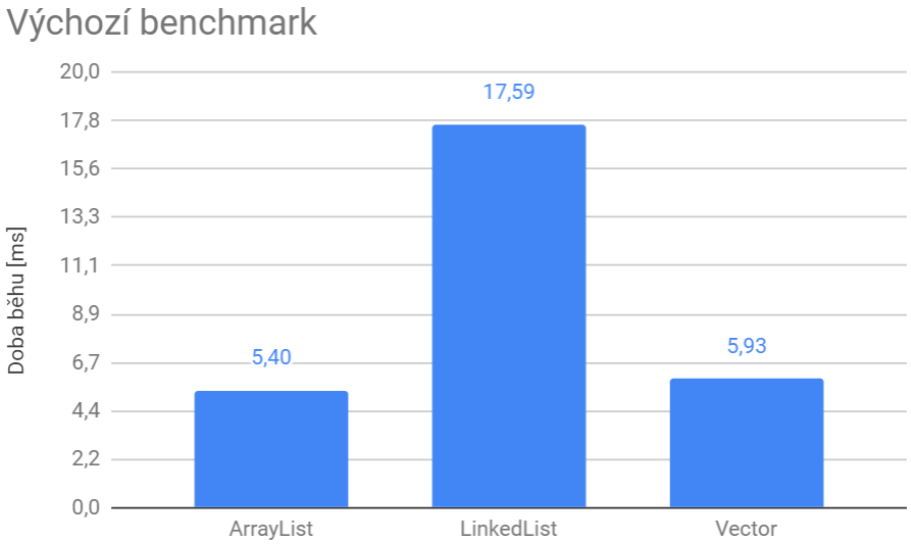
Ukázka 4.1: Zamezení eliminaci mrtvého kódu

Výchozí benchmark

Výchozí benchmark je spuštěn v ideálních podmínkách a s vhodným nastavením. Před skutečným měřením je proveden dostatečný počet iterací fázi warmup, aby došlo k aplikování JVM optimalizací. Dále je spuštěn benchmark v několika desítkách iterací pro získání co nejpřesnějšího výsledku. Je optimální, když na serveru běží pouze jeden benchmark v danou chvíli. Výchozí benchmark slouží pro porovnávání s ostatními benchmarky provedenými v horších podmínkách.

Pro testování je spuštěn benchmark, jehož jednotlivé fragmenty kódu obsahují metodu `contains()` na různých typech kolekcí. Jedná se o kolekce `ArrayList`, `LinkedList` a `Vector`. Každá kolekce je naplněna celkem 1 500 000 čísly. Nad každou kolekcí je zavolána metoda `contains(1499001)`. Zadání benchmarku se nachází v příloze C.1.1. Každý fragment kódu je proveden celkem 15x ve fázi warmupu. Následně je spuštěn 20x již ve fázi měření. Cílem benchmarku je zjistit, nad jakou kolekcí je nejrychleji provedena metoda `contains()`.

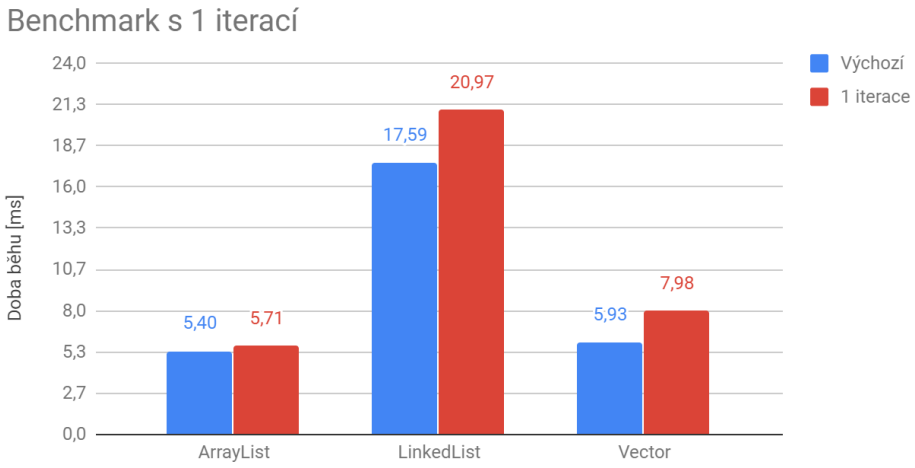
Benchmark běžel téměř 20 minut. Využil celkem přibližně 200 MB paměti. Nejrychleji se metoda `contains()` provedla nad kolekcí `ArrayList`. O přibližně 6 desetin milisekundy je pomalejší kolekce `Vector`. Téměř 3x delší čas trvalo provést metodu nad kolekcí `LinkedList`. Naměřené výsledky jsou k nahlédnutí v příloze C.1.2 nebo v grafu 4.6. Výsledek lze považovat za nejpřesnější, jelikož benchmark běžel v ideálních podmínkách a s dostatečným počtem opakování nad velkou množinou dat. S tímto výsledkem jsou porovnávány níže provedené benchmarky.



Obr. 4.6: Výsledky výchozího benchmarku

Benchmark s 1 iterací

Benchmark měří stejné fragmenty kódu jako výše provedený benchmark. Liší se v počtu iterací ve fázi warmup a fázi měření. V každé fázi je provedena pouze jedna iterace. To znamená, že se benchmark provede 1x nanečisto a 1x bude změřen jeho běh. Očekává se, že JVM nezoptimalizuje měřené fragmenty kódu jako v předchozím měření. Výsledek by měl obsahovat pro každou kolekci vyšší naměřené časy.



Obr. 4.7: Výsledky benchamrku s 1 iterací

Graf 4.7 porovnává výsledky výchozího a tohoto benchmarku. Sloupce označené červenou barvou reprezentují tento provedený benchmark. Z grafu lze vyčíst, že každá kolekce

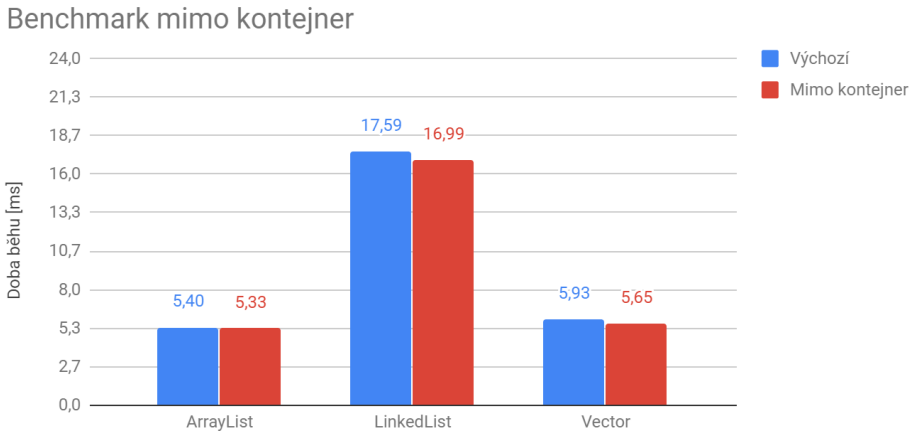
se prováděla déle než ve výchozím benchmarku. U kolekce `ArrayList` je rozdíl v desetinách milisekund, což lze považovat za minimální rozdíl. Pro další kolekce se naměřené časy liší v jednotkách milisekund.

Porovnáním časů výchozího a tohoto benchmarku lze prohlásit, že by měl být benchmark proveden nejprve několikrát nanečisto ve fázi warmup a následně opakovaně ve fázi měření. Provedením několika iterací ve fázi warmup se zajistí optimalizace kódu. Více naměřených časů pro jeden fragment může sloužit k ignorování iterace, která trvala delší dobu například z důvodu JVM optimalizace.

Běh mimo kontejner

Vytvořená aplikace spouští každý benchmark uvnitř Docker kontejneru kvůli zajištění bezpečnosti hostitelského systému. Popis fungování Docker kontejneru je popsán v kapitole 4.2.1. Jelikož procesy běžící v rámci Docker kontejneru jsou jistým způsobem virtualizovány, lze předpokládat, že běh procesu uvnitř kontejneru bude trvat déle než mimo kontejner.

Aplikace vygenerovaný JMH projekt zkompiluje. Výsledkem kompilace je spustitelný `JAR` soubor. Tento soubor je zkopírován do Docker kontejneru a spuštěn. V rámci testování je spuštěn `JAR soubor` výchozího benchmarku mimo Docker kontejner přímo na serveru, kde benchmarky spouští aplikace. Tímto testem je možné zjistit, jak moc je běh benchmarku ovlivněn kontejnerem a virtualizací.



Obr. 4.8: Výsledky benchamrku mimo kontejner

Graf 4.8 porovnává výsledky výchozího a tohoto benchmarku. Sloupce označené červenou barvou reprezentují benchmark běžící mimo Docker kontejner. Z grafu lze zjistit, že běh benchmarku běžícího uvnitř kontejneru může být zpomalen oproti běhu benchmarku mimo kontejner. Zpomalení je o pouhé desetiny milisekund, což je velice minimální až zanedbatelné. I kdyby byl benchmark zpomalen kontejnerem více, stále se jedná o validní naměřené hodnoty, protože kontejnerem jsou ovlivněny všechny měřené fragmenty kódu

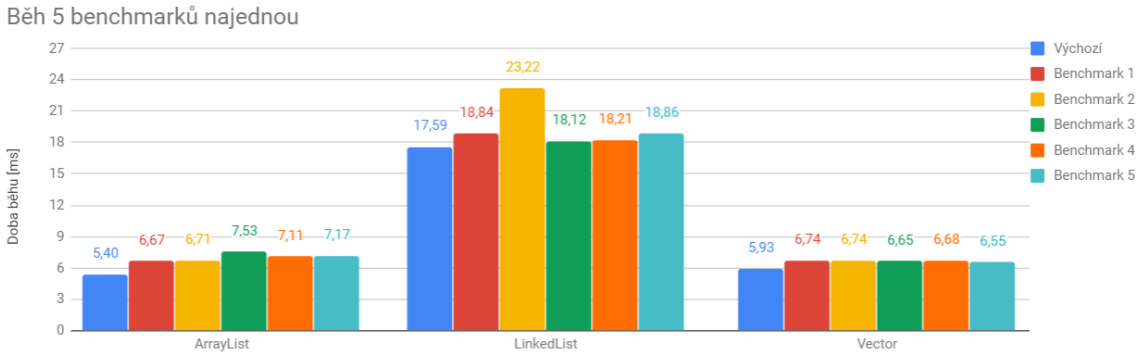
stejně. Na základě provedeného testu lze prohlásit, že Docker kontejner nemá žádné negativní vlastnosti pro běh benchmarku.

Zpomalení může být způsobeno i běžící backendovou aplikací, která benchmark spustila. Aplikace získává a analyzuje JMH log. Ze získaných informací provádí další různé operace. Aplikace například vypočítává přibližný čas dokončení benchmarku nebo ukládá průběžné informace do databáze. I tato aktivita může být důvodem, proč jsou naměřené časy mimo Docker kontejner menší. Aplikace v průběhu běhu benchmarku musí využívat procesor, který se proto nemůže věnovat pouze benchmarku.

Více benchmarků najednou

Webovou aplikaci může používat několik uživatelů najednou. Může nastat situace, že se provádí na serveru několik benchmarků v jednu chvíli. Benchmarky se musí střídat o procesorový čas, což vede ke vzájemnému ovlivňování. Bohužel nelze nastavit, aby se jedno procesorové jádro věnovalo pouze jednomu benchmarku. Pokud aplikace provádí pouze jeden benchmark a najednou začne vykonávat další, výsledky obou mohou být ovlivněny.

V rámci testování je spuštěno celkem pět benchmarků současně. Cílem testu je zjistit, jestli budou výsledky jednotlivých benchmarků odlišné od výchozího benchmarku.



Obr. 4.9: Běh více benchmarků najednou

V grafu 4.9 jsou zobrazeny naměřené hodnoty všech spuštěných benchmarků. U kolekce `ArrayList` jsou výsledky všech benchmarků vyšší než u výchozího benchmarku. Liší se v jednotkách milisekund. Podobné výsledky vyšly i u kolekce `Vector`. V každém benchmarku je naměřený čas vyšší než u výchozího benchmarku. Jednotlivé časy jsou velice vyrovnané, liší se v desetinách milisekundy. Dokonce poslední tři časy kolekce `Vector` mají nižší čas než poslední tři benchmarky kolekce `ArrayList`. U poslední kolekce `LinkedList` jsou všechny naměřené hodnoty vyšší než u výchozího benchmarku. Až na benchmark číslo dva jsou naměřené hodnoty vyrovnané. Liší se pouze v desetinách milisekundy. Benchmark číslo dva kolekce `LinkedList` dosáhl vyšší hodnoty oproti výchozímu benchmarku o necelých 6 milisekund.

Metoda `contains()` u kolekcí `ArrayList` a `Vector` je zpracována téměř za stejnou dobu. Proto může nastat situace, že v jednom benchmarku má rychlejší čas kolekce `ArrayList` a v jiném kolekce `Vector`. Na základě hodnot z provedených benchmarků lze prohlásit, pokud aplikace provádí současně několik benchmarků, výsledky jednotlivých měření mohou být ovlivněny. Nicméně v tomto testovacím případu se jednalo převážně o desetiny až jednotky milisekund, což je minimální rozdíl. Aplikace posílá společně s výsledky i informaci o tom, kolik benchmarků běželo společně s dokončeným benchmarkem.

Bezpečnost

Jak již je zmíněno výše, u online aplikace je nutné, aby testované benchmarky byly bezpečné a nemohly provádět škodlivý kód. Aby běžící benchmark neohrozil server je zajištěno izolací pomocí Docker kontejneru.

Další způsob jak zneužít benchmark ke škodlivé činnosti je využití internetového připojení. Útočník by mohl využít výpočetní výkon serveru například k vyřešení složitého výpočtu. Cílem benchmarku by nebylo zjistit, jak dlouho trvá zpracovat danou operaci, ale využít benchmark k provedení složitého výpočtu. Výsledek výpočtu by si následně mohl získat skrze internet. V Docker kontejneru lze nastavit, aby v kontejneru nebylo dostupné internetové připojení. Zakázané internetové připojení je nastaveno v každém spuštěném kontejneru. Útočník nemá možnost získat výsledek jeho výpočtu skrze internet. Níže je uvedena ukázka kódu, která dokazuje nemožnost připojení k internetu v průběhu vykonávání benchmarku.

```
1 try {
2     URL url = new URL("http://www.google.com");
3     URLConnection connection = url.openConnection();
4     connection.connect();
5 } catch (IOException e){
6     throw new RuntimeException("No internet connection");
7 }
```

Ukázka 4.2: Otestování internetového připojení

Ukázka 4.2 obsahuje kód, který se pokusí připojit na URL `http://www.google.com`. Pokud se mu to nepodaří, vyhodí runtime výjimku. Pro otestování internetového připojení je tento kód proveden jako klasický benchmark. Jakmile se začne benchmark vykonávat, benchmark vyhodí výjimku, ukončí běh a aplikace informuje uživatele o vyhozené výjimce. Tímto způsobem je otestováno, že se není možné připojit v průběhu benchmarku k internetu.

Útočník by se mohl dále chtít připojit na server skrze port, který si otevře implementací benchmarku. Samotný server je vystaven do internetu. Tohoto faktu by mohl útočník využít a otevřít si port, pomocí kterého by se následně na server připojil. Nicméně port by byl otevřen v Docker kontejneru, který je nastaven, aby nevystavoval žádné porty. Díky této vlastnosti nemá útočník možnost se připojit na port, který si sám otevřel.

Bohužel nelze zabránit útočníkovi ve vykonání škodlivého kódu, ale je možnost mu znemožnit získat výsledku jeho operace.

4.3.4 Zabezpečení REST API

REST API musí být zabezpečeno, aby neoprávněný uživatel nemohl provádět různé akce aplikace. Zabezpečení REST API je popsáno v kapitole 4.2.6.

Jedna z podmínek zabezpečení je zakázat uživateli s rolí `DEMO` možnost provádět benchmarky. Pokud uživatel s rolí `DEMO` pošle požadavek na vytvoření benchmarku, odpověď na požadavek obsahuje stavový kód `403`. Kompletní odpověď na požadavek je zobrazena v ukázce 4.3. Stejný typ odpovědi se vrací v odpovědi i pro endpointy na import knihoven, kompilaci a pro websocketovou zprávu na zahájení benchmarku. Pokud je uživatel přihlášen a nemá právo na daný endpoint, je mu vrácena právě tato odpověď. Odpověď obsahuje tzv. unixový čas[9], stavový kód, typ chyby, chybovou zprávu a zavolaný REST endpoint.

```
{
    "timestamp": 1556698890242,
    "status": 403,
    "error": "Forbidden",
    "message": "User is not allowed access this url",
    "path": "/api/project/create"
}
```

Ukázka 4.3: Neoprávněná akce - vytvoření benchmarku

Pokud nepřihlášený uživatel zavolá zabezpečený endpoint, je mu vrácena odpověď v ukázce 4.4. Odpověď má stejnou strukturu jako předchozí popsaná odpověď.

```
{
    "timestamp": 1556700951143,
    "status": 401,
    "error": "Unauthorized",
    "message": "No message available",
    "path": "/api/benchmarkState"
}
```

Ukázka 4.4: Neoprávněná akce - nepřihlášený uživatel

Některé endpointy může zavolat pouze přihlášený uživatel. V průběhu zpracování požadavku mohou být navíc zohledněny uživatelovy role. Pokud uživatel s rolí `USER` nebo `DEMO` požaduje vrátit data, je nutné uživateli zaslat pouze jeho data. Pokud stejný endpoint zavolá uživatel s rolí `ADMIN`, odpověď požadavku obsahuje data všech uživatelů. Tento typ zabezpečení obsahuje endpoint pro vrácení všech benchmarků (`GET /api/benchmarks`). Pokud endpoint zavolá uživatel s rolí `USER` nebo `DEMO`, vrátí se mu odpověď v ukázce 4.5. Ukázka obsahuje pouze určité informace z vrácené odpovědi. Je vidět, že odpověď obsahuje provedené benchmarky pouze uživatele s ID 3.

[9]Unixový čas je systém pro označení časových okamžiků. Systém identifikuje časové okamžiky pomocí počtu sekund uplynulých od 1. ledna 1970.

```
[
    {
        "id": 2,
        "projectId": "20b38910-e7fa-41fa-a14c-d8d6843533ed",
        "user": {
            "id": 3,
            "email": "demo@mbmark.cz"
        }
    },
    {
        "id": 4,
        "projectId": "a7d8c03d-5f03-4c86-93c3-8b0bd3a0fe8a",
        "user": {
            "id": 3,
            "email": "demo@mbmark.cz"
        }
    }
]
```


Ukázka 4.5: Benchmarky uživatele s rolí DEMO/USER

Pokud stejný endpoint zavolá uživatel s rolí `ADMIN`, aplikace vrátí všechny provedené benchmarky. Odpověď v ukázce 4.6 se skládá opět pouze z vybraných atributů objektu. Odpověď obsahuje provedené benchmarky všech uživatelů nebo benchmarky provedené nepřihlášeným uživatelem.


```
[
    {
        "id": 1,
        "projectId": "e6277036-07fe-44ac-9895-95a33ca68137",
        "user": {
            "id": 2,
            "email": "user@mbmark.cz"
        }
    },
    {
        "id": 2,
        "projectId": "20b38910-e7fa-41fa-a14c-d8d6843533ed",
        "user": {
            "id": 3,
            "email": "demo@mbmark.cz"
        }
    },
    {
        "id": 3,
        "projectId": "087d106f-81b0-4c0d-9669-f6ea93b0c4bd",
        "user": null
    },
    {
        "id": 4,
        "projectId": "a7d8c03d-5f03-4c86-93c3-8b0bd3a0fe8a",
        "user": {
            "id": 3,
            "email": "demo@mbmark.cz"
        }
    }
]
```


Ukázka 4.6: Všechny provedené benchmarky

Endpoint vrátí uživateli s rolí `ADMIN` navíc benchmarky s ID `1` a `3`. Jestliže zavolá uživatel s rolí `USER` nebo `DEMO` endpoint `GET /api/benchmarks/id` s parametrem `1` nebo `3`, aplikace se chová jako kdyby benchmark neexistoval, jelikož nemá uživatel právo tyto benchmarky získat. Odpověď je znázorněna v ukázce 4.7.

```
{
    "timestamp": "2019-05-01T14:12:12.256",
    "message": "Benchmark with ID 3 was not found.",
    "details": "uri=/api/benchmark/3"
}
```

Ukázka 4.7: Cizí benchmark

Podobné zabezpečení je vytvořeno i na endpointu `GET /api/users/id`, který slouží pro získání údajů o určitém uživateli. Pokud tento endpoint s ID svého účtu zavolá uživatel s rolí `USER` nebo `DEMO`, aplikace vrátí informace o uživatelově účtu. Pokud by se uživatel zeptal na účet s jiným ID než je jeho, aplikace mu vrátí odpověď v ukázce 4.8. Pokud endpoint zavolá uživatel s rolí `ADMIN`, odpověď obsahuje detail uživatele s daným ID.

```
{
    "timestamp": "2019-05-01T14:06:10.547",
    "message": "User with ID 1 was not found.",
    "details": "uri=/api/user/1"
}
```

Ukázka 4.8: Cizí účet

Speciální případy, kdy může jeden endpoint vracet data závisející na roli uživatele, jsou otestovány pomocí JUnit testů.

Validace REST API

Každý REST endpoint, který vyžaduje v těle požadavku zaslat objekt, je validován. Validace spočívá v kontrole, jestli objekt obsahuje všechny potřebné hodnoty. U číselných hodnot je dále kontrolováno, jestli je zadané číslo z povoleného rozsahu. Pokud například při registraci nového uživatele není zaslán atribut definující uživatelův e-mail, je vrácena odpověď se stavovým kódem `400`. Ukázka 4.9 obsahuje odpověď na nevalidní požadavek registrace uživatele.

```
{
    "timestamp": "2019-05-01T14:49:29.803",
    "message": "Validation Failed",
    "details": "org.springframework.validation.
       BeanPropertyBindingResult: 1 errors. Field error in object '
       userRegistrationForm' on field 'email': rejected value [null
       ]; codes [NotNull.userRegistrationForm.email,NotNull.email,
       NotNull.java.lang.String,NotNull]; arguments [org.
       springframework.context.support.
       DefaultMessageSourceResolvable: codes [userRegistrationForm.
       email,email]; arguments []; default message [email]];
       default message [nesmi byt null]]"
```

```
}
```

Ukázka 4.9: Nevalidní registrace uživatele

Tento typ chyby je vrácen v případě, že požadavek neprojde prvotní validací. Pokud objekt obsahuje všechny potřebné informace, požadavek začne být zpracováván. Nicméně v průběhu zpracování požadavku se může vyskytnout problém v další validaci. Při registraci nového uživatele může chtít uživatel zaregistrovat e-mail, který je ale již zaregistrován v aplikaci. V takovém případě aplikace informuje uživatele o existenci uživatele s požadovaným emailem. Stavový kód odpovědi je nastaven na `400 Bad Request`. Odpověď je zobrazena v ukázce 4.10.

```
{
    "timestamp": "2019-05-01T15:05:09.545",
    "message": "User with email user@mbmark.cz has already existed.
        ",
    "details": "uri=/api/user"
}
```

Ukázka 4.10: E-mail již existuje

Další situací, která může nastat v průběhu zpracování požadavku, je zobrazení konkrétního benchmarku. Aplikace přijme požadavek na vrácení informací o benchmarku s konkrétním ID. Pokud benchmark s daným ID existuje a uživatel má právo ho získat, aplikace vrátí v odpovědi daný benchmark. Jestliže ale benchmark se zadaným ID neexistuje, nebo na něj uživatel nemá právo, aplikace vrací uživateli odpověď se stavovým kódem `404` včetně informace o neexistenci benchmarku se zadaným ID. Ukázka 4.11 obsahuje odpověď na neexistující benchmark.

```
{
    "timestamp": "2019-05-11T09:55:53.986",
    "message": "Benchmark with ID 1 was not found.",
    "details": "uri=/api/benchmark/1"
}
```

Ukázka 4.11: Benchmark neexistuje

Nevalidní situace, které mohou nastat v průběhu zpracování požadavku, jsou otestovány pomocí JUnit testů.

5 ZÁVĚR

V rámci této práce byly prozkoumány různé nástroje pro provedení mikrobenchmarku v programovacím jazyce Java. Pomocí každého nástroje byl proveden mikrobenchmark měřící stejné fragmenty kódu. Nástroje Caliper od firmy Google a Java Microbenchmark Harness z projektu OpenJDK se jevily jako nejpoužitelnější dostupné frameworky pro psaní mikrobenchmarků. Na základě zvolených kritérií a porovnání nedostatků jednotlivých nástrojů byl vybrán Java Microbenchmark Harness. Ke zvolení tohoto nástroje pomohl i fakt, že je nástroj implementován do knihoven Javy ve verzi 12. Zároveň v rámci analýzy jednotlivých nástrojů nebyl nalezen téměř žádný nedostatek, který by komplikoval provedení mikrobenchmarku. Naopak nástroj Java Microbenchmark Harness nabízí mnoho parametrů pro konfiguraci mikrobenchmarku a dále nástroje pro sledování jeho běhu.

Cílem práce bylo vytvořit aplikaci, která porovná zadané fragmenty kódu a zjistí, jaký fragment kódu je rychlejší. Aby bylo možné fragmenty kódu porovnat, aplikace musela vygenerovat mikrobenchmark obsahující zadanou konfiguraci a fragmenty. Aplikace musela dále mikrobenchmark zkompilovat a následně spustit. Na závěr získat naměřené hodnoty jednotlivých fragmentů a ty zobrazit uživateli. Před samotnou implementací aplikace bylo potřeba rozhodnout, zda je uživatelsky přívětivější používat pro zadání a provedení mikrobenchmarku offline, nebo online aplikace. V dnešní době je běžné používat webové aplikace, jelikož uživatel nemusí instalovat žádné programy a další nástroje, protože si může aplikaci zobrazit v internetovém prohlížeči. Na základě zanalyzovaných výhod a nevýhod obou přístupů byla vybrána online aplikace. Online aplikace se skládá z frontend a backend části. Frontend aplikace obsahuje uživatelské rozhraní, pomocí kterého může uživatel pohodlně provádět mikrobenchmarky. O samotné provádění mikrobenchmarků a dalších akcí se stará backend aplikace. V rámci této práce byla implementována pouze backend aplikace. Frontend aplikace vznikala souběžně jako bakalářská práce.

Vytvořená aplikace obsahuje mnohem více funkcionality. Kromě provedení mikrobenchmarku nabízí správu již provedených mikrobenchmarků. Dále nabízí možnost přihlášení a správu uživatelů. S aplikací lze komunikovat skrze REST API a websockety.

Pro běh benchmarků je vhodné mít k dispozici izolované prostředí, kde běží pouze mikrobenchmark a žádné jiné procesy, které mohou ovlivnit běh mikrobenchmarku. Drobným nedostatkem implementovaného řešení je běh aplikace a spuštěných mikrobenchmarků na stejném serveru. Pokud je na serveru prováděn mikrobenchmark a zároveň aplikace obsluhuje další požadavky uživatelů, samotná aplikace může ovlivnit běh mikrobenchmarku. Tento problém by bylo možné vyřešit spouštěním Docker kontejnerů na samostatném serveru, kde by neběžely žádné jiné procesy. Nicméně pokud běží aplikace na vícejádrovém serveru, je možné tento problém zcela zanedbat. Dále je na zvážení, jestli by aplikace neměla mít omezen maximální počet mikrobenchmarků, které mohou běžet současně. Jiné

nedostatky implementovaného řešení nebyly nalezeny. Pro snadnější provádění mikrobenchmarků vznikla nad rámec zadání i jednoduchá frontend aplikace, která sloužila pro otestování správné funkčnosti backend aplikace.

LITERATURA

[1] Caliper. GitHub [obrázek]. Dostupné z: https://github.com/google/caliper/graphs/contributors. [cit. 2019-04-08].

[2] V. Dyuzhev. Hibernate: hbm2ddl.auto=update in production? [online]. Dostupné z: https://stackoverflow.com/a/221422, 2008-10-21. [cit. 2019-04-19].

[3] B. Goetz. Dynamic compilation and performance measurement [online]. Dostupné z: https://www.ibm.com/developerworks/java/library/j-jtp12214/, 2004-12-21. [cit. 2019-03-02].

[4] V. Hordějčuk. Jpa (java persistence api) [online]. Dostupné z: http://voho.eu/wiki/java-jpa, 2018. [cit. 2019-04-19].

[5] James. How to benchmark your pc: Best benchmarking software (free and paid) [online]. Dostupné z: http://blog.logicalincrements.com/2016/05/how-to-benchmark-your-pc/, 2016-05-02. [cit. 2019-03-02].

[6] JBenchX. GitHub [obrázek]. Dostupné z: https://github.com/iquadrat/jbenchx/graphs/contributors. [cit. 2019-04-08].

[7] J. Jenkov. Jmh - java microbenchmark harness [online]. Dostupné z: http://tutorials.jenkov.com/java-performance/jmh.html, 2015-09-16. [cit. 2019-04-14].

[8] F. Khatib. Jvm jit - loop unrolling [online]. Dostupné z: http://fasihkhatib.com/2018/05/20/JVM-JIT-Loop-Unrolling/, 2018-05-20. [cit. 2019-03-02].

[9] R. Lipka. Základy testování výkonnosti a spolehlivosti hw [online]. Dostupné z: https://portal.zcu.cz/CoursewarePortlets2/DownloadDokumentu?id=121914, 2017-11-14. [cit. 2019-03-02].

[10] F. Marvan. Není virtuál jako virtuál [online]. Dostupné z: https://diit.cz/clanek/virtualizace-na-urovni-operacniho-systemu, 2012-08-02. [cit. 2019-04-19].

[11] Oracle. Java virtual machine specification [online]. Dostupné z: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html#jvms-1.2. [cit. 2019-04-14].

[12] Oracle. The java hotspot performance engine architecture [online]. Dostupné z: https://www.oracle.com/technetwork/java/whitepaper-135217.html, 2013-06-19. [cit. 2019-04-09].

[13] A. S. (Oracle). Java microbenchmark harness (the lesser of two evils) [prezentace]. Dostupné z: https://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf. [cit. 2019-04-14].

[14] K. Peters. Performance measurement with jmh – java microbenchmark harness [online]. Dostupné z: https://blog.codecentric.de/en/2017/10/performance-measurement-with-jmh-java-microbenchmark-harness/, 2017-10-23. [cit. 2019-03-02].

[15] P. Software. Spring boot [online]. Dostupné z: https://spring.io/projects/spring-boot, 2019. [cit. 2019-04-16].

[16] G. Stokol and M. Young. Java garbage collection basics [online]. Dostupné z: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html, 2016-05-20. [cit. 2019-03-02].

[17] P. Venezia. Docker výrazně zjednodušuje pojetí virtualizace [online]. Dostupné z: https://computerworld.cz/internet-a-komunikace/docker-vyrazne-zjednodusuje-pojeti-virtualizace-51785, 2015-02-14. [cit. 2019-04-19].

SEZNAM OBRÁZKŮ

SEZNAM PŘÍLOH

SEZNAM ZKRATEK

JVM	Java Virtual Machine
JRE	Java Runtime Environment
JIT	Just-in-time
OSR	On-stack replacement
JMH	Java Microbenchmark Harness
ORM	Objektově relační mapování
JPA	Java Persistence API
REST	Representational State Transfer

A VÝSLEDKY BENCHMARKU

Níže jsou uvedeny naměřené časy zkušebního mikrobenchmarku. Tento mikrobenchmark sloužil jako otestování jednotlivých frameworků, zda poskytují podobné výsledky měření. Pro každý nástroj byl vytvořen mikrobenchmark testující stejný problém - otestování metody `constains()` nad kolekcemi `ArrayList` a `LinkedList`.

Výstup mikrobenchmarku nástroje JBenchX:

```
Running on Windows 10 10.0
Max heap = 3793747968 System Benchmark = 2,18ns
Performing 6 benchmarking tasks..
[0] ContainsBenchmark.arrayListContains(100)   137ns
[1] ContainsBenchmark.arrayListContains(50000)   60.9us
[2] ContainsBenchmark.arrayListContains(99999)   145us
[3] ContainsBenchmark.linkedListContains(100) 264ns
[4] ContainsBenchmark.linkedListContains(50000) 312us
[5] ContainsBenchmark.linkedListContains(99999) 838us
Success.
```

Ukázka A.1: Mikrobenchmark nástroje JBenchX

Výstup mikrobenchmarku nástroje Caliper:

SCENARIO.BENCHMARKSPEC.METHODNAME	SCENARIO.BENCHMARKSPEC.PARAMETERS.TESTEDVALUE	RUNTIME (NS)
arrayListContains	100	105.935
arrayListContains	50000	70,465.270
arrayListContains	99999	178,331.144
linkedListContains	100	259.106
linkedListContains	50000	276,935.522
linkedListContains	99999	677,577.273

Obr. A.1: Mikrobenchmark nástroje Caliper

Výstup mikrobenchmarku nástroje Java Microbenchmark Harness:

```
Benchmark                                (testedValue)  Mode  Cnt     Score
        Error  Units
ContainsTest.arrayListContains                     100  avgt   10     0,112
        0,005  us/op
ContainsTest.arrayListContains                   50000  avgt   10    66,707
        2,321  us/op
ContainsTest.arrayListContains                   99999  avgt   10   170,560
       28,853  us/op
ContainsTest.linkedListContains                    100  avgt   10     0,280
        0,037  us/op
ContainsTest.linkedListContains                  50000  avgt   10   294,344
       54,683  us/op
ContainsTest.linkedListContains                  99999  avgt   10   805,334
      103,484  us/op
```

Ukázka A.2: Mikrobenchmark nástroje Java Microbenchmark Harness

B IMPLEMENTACE APLIKACE

B.1 Použitý Dockerfile

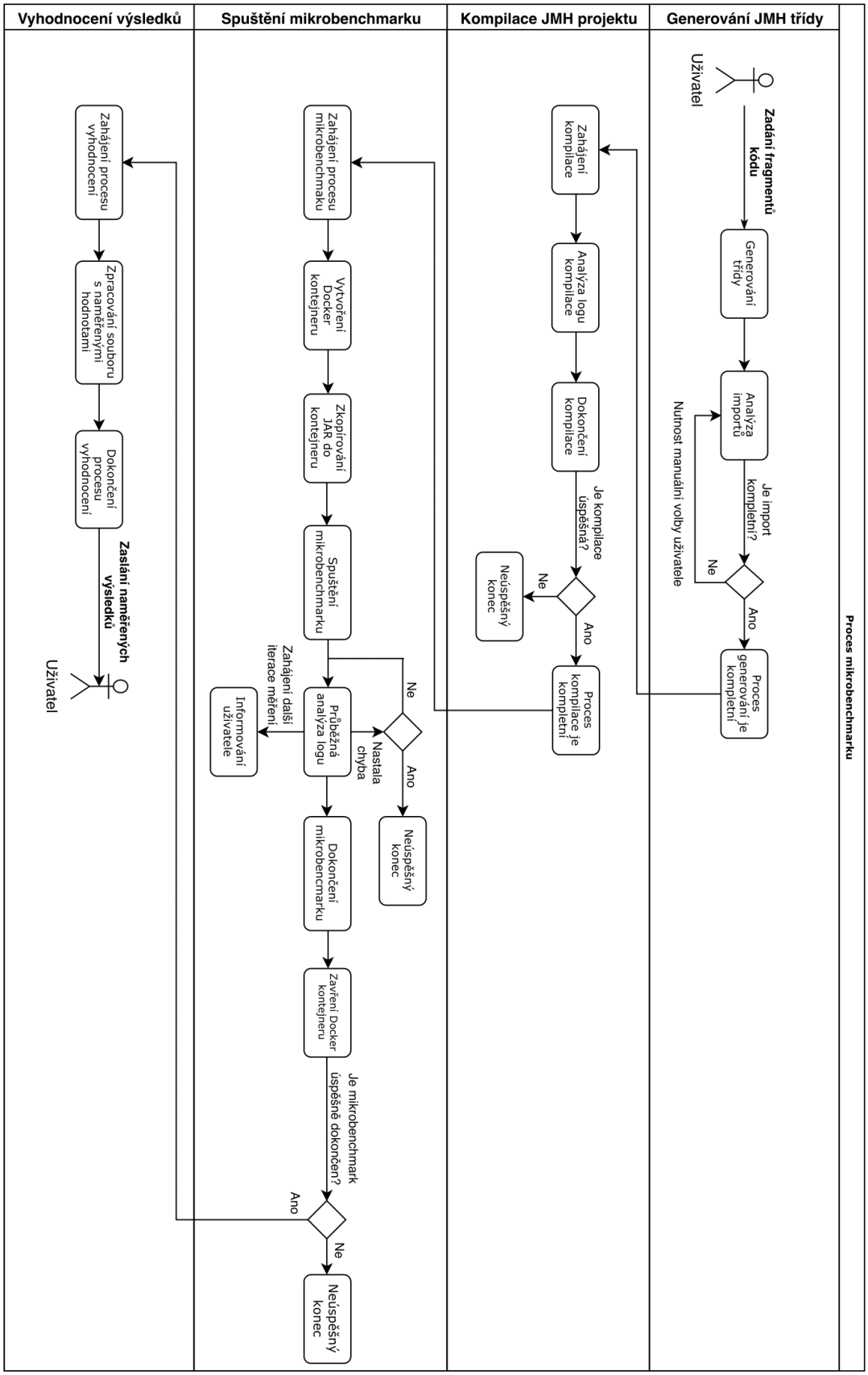
Každý Docker kontejner se vytváří z Docker image. Pro tuto práci je vytvořen následující image:

```
FROM alpine:latest

USER root

RUN \
  apk update && \
  apk upgrade && \
  apk add --update bash && \
  apk add openjdk8 && \
  rm -rf /var/cache/apk/* && \
  mkdir -p /benchmark

ENV JAVA_HOME /usr/lib/jvm/java-1.8-openjdk

WORKDIR /benchmark

CMD ["bash"]
```

Ukázka B.1: Použitý Docker image

B.2 Proces provedení mikrobenchmarku

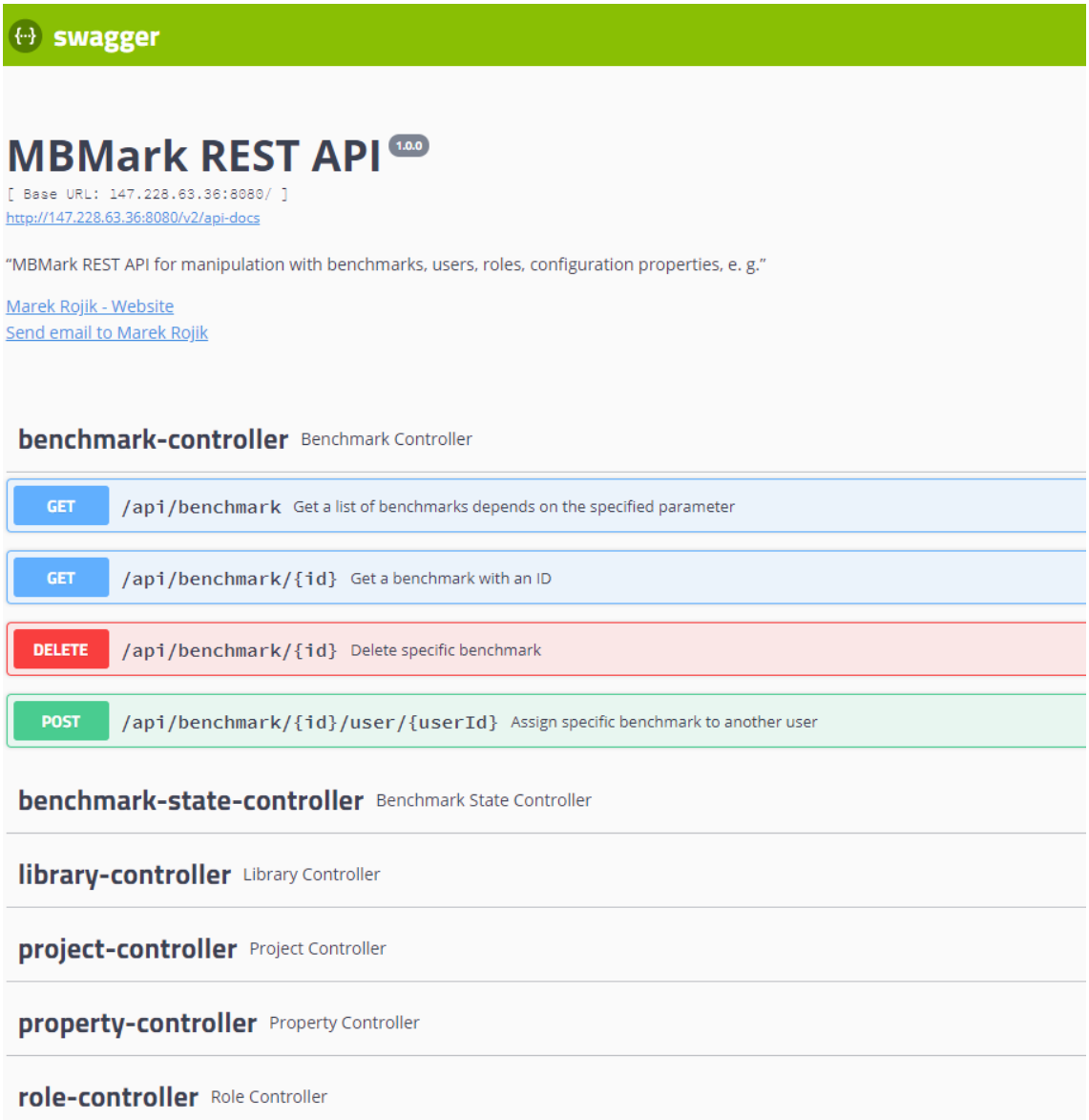
Obrázek B.1 zobrazuje kompletní proces pro provedení mikrobenchmarku.



Obr. B.1: Proces mikrobenchmarku

B.3 Swagger

Ukázka výstupu Swagger UI aplikace zobrazující dokumentaci k vytvořenému REST API. Swagger UI je dostupné na adrese `http://147.228.63.36:8080/swagger-ui.html`.



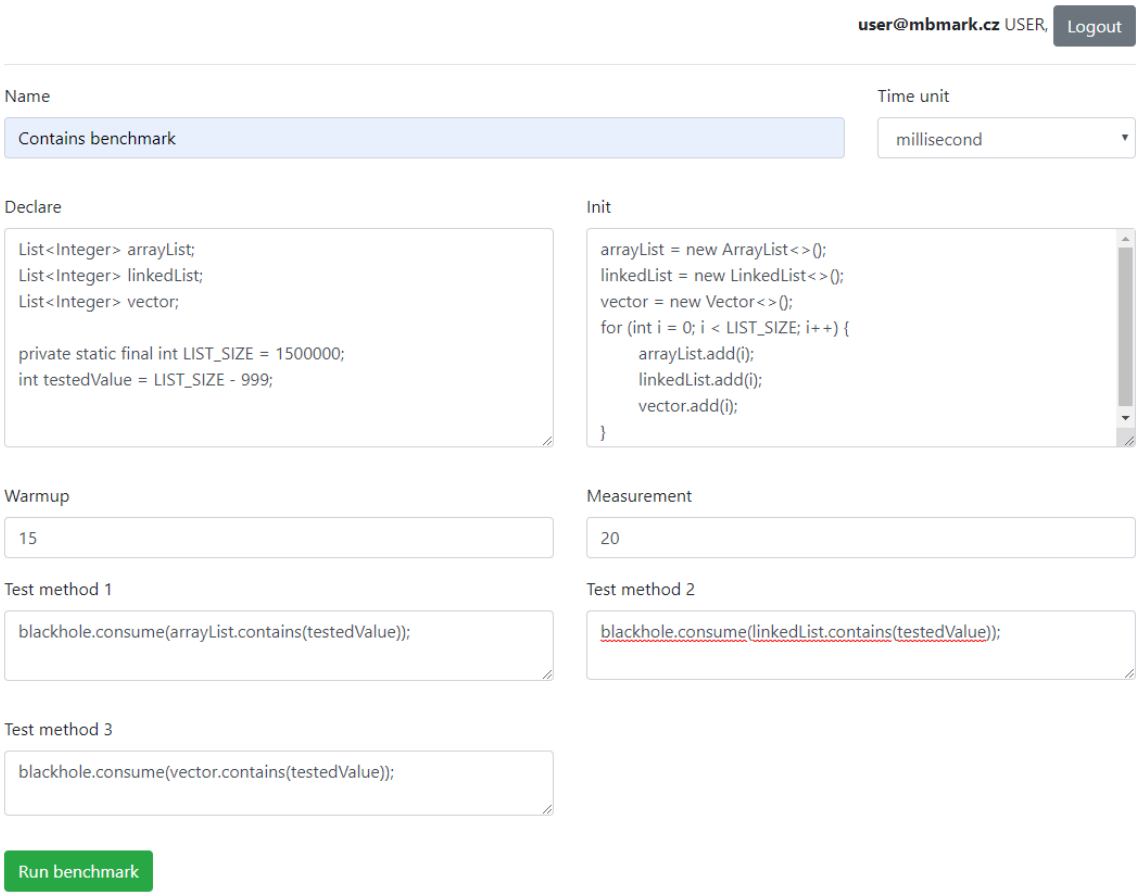
Obr. B.2: Ukázka aplikace Swagger

C TESTOVÁNÍ APLIKACE

C.1 Frontend aplikace

C.1.1 Zadání

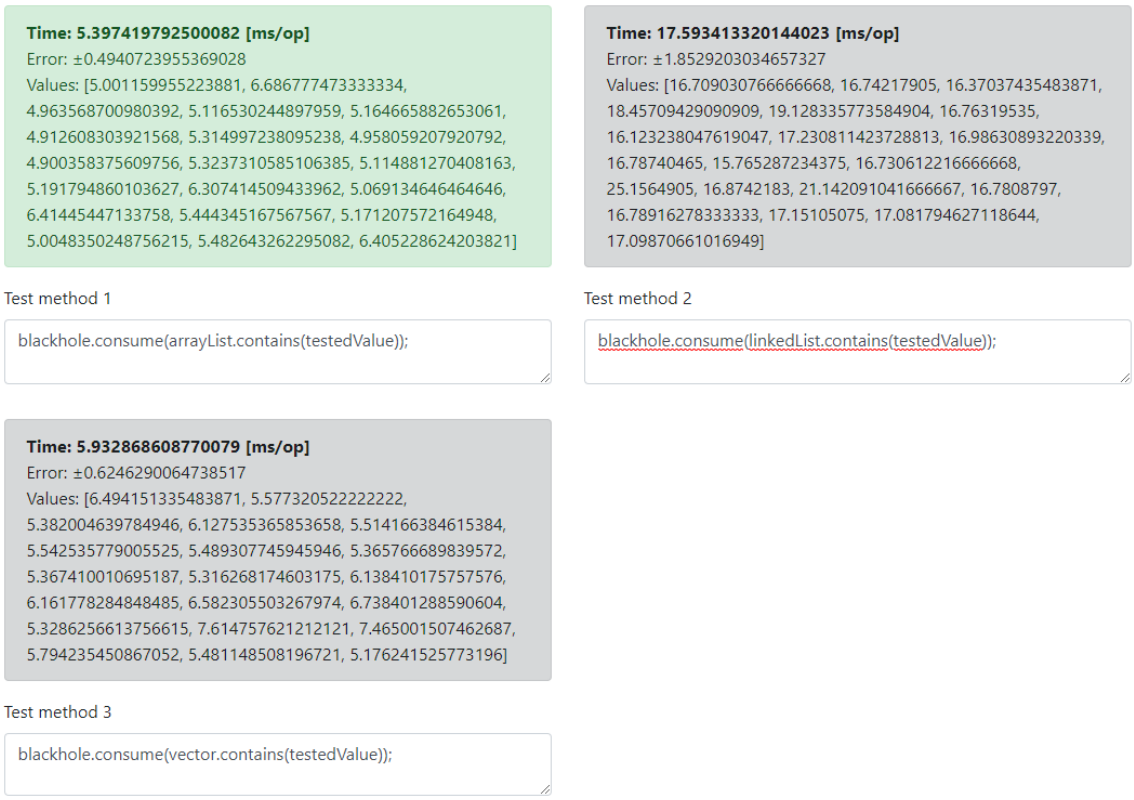
Obrázek C.1 zobrazuje část frontend aplikace, ve které se zadávají konfigurační hodnoty benchmarku a fragmenty kódu pro porovnání. Aplikace slouží pouze pro testování backend aplikace.



Obr. C.1: Ukázka frontend aplikace

C.1.2 Výsledek

Obrázek C.2 zobrazuje výsledek mikrobenchmarku z obrázku C.1.



Obr. C.2: Výsledek mikrobenchmarku

C.1.3 Získané informace

Obrázek C.3 ukazuje, jaké informace lze získat v průběhu vykonávání mikrobenchmarku. Benchmark je spuštěn s konfigurací warmup = 2 a measurement = 2. Benchmark porovnává dva fragmenty kódu.

Operation: START_CREATE_PROJECT

Operation: START_IMPORT_LIBRARIES

Operation: END_IMPORT_LIBRARIES, project id: 1dd702ad-67b5-4e95-b121-a8141a120f13

Operation: START_COMPILE, project id: 1dd702ad-67b5-4e95-b121-a8141a120f13

Operation: END_COMPILE, project id: 1dd702ad-67b5-4e95-b121-a8141a120f13

Operation: START_RUN, project id: 1dd702ad-67b5-4e95-b121-a8141a120f13

Time: 2019-04-23T20:59:34.523, estimated time: null, operation: WARMUP, iteration: 1/2

Time: 2019-04-23T20:59:36.145, estimated time: null, operation: WARMUP, iteration: 2/2

Time: 2019-04-23T20:59:37.771, estimated time: 2019-04-23T21:00:01.771, operation: MEASUREMENT, iteration: 1/2

Time: 2019-04-23T20:59:39.407, estimated time: 2019-04-23T21:00:01.771, operation: MEASUREMENT, iteration: 2/2

Time: 2019-04-23T20:59:40.444, estimated time: 2019-04-23T20:59:44.444, operation: RESULT, iteration: 1/2

Time: 2019-04-23T20:59:46.479, estimated time: 2019-04-23T20:59:58.480, operation: WARMUP, iteration: 1/2

Time: 2019-04-23T20:59:48.093, estimated time: 2019-04-23T20:59:58.480, operation: WARMUP, iteration: 2/2

Time: 2019-04-23T20:59:49.716, estimated time: 2019-04-23T20:59:51.716, operation: MEASUREMENT, iteration: 1/2

Time: 2019-04-23T20:59:51.343, estimated time: 2019-04-23T20:59:51.716, operation: MEASUREMENT, iteration: 2/2

Time: 2019-04-23T20:59:52.372, estimated time: 2019-04-23T20:59:52.373, operation: RESULT, iteration: 2/2

Time: 2019-04-23T20:59:52.813
Number of connection: 2
Benchmark 1: 0.0005241507924647072 [ms/op], error: -1, values: [0.0005481790321643613, 0.000500122552765053]
Benchmark 2: 0.0031274235189805923 [ms/op], error: -1, values: [0.003255186257534532, 0.0029996607804266525]

Obr. C.3: Získané informace v průběhu běhu

D UŽIVATELSKÁ DOKUMENTACE

V rámci diplomové práce běží aplikace na školním serveru `147.228.63.36` na portu `8080`. Server je dostupný pouze ze školní sítě. Dále běží testovací frontend aplikace na adrese `http://147.228.63.36/mbmark`.

V aplikaci jsou vytvořené 3 testovací účty:

1. účet s ADMIN rolí - email: `admin@mbmark.cz`, heslo: `admin`
2. účet s USER rolí - email: `user@mbmark.cz`, heslo: `user`
3. účet s DEMO rolí - email: `demo@mbmark.cz`, heslo: `demo`

D.1 Technologie

Aplikace je vyvíjena v programovacím jazyce Java `1.8`. V této verzi Javy jsou spouštěny vygenerované mikrobenchmarky pomocí frameworku JMH v Docker kontejneru. Framework Spring Boot je použit ve verzi `2.0.6`.

Aplikace potřebuje ke svému správnému běhu několik externích nástrojů. Jako databáze je použit PostgreSQL 9.6.10. Docker je nainstalován ve verzi `18.09.0`. Pro kompilaci aplikace a benchmarků je použit nástroj Maven v3.6.0. JMH je použit ve verzi 1.21.

Správné fungování aplikace je otestováno v operačním systému Windows 10 a Linux Debian 4.9.110-3+deb9u2. Všechny použité nástroje lze nainstalovat na oba uvedené operační systémy.

D.2 Rozběhnutí aplikace

D.2.1 Nastavení serveru

V této kapitole je popsán podrobný návod, jak nastavit linuxový server, aby na něm mohla aplikace bezproblémově běžet. Návod obsahuje přímo linuxové příkazy, které je nutné provádět v uvedeném pořadí. Všechny příkazy je nutné provádět s `root` uživatelem.

Obecné příkazy

Následující příkazy pouze aktualizují systém a nainstalují základní sadu nástrojů.

1. `sudo -i`
2. `apt-get update`
   (a) pokud nastane v průběhu update chyba, pravděpodobně tyto příkazy vyřeší problém
   (b) `apt-get install dirmngr`

   (c) `apt-key adv -keyserver keyserver.ubuntu.com -recv-keys C2518248EEA14886`
3. `apt-get install net-tools`

Java 1.8

Níže jsou napsány příkazy pro nainstalování Java 1.8.

1. `add-apt-repository ppa:openjdk-r/ppa`
2. `apt-get update`
3. `apt-get install openjdk-8-jdk`
4. `export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64/`
5. `export PATH=$PATH:$JAVA_HOME/bin`

Databáze

Níže jsou napsány příkazy pro nainstalování PostgreSQL databáze.

1. `apt-get install postgresql postgresql-contrib`
2. `sudo -u postgres psql`
3. DB dotazy pro vytvoření databáze a uživatele
   (a) `CREATE DATABASE mbmark;`
   (b) `CREATE USER mbmark WITH ENCRYPTED PASSWORD 'mbmark';`
   (c) `GRANT ALL PRIVILEGES ON DATABASE mbmark TO mbmark;`

Docker

Níže jsou napsány příkazy pro nainstalování Dockeru.

1. `apt-get update`
2. `apt-get install apt-transport-https ca-certificates curl gnupg2 software-properties-common`
3. `curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -`
4. `sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/debian $(lsb_release -cs) stable"`
5. `apt-get update`
6. `apt-get install docker-ce`
7. vytvoření Dockefile
   (a) `mkdir /var/docker`
   (b) `cd /var/docker`
   (c) `nano Dockerfile`

i. Dockerfile se nachází na přiloženém CD

(d) `docker build -f Dockerfile -t docker-microbenchmark .`

Maven

Níže jsou napsány příkazy pro nainstalování nástroje Maven.

1. `cd /tmp`
2. `apt-get install wget`
3. `wget https://bit.ly/2VDHsKZ -O apache-maven-3.6.0-bin.tar.gz`
4. `tar xzvf apache-maven-3.6.0-bin.tar.gz`
5. `mv apache-maven-3.6.0 /opt/apache-maven-3.6.0`
6. `export PATH=/opt/apache-maven-3.6.0/bin:$PATH`
7. `export M2_HOME=/opt/apache-maven-3.6.0`

Apache

Níže jsou napsány příkazy pro nainstalování nástroje Apache.

1. `apt-get install apache2`
2. `nano /etc/apache2/ports.conf`
   (a) `Listen 147.228.63.36:80` (jedná se o IP adresu serveru)
3. `nano /etc/apache2/sites-available/000-default.conf`
   (a) `<VirtualHost *:80>`
4. `/etc/init.d/apache2 restart`
5. `mkdir -p /var/mbmark/projects`
6. `ln -s /var/mbmark/projects /var/www/html/projects`
7. `chmod 755 /var/mbmark/projects/`

Firewall

Níže jsou napsány příkazy pro povolení portů ve firewallu. Jedná se o port `80` (Apache) a `8080` (backend aplikace).

1. `/sbin/iptables-save > /etc/network/iptables`
2. `nano /etc/network/iptables`
   (a) přidat následující pravidla
   (b) `-A INPUT -m tcp -p tcp --dport 80 -s 147.228.0.0/16 -j ACCEPT`
   (c) `-A INPUT -m tcp -p tcp --dport 80 -j DROP`
   (d) `-A INPUT -m tcp -p tcp --dport 8080 -s 147.228.0.0/16 -j ACCEPT`
   (e) `-A INPUT -m tcp -p tcp --dport 8080 -j DROP`

3. `/sbin/iptables-restore < /etc/network/iptables`

Vytvoření adresářů

Aby měla aplikace kam ukládat logy a získané výsledky, je nezbytné vytvořit složky na disku.

1. `mkdir -p /var/mbmark/logs`
2. `mkdir -p /var/mbmark/results`

D.2.2 Kompilace a spuštění aplikace

Kompletní zdrojové kódy jsou přiloženy na CD ve složce `backend`. V této složce se nachází jednotlivé moduly a soubor `pom.xml`.

Kompilace

Pro zkompilování aplikace je potřeba ve složce `backend` otevřít příkazovou řádku a provést příkaz `mvn clean install`. V průběhu kompilace jsou provedeny všechny JUnit testy. Pokud doběhne kompilace úspěšně, ve složce `microbenchmark-web/target` je vygenerován soubor `MBMark_application.jar`.

Spuštění

Spring Boot aplikace v sobě obsahuje Tomcat server. Díky této vlastnosti stačí spustit vygenerovaný `JAR` soubor a backend aplikace se spustí. Aplikaci je nutné spustit pod `root` uživatelem.

Pro spuštění aplikace slouží bashový skript `run.sh` ve složce `backend`. Skript lze spustit pouze v operačním systému Linux. Ve skriptu se nachází příkaz `java` s parametry pro správné spuštění aplikace. Příkaz obsahuje parametr pro nastavení portu a Maven profilu definující např. cestu pro ukládání vygenerovaných projektů. Dále se zde nachází parametr pro zabezpečení aplikace. Jako předposlední parametr je uvedena cesta k `JAR` `souboru`. Jako poslední je znak `&`, pomocí kterého je zajištěno, aby aplikace běžela i po ukončení terminálu. Před provedením příkazu `java` je provedena kontrola, jestli je v systému nainstalován Docker a PostgreSQL databáze.

Výchozí uživatel

Po prvním spuštění je automaticky vytvořen uživatel s rolí `ADMIN`. Uživatel má nastavený e-mail *admin@rojik.cz* a heslo *78X2Dup4O9nuoCEg*. Uživatel je vytvořen pomocí Liquibase skriptu v adresáři `microbenchmark-web/src/main/resources/db/changelog/`

`insert_data.xml`. Pokud chcete změnit výchozího uživatele, stačí editovat tento soubor a zkompilovat aplikaci. Heslo musí být zahashováno funkcí Bcrypt.

D.2.3 Konfigurace aplikace

V jednotlivých kapitolách je popsáno, jakým způsobem lze konfigurovat aplikaci.

Databáze a e-mail

Ke spouštění backend aplikace je nutné nastavit správné údaje pro připojení k databázi. Databáze musí běžet před spuštěním backend aplikace.

Aby bylo možné novému zaregistrovanému uživateli poslat informační e-mail, je potřeba nakonfigurovat připojení k SMTP serveru.

Soubor `microbenchmark-web/src/main/resources/application.properties` obsahuje právě konfiguraci pro připojení k databázi a SMTP serveru. Hodnoty s prefixem `spring.datasource` slouží k nastavení připojení k databázi. Hodnoty s `spring.mail` nastavují připojení k SMTP serveru.

Pokud dojde ke změně těchto konfigurací, je nutné aplikaci znovu zkompilovat a spustit.

Konfigurace pomocí REST API

Některé konfigurace aplikace se nacházejí přímo v databázi. Jedná se o konfigurační hodnoty, které se nastavují skrze REST API `/api/property`. Provedením požadavku `GET /api/property` administrátor aplikace zjistí, jaké konfigurační hodnoty jsou uloženy v databázi. Pokud konfigurace nabývá hodnoty `null`, znamená to, že konfigurační hodnota není manuálně nastavena a aplikace používá výchozí hodnotu.

Konfigurace `libraries` a `ignoreClasses` se používají pro automatický import knihoven v rámci fáze generování benchmarku. Výchozí hodnoty čte aplikace ze souborů.

Konfigurace `jmhVersion` a `javaVersion` značí verze JMH a Javy, které se používají pro benchmark. Výchozí hodnota pro JMH je `1.21` a pro Javu `1.8`.

Konfigurace `blindCopyEmail` slouží pro definování e-mailu, na který mají chodit skryté kopie všech poslaných e-mailů. Aplikace nemá nastavený žádný výchozí e-mail.

Poslední `maxMemory` definuje maximální velikost paměti, kterou může běžící benchmark využít. Výchozí hodnota je nastavena na 512 MB.

D.2.4 REST API

REST API je zdokumentováno pomocí nástroje Swagger. Swagger UI je dostupné na adrese `http://147.228.63.36:8080/swagger-ui.html`. Nicméně jeden endpoint Swagger nezdokumentoval. Jedná se o endpoint sloužící pro přihlášení uživatele.

Pro přihlášení je potřeba poslat HTTP požadavek typu `POST` na adresu `/api/login`. V těle požadavku je uveden e-mail a heslo uživatele. Ukázka D.1 obsahuje příklad těla požadavku.

```
{
  "email": "user@mbmark.cz",
  "password": "user"
}
```

Ukázka D.1: Přihlášení uživatele

Po úspěšném přihlášení je v odpovědi zaslán token, pomocí kterého uživatel komunikuje se serverem.

D.2.5 Websocket

Ve Swaggeru dále chybí informace o navázání websocketového spojení, adresa pro zaslání zprávy a adresa, na kterou server zasílá informační zprávy.

Pro navázání spojení slouží adresa `/socket/websocket`. Spuštění benchmarku se zahájí posláním zprávy na adresu `/app/benchmark/run` s parametrem ID projektu. Pokud je uživatel přihlášen, součástí hlavičky zprávy by měl být i autorizační token. Průběžné informace o aktuální prováděné fázi benchmarku lze získávat odposloucháváním adresy `/user/benchmark/result/step`. Po úspěšném či neúspěšném provedení benchmarku jsou výsledky dostupné na adrese `/user/benchmark/result`.

D.2.6 Frontend aplikace

Pro otestování a pohodlnější provádění testovacích benchmarků je vytvořena jednoduchá frontend aplikace ve frameworku Angular 5. Aplikace je přiložena v adresáři `frontend`. V implementaci aplikaci lze případně najít, jaké všechny HTTP požadavky nebo websocketové zprávy je nutné poslat pro správné provedení benchmarku.

Tato aplikace běží na adrese `http://147.228.63.36/mbmark`.

D.2.7 Aktualizace verze Javy

Benchmarky jsou generovány ve verzi Javy 1.8. Pokud by v budoucnu bylo potřeba použít novější verzi Javy, je potřeba provést několik kroků.

Jako první je potřeba změnit konfigurační hodnotu `javaVersion` skrze REST API `PUT /api/property`. V těle požadavku se uvede jako klíč `javaVersion` a hodnota nová verze Javy.

Dále je nutné vytvořit novou mapu tříd, která se používá pro automatický import. K tomu slouží endpoind `PUT /api/library`. V těle požadavku je uveden jediný atribut. Jedná se o uvedení absolutní cesty k adresáři, ve kterém se nacházejí `JAR` soubory k proskenování a nalezení všech dostupných tříd Javy.

Pro správnou kompilaci benchmarku nástrojem Maven je potřeba mít správně nastavenou systémovou proměnnou `JAVA_HOME`. Proměnná musí obsahovat cestu do adresáře nové verze Javy.

Jako poslední je nezbytné upravit současný Dockerfile. V definici Dockerfile je uvedena Java, která má být ve spuštěném kontejneru nainstalována (`apk add openjdk8`). V tomto příkazu je potřeba vyměnit balíček `openjdk8` za novější. Upravený Dockerfile se následně musí zkompilovat, aby nahradil současnou verzi obrazu.

Všechny tyto kroky je nutné provést pro použití novější verze Javy. Tyto kroky nevyžadují restartování běžící backend aplikace.